

Naformátováno: Zdůraznění – jemné, Písmo: není Tučné



ALGORITMY IAL

Studijní opora

Autor verze: Prof. Ing.Jan M Honzík, CSc.
Verze:17-B5-A

Obsah		
1. Předmluva		3
1.1. Organizační informace		3
1.2. Metodické informace		4
2. Úvod do datových struktur a algoritmů		8
2.1. Algoritrický jazyk		8
2.2. Datové typy a struktury		10
2.3. Příkazy		18
2.4. Základní pojmy složitosti algoritmů		26
2.5. Otázky a úkoly		29
3. Abstraktní datové typy		31
3.1. Principy dynamického přidělování paměti		31
3.2. Abstraktní datový typ, základní pojmy		40
3.2.1. Seznamy		43
3.2.2. Zásobník		60
3.2.3. Fronta		65
3.2.4. Vyhledávací tabulka		70
3.2.5. Pole		72
3.2.6. Graf		75
3.2.7. Strom		79
3.2.8. Jiné ADT		89
3.2.9. Kontrolní otázky		89
3.2.10. Kontrolní příklady		91
4. Vyhledávání		92
4.1. Sekvenční vyhledávání v poli		95
4.2. Rychlé sekvenční vyhledávání		97
4.3. Sekvenční vyhledávání v seřazeném poli		98
4.4. Sekvenční vyhledávání s adaptivní rekonfigurací pole podle četnosti		98
4.5. Binární vyhledávání v seřazeném poli		99
4.6. Dijkstrova varianta binárního vyhledávání		100
4.7. Uniformní a Fibonacciho vyhledávání		102
4.8. Binární vyhledávací strom - BVS		108
4.9. Binární vyhledávací strom se zarážkou		117
4.10. Bínární vyhledávací strom se zpětnými ukazateli		119
4.11. Kontrolní otázky a úlohy		121
4.12. AVL strom		122
4.13. Tabulky s rozptýlenými položkami		133
5. Řazení		140
5.1. Úvodní pojmy a principy		140
5.1.1. Řazení podle více klíčů		141
5.1.2. Řazení bez přesunu položek		143
5.2. Řazení na principu výběru		147
5.2.1. Bublinové řazení na principu výběru		148
5.2.2. Heap sort - "řazení hromadou"		149
5.3. Řazení na principu vkládání		152
5.4. Quick sort - řazení rozdělováním		155
5.5. Shell sort		158
5.6. Řazení setřídováním		159
5.7. Merge sort		160
5.8. List Merge Sort		162
5.9. Radix sort		164

5.10. Sekvenční řazení	166
6. Vyhledávání v textu	171
6.1. Knuth-Morris-Prattův algoritmus	172
6.2. Boyer Mooreův algoritmus	175
7. Rekurze	180
7.1. Hilbertovy křivky	183
7.2. Algoritmy s návratem - "Cesta koně" a "Osm dam"	185
7.3. Hanojské věže	187
8. Dokazování správnosti programu	189
8.1. Správnost algoritmu indukcí	193
8.2. Dikstrova tvorba dokázaných programů	195
8.2.1. Základní matematický aparát	196
8.2.2. Definice základních mechanizmů	197
8.2.3. Definice složených příkazů	198
8.2.4. Teorém alternativního příkazu "IF"	201
8.2.5. Teorém invariance pro repetiční příkaz "DO"	201
8.2.6. Příklady	203
9. Přílohy	209
9.1. Příklady operací nad seznamy	209
9.1.1. Jednosměrný seznam	209
9.1.2. Dvojsměrný seznam	228
9.1.3. Kruhový seznam	240
9.2. Příklady operací nad binárními stromy	247
9.3. Příklad protokolu operace Insert pro AVL strom	251
9.4. Studentský projekt v jazyce C pro operaci insert nad AVL stromem	256

1. Předmluva

1.1 Organizační informace

Tato studijní opora je pomocný učební text pro stejnojmenný studijní předmět. Je zdrojem minimálního rozsahu znalostí potřebných pro úspěšné absolvování předmětu. Je psána strohým a věcným až heslovitým stylem a nenahrazuje knižní učebnici.



Předmět sestává z přednášek, domácích úloh řešených na počítači v nerozvrhovaných laboratořích, z projektu obhajovaného v závěru semestru. Písemná půlsezestrální zkouška seznamuje v polovině semestru se stylem závěrečné zkoušky a je oboustrannou diagnostickou informací pro studenty i učitele. Domácí úlohy a projekt jsou termínované. Nesplnění termínu vede ke ztrátě získatelných bodů. Podrobné informace o předmětu lze najít na internetové adrese <https://www.fit.vutbr.cz/study/courses/IAL/private/>, ke které mají přístup jen zapsaní studenti FIT/FEKT.

Zkouška



Podmínkou úspěšného absolvování předmětu je získání nejméně 50 ze 100 možných bodů. Závěrečnou písemnou zkoušku smí psát jen student, který získal v průběhu semestru alespoň 20 z 50 možných bodů. Přednášející a učitelé mají právo ocenit mimořádnou aktivitu studenta prémiovými body. Prémie není nároková. Pravomoci udělování prémiových bodů stanovuje přednášející. Klasifikace úspěšných studentů se řídí studijními předpisy FIT VUT a zásadami

ECTS (Evropského kreditového systému).

Účast na přednáškách není povinná. Zkušenosti přesvědčivě ukazují, že mezi nejčastější příčiny neúspěchu patří nedostatek orientace v základních požadavcích i v látce předmětu způsobené výraznou neúčastí na přednáškách.

Body



Za jednotlivé aktivity lze v průběhu semestru získat tyto body:

- dvě domácí úlohy po 10 bodech 20 b
- půlsemestrální zkouška 15 b
- společný projekt (prezentace 10, dokumentace 5) 15 b
- semestrální zkouška 50 b

Půlsemestrální zkouška nemá náhradní termín. V případě hodného zvláštního zřetele má přednášející právo zadat doplňkový projekt v rozsahu 10 b k částečné kompenzaci nezaviněné ztráty bodů při půlsemestrální zkoušce.

Termíny odevzdávání domácích úloh a projektů jsou závazné. Jejich nesplnění vede ke ztrátě bodového hodnocení.

Odhad časové náročnosti



Přibližný odhad časové náročnosti předmětu lze stanovit takto:

1 kredit = 25-30 hodin práce

7 kreditů odpovídá min. 175 hod. studijní práce, z toho:

- přednášky 40 hod
- 2 domácí úlohy 30 hod
- práce na projektu 40 hod
- průběžné studium 35 hod
- příprava na půlsem. a záv. zk. 30 hod

1.2 Metodické informace



Předmět Algoritmy je nadstavbou nad základy programování a užití některého z programovacích jazyků. Představuje "femeslný" základ každého budoucího profesionálního programátora a odborníka v oblasti informačních technologií. Předmět Algoritmy prošel více než 30 letým vývojem. Byl vytvořen z principů publikovaných světovými autory jako Niela Wirth a Donald Knuth, Edsgar Dijkstra nebo C.A.R.Hoare, jejichž publikace zůstanou ještě dlouho pilíří této disciplíny. Hlavní studijní literaturou byla skripta "Programovací techniky", nyní dostupná v redukované podobě pod názvem "Vybrané kapitoly z programovacích technik" [1]. Kapitoly "Vyhledávání" a "Řazení" této publikace zůstávají stále aktuální a proto bude text opory v této části stručnější. Poslední úprava obsahu předmětu vyplývá ze zavedení tříletého Bc. studijního programu. Původně dominantní orientace na jazyk Pascal ustoupila v oblasti vlastní aplikace a realizace programů současně vyučovanému jazyku. V oblasti výkladu algoritmů se používá algoritický jazyk na bázi podmnožiny Pascalu nejen pro vyšší srozumitelnost zápisu, ale také proto, že základy pascalovské kultury patří k nezbytné výbavě profesionálů v IT, na kterou navazuje řada dalších předmětů v bakalářském i magisterském studiu. IAL tak zůstává jediným předmětem, který nezbytnou úroveň těchto znalostí zajišťuje. Studenti budou své domácí úlohy a projekty realizovat v jazyce C. Tím prokáže porozumění probíraným algoritmům a prohloubí si aplikační schopnosti produkčního jazyka.

Odborná terminologie

Přesné myšlení, které je nástrojem inženýrské práce, je možné jen ve spojení s přesnými pojmy a správnou terminologií. Každý obor si vytváří svůj odborný



žargon, který je živým komunikačním nástrojem každodenního života. Žargon však není vhodný ani pro publikační ani pro vyspělé prezentacní aktivity. Učebnice a přednášky dobrých učitelů by měly být příkladem studentům. Každý pojem ze "slangového" světa, který učitel, nejčastěji v ústím projevu použije (a je pro to řada dobrých důvodů), musí být uveden jako "slangový". **Vyhýbejte se žoviálnímu a slangovému vyjadřování ve všech písemných projevech!**

Početná masově vydávaná a často popularizační počítačová literatura i internetové zdroje nejsou vždy zářným příkladem a v mnohých lze nalézt nepřesné, nevhodné i pokleslé pojmy i obraty, často spojené s nevhodným počešťováním anglických konstrukcí nebo špatným překladem.

Často se v našem oboru dělají v české terminologii chyby ve slově

"standardní" (nikoli *standartní*) a ve slově "**řídící**" - určeno k řízení (nikoliv *řídíci* - něco co právě teď řídí ; viz rozdíl mezi *konví kropicí* - na kropení a *kropicí* - právě zavlažující záhon).



Anglická terminologie

V textu se vyskytnou některé anglické termíny. Budou v souvislosti s vysvětlivkou nebo za českým pojmem v závorce a vždy budou zobrazeny *kurzívou*.

Grafická úprava

Text bude uveden standardním typem Times New Roman. Významné pojmy nebo části textu budou zvýrazněny **tučně** a/nebo **podtrženě**, v některých případech v "uvozovkách". Ty části programových úseků u nichž to bude významné, budou s ohledem na odsazování uvedeny typem **Courier**. Doplňující poznámky budou psány menším typem.

Prerekvizity



Předmět předpokládá ovládnutí základů programování (předmět IZP 1.r. BC studia) v některém z často užívaných programovacích jazyků. V rámci předmětu se předpokládá využití jazyka C k implementaci a realizaci programů demonstруjících správnost probíraných algoritmů.

Učební cíle a kompetence - specifické



Seznámit se s principy metod dokazování správnosti programů a s tvorbou dokázaných programů. Seznámit se základními principy složitosti algoritmů. Seznámit se s principy dynamického přidělování paměti. Seznámit se základními abstraktními datovými typy a strukturami, naučit se je implementovat a používat. Naučit se rekurzívni a nerekurzívni zápis základních algoritmů. Naučit se vytváret a analyzovat algoritmy vyhledávání a řazení. Osvojit si zásady dobrého programovacího stylu.

Učební cíle a kompetence - generické



Seznámit se pasivně s algoritmickým jazykem pascalovského typu. Naučit se základům týmové práce při tvorbě malého projektu. Naučit se základům tvorby dokumentace projektu. Naučit se základům prezentace projektu a obhajobě dosažených výsledků. Seznámit se se základy anglické terminologie v oblasti algoritmizace.

Anotace

Přehled základních datových struktur a jejich použití. Principy dynamického přidělování paměti. Specifikace abstraktních datových typů (ADT). Specifikace a implementace ADT: seznamy, zásobník, fronta, pole, vyhledávací tabulka, graf, binární strom. Algoritmy nad binárním stromem. Vyhledávání: sekvenční, v nesřazeném a seřazeném poli, vyhledávání se zarážkou, binární vyhledávání,

binární vyhledávací strom, vyvážený strom (AVL). Vyhledávání v tabulkách s rozptýlenými položkami. Řazení, principy, řazení bez přesunu položek, řazení podle více klíčů. Nejznámější metody řazení: Select-sort, Bubble-sort, Heap-sort, Insert-sort a jeho varianty, Shell-sort, Quick sort v rekurzívní a nerekurzívní notaci, Merge-sort, List-merge-sort, Radix-sort. Sekvenční metody řazení. Rekurze a algoritmy s návratem. Vyhledávání podřetězců v textu. Dokazování programů, tvorba dokázaných programů.

Přibližný rozpis přednášek

1. Algoritmický jazyk. Přehled datových struktur. Abstraktní datový typ a jeho specifikace.
2. Specifikace, implementace a použití ADT seznam.
3. Specifikace, implementace a použití ADT zásobník, fronta. Vyčíslení výrazů s použitím zásobníku.
4. ADT pole, množina, graf, binární strom.
5. Algoritmy nad binárním stromem.
6. Vyhledávání, sekvenční, v poli, binární vyhledávání.
7. Binární vyhledávací stromy, AVL strom.
8. Vyhledávání v tabulkách s rozptýlenými položkami.
9. Řazení, principy, bez přesunu, s vícenásobným klíčem.
10. Známé metody řazení polí
11. Známé metody řazení polí, řazení souborů.
12. Rekurze, algoritmy s návratem.
13. Dokazování správnosti programů, tvorba dokázaných programů.

Studijní literatura



1. Honzík, J., Hruška, T., Máčel, M.: Vybrané kapitoly z programovacích technik, Ed.stř.VUT Brno,1991. (Text skript je studentům FIT/FEKT dostupný ve formátu pdf na <https://www.fit.vutbr.cz/study/courses/IAL/private/>)
2. Knuth, D.: The Art of Computer programming, Vol.1,2,3. Addison Wesley, 1968
3. Wirth, N.: Algorithms+Data Structures=Programs. Prentice Hall, 1976
4. Cormen, T.H., Leiserson, Ch.E., Rivest, R.L.: Introduction to Algorithms. McGraw Hill, 1990.
5. Kruse, R.L.: Data Structures and Program Design. Prentice- Hall, Inc. 1984
6. Baase, S.: Computer Algorithms - Introduction to Design and Analysis. Addison Wesley, 1998
7. Sedgewick, R.: Algoritmy v C. (Základy. Datové struktury. Třídění. Vyhledávání. Addison Wesley 1998. Softpress 2003.
8. Dr.Dobbs CD-ROM: Algorithms and Data Structures. (CD ROM s 10 knižními publikacemi z oblasti algoritmů. Možno zapůjčit v knihovně FIT VUT v Brně).
9. Kučera Luděk: Algovize aneb procházka Krajinou algoritmů, Univerzita Karlova 2009, ISBN 978-80-902938-5-4; www.algovision.org

Poznámka

Text studijní opory vznikal v krátkém časovém období s využitím dříve vytvořených elektronických zdrojových textů a častým použitím techniky "cut and paste". To může být častou příčinou chybých konstrukcí a překlepů. Autor bude vděčný čtenářům za elektronické zprávy o chybách. Posílejte je na adresu: honzik@fit.vutbr.cz ve formátu:

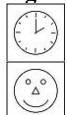
<příjmení a jméno odesilatele zprávy>, <elektronická adresa odesilatele>

(<cílo chyby ve zprávě>) <cílo stránky>/<cílo řádku - (+shora nebo -zdola)>/ <vadný text> → <správný text>

nebo

<cílo stránky>/<cílo řádku - +shora nebo -zdola>/ <popis chyby>

2 Úvod do datových struktur a algoritmů



Cílem kapitoly 2 je uvést algoritický jazyk - "pseudojazyk" pascalovského typu, který bude sloužit jako nástroj formálního popisu algoritmů.

K přečtení kapitoly vystačí pro čtenáře, který dobře zná některý imperativní, procedurálně zaměřený jazyk (jako je jazyk C) 5 hodin.

Pro čtenáře, který zná dobře Pascal stačí k zopakování 1 hodina

2.1 Algoritmický jazyk

2.1 Algoritmický jazyk je tvořen podmnožinou standardního jazyka Pascal, rozšířenou o některé praktické pomůcky a konvence. Algoritmický jazyk není určen k zápisu programu pro konkrétní počítač. Slouží k účelnému a srozumitelnému zápisu algoritmů. Má charakter generického (nikoliv interpretačního, nebo skriptovacího) jazyka, tzn. že má vlastnosti jazyka, který se ve fázi překladu přeloží do kódu počítače a ve fázi výpočtu se výsledný produkt spustí k výpočtu.

Pozn. Studenti předmětu IAL nebudou zkoušeni z aktivního použití algoritmického jazyka při vytváření rozsáhlejších programů. Měli by však ovládnout jeho plné pasivní porozumění a tvorbu elementárních algoritmických konstrukcí.

Příklady uváděné ve studijní opoře budou mít tvar procedur, funkcí, nebo úseků kódu ve tvaru sekvence příkazů, bez operací pro vstup a výstup. Zápis algoritmů sestává z pseudopříkazů a příkazů. Pseudopříkazy a příkazy se oddělují **středníkem**, který zapisujeme za předcházející (pseudo)příkaz

Syntax

Pojmem **syntax** označujeme soubor pravidel, která určují správnost jazykové konstrukce.

Sémantika

Pojmem **sémantika** označujeme popis účinků, které jsou výsledkem provedení jazykové konstrukce v době výpočtu.

Pseudopříkaz

Pseudopříkaz je pokyn k akci, která se provádí v době překladu. Pro algoritmický jazyk má pomocný, informativní význam. Pseudopříkazy jsou uvedeny v úvodní sekci zápisu programu nebo procedury/funkce (definice konstant, specifikace typů, deklarace proměnných a procedur). V učebním textu budou pseudopříkazy ve formě specifikací typů (deklarací proměnných) předcházet ukázky algoritmů v podobě procedur a funkcí. Některé často se vyskytující typy budeme používat bez jejich opakování specifikace.

Příkaz

Příkaz je pokyn k akci, která se provádí v průběhu chodu programu. Pro zápis algoritmů je základním stavebním kamenem algoritmu. Sekvenci příkazů uzavřené mezi příkazové závorky **begin** a **end** říkáme **složený příkaz**. Složený příkaz má význam jako jeden příkaz. (Příkazové závorky "zabalí" několik příkazů a tím vytvoří jeden příkaz). Středník před end v posloupnosti příkazů obvykle nepíšeme. Interpretuje se jako prázdný příkaz mezi středníkem a end. V některých příkladech budeme příkaz označovat písmenem **S**, podle anglického **statement**.

Pozn. Středník se nepoužívá k ukončení příkazu, ale k oddělení dvou po sobě jdoucích příkazů. Proto za příkaz, který je v textu uvedený jako ukázka na jednom rádku, středník psát nemusíme.

Identifikátor Identifikátory sestávají obvykle z písmen a číslic. Velikost znaku neodlišuje dva různé

identifikátory. Ačkoliv to nemá zvláštní význam, nebudeme pro identifikátory používat diakritiku.

Komentář	Komentáře jsou uzavřeny do závorek (* a *). Příklad: (* mezi hvězdičkami je komentář *). Komentáře mohou být zanořené. Použití diakritiky v komentáři není neobvyklé. S ohledem na úsporu prostoru na rádku nemusí být komentáře zapsány typem Courier. Dobrý programovací styl vede k použití stručných ale výstižných komentářů vysvětlujících zejména význam procedur/funkcí, jejich parametrů a také význam složených příkazů, nebo významnějších příkazových sekvencí. Dobrý program by měl být čitelný na základě komentářů i kompetentnímu čtenáři, který dobře nezná použitý programovací jazyk.
Konvence	Při zápisu algoritmů jsou použity některé konvence (doporučení pro zvýšení srozumitelnosti programu). Doporučuje se, aby identifikátor typu začínal velkým písmenem T (TUzel je typ pro strukturu uzlu). Identifikátor datového typu - kromě standardních - se specifikuje v sekci specifikace typu. Příklad: type TPole = array[1..10] of integer; (* jednorozměrné pole 10 prvků typu integer *) Doporučuje se, aby identifikátor ukazatele obsahoval podřetězec "uk" nebo "ptr" (příkl. UkUzel, ListPtr). Identifikátory složené ze (zkratky) několika slov zvýrazňují začátek slova velkým písmenem (Příklad : PoleZnaku). Identifikátor proměnné, která je jediným prvkem svého typu, se liší absencí znaku T (příklad. var Den : TDen). Pro zvýšení srozumitelnosti textu programu se doporučuje odsazování (<i>indentation</i>) zanořených datových a příkazových struktur odsazením dalšího rádku o dva znaky, používání příkazových závorek begin a end tak, aby pravá závorka end byla pod odpovídající levou závorkou begin, nebo pod vyhrazeným identifikátorem uvozujícím datovou nebo příkazovou strukturu. Příklady budou uvedeny u jednotlivých typů příkazů. Jazyky, které využívají odsazování používají většinou tiskový font "courier", který zachovává pozici jednotlivých znaků. Mezery mezi základními prvky textu programu nemají význam a používají se jen k zpřehlednění zápisu.
Vybrané identifikátory	Algoritický jazyk používá některých vybraných identifikátorů (slov) pro účely tvorby programových, datových a příkazových konstrukcí. Mezi nejpoužívanější vybraná slova námi používaného pascalovského jazyka budou patřit: begin, end levá a pravá příkazová závorka type, var zahájení sekce specifikace typů a deklarace proměnných integer, real, boolean, char ident. standardních jednoduchých typů array, record, (file, set) identifikátory standardních strukturovaných typů for, to, downto, while, repeat, until vybraná slova pro tvorbu cyklů

<code>if, then, else, case, of</code>	vybraná slova pro konstrukci alternativních příkazů
<code>procedure, function</code>	vybraná slova označující deklaraci procedury resp. funkce
<code>div, mod, not, and, or</code>	vybraná slova pro aritmetické a logické operátory
<code>false, true</code>	vybraná slova (literály) booleovských hodnot

Algoritrický jazyk používá identifikátory vyhrazené pro některé standardní procedury funkce, jako "succ, pred, ord, chr, even, odd" a další, které budou uvedeny ad hoc při svém použití.

2.2 Datové typy a struktury

2.2 Datové typy a struktury

Datové typy se dělí na **jednoduché, strukturované** a **typ ukazatel**. Nad všemi datovými typy je definována operace přiřazení, jejímž symbolem je operátor "`:=`". (Výjimku v Pascalu tvoří typ soubor - **file**, se kterým se v rámci výkladu algoritmů setkáme jen okrajově, a nad nímž operace přiřazení není definovaná, protože nemá hodnotu). Podmínkou přiřazení je **kompatibilita vzhledem k přiřazení** pro proměnnou na levé straně a výraz na straně pravé. Datový typ se specifikuje v pseudopříkazové sekci "type". Proměnné daného typu se deklaruji (vytvářejí) v pseudopříkazové části "var". Po deklaraci nemají proměnné hodnotu (jejich hodnota není definovaná). Proměnná může získat hodnotu jen na levé straně přiřazovacího příkazu nebo jako výstupní parametr procedury.

Pozn. Některé jazyky připouštějí inicializační zápis deklarace, přiřazující počáteční hodnotu proměnné při její tvorbě.

Příklad:

Proměnné A a B jsou libovolného téhož typu (kromě file). Pokud má proměnná A definovanou hodnotu (v případě strukturovaného typu musí mít definovány hodnotu všechny složky strukturované struktury), pak má přiřazovací příkaz tvar:

`B := A`

Pozn. Některé algoritrické jazyky používají místo operátoru "`:=`" operátor "`←`". Použití operátoru "`:=`" je malá, ale pro programátory zvyklé na jazyk C nepřijemná odchylka. Také je vhodné si uvědomit rozdíl mezi operátorem jazyka C pro přiřazení "`=`" a relací ekvivalence "`==`". Omyly v jejich interpretaci jsou častými zdroji chyb.

Pro usnadnění zápisu bude algoritrický jazyk používat operace výměny hodnot (*swap*) ve tvaru:

`B := A`

která nahrazuje trojici příkazů:

`P := A; A := B; B := P ;` (* kde P je pomocná proměnná téhož typu jako A a B *)
nebo trojici příkazů:

`A := A - B; B := B + A; A := B - A;` (* pro proměnné typu integer *)

Nad všemi jednoduchými typy a nad typem ukazatel je definovaná operace

ekvivalence, jejíž výsledkem je hodnota typu boolean, (s hodnotou true nebo false). (operátorem je "=", v jazyce C "==").

Pro jednoduché datové typy vystačíme ve většině příkladů s ordinálními typy **integer**, **výčtovým typem** a **typy char** a **boolean**.

DEF

Definice:

Ordinální typ je jednoduchý typ, pro jehož každou hodnotu (s výjimkou největší hodnoty typu) existuje právě jedna následující a (s výjimkou nejmenší hodnoty typu) právě jedna předcházející hodnota.

Pozn. Typy real, ukazatel a string nejsou ordinální.

Typ real a jeho operace nebudou pro výklad látky většinou zapotřebí (výjimkou může být v některých příkladech např. výpočet průměrné hodnoty nebo rozptylu, které jsou neceločíselným výsledkem operace dělení). Příklad:

x+y

```
type (* úsek pseudopříkazu pro specifikaci typů *)
TDen= (pondeli,utery,streda,ctvrtek,patek,sobota,nedele);
(* Specifikace typu pro dny v týdnu *)
TStav = (svobodny, zenaty, rozvedeny, vdovec);

var (* úsek pseudopříkazů pro deklaraci proměnných *)
i,j,k:integer; (* tři proměnné typu integer*)
ch1,ch2:char; (* dvě proměnné typu znak *)
den1,den2:TDen; (* dvě proměnné typu TDen *)
Stav:TStav; (* proměnná typu TStav *)
```

Jednoduché typy mohou být operandy v operacích, které mají tvar **výrazu**.

Nad ordinálními typy budeme někdy používat vymezení souvislé podmnožiny dané množiny, zapisované jeho okrajovými hodnotami oddělenými dvěma tečkami.

x+y

Příklad:

0 .. 9 je označení podmnožiny jednočíslicových hodnot typu integer. Ve většině příkladů algoritmů předmětu IAL bychom vystačili s celočíselným typem **0 .. maxint** - což v některých jazyčích odpovídá typu pro "celé číslo bez znaménka".

DEF

Definice:

Výraz je předpis na získání hodnoty daného typu a je současně nositelem této hodnoty.

x+y

Pozn. **Funkce** se formálně liší od procedury tím, že má hodnotu a je tudíž **výrazem**, zatím co procedura má postavení příkazu. Další rozdíly viz odstavec "procedura".

Příklad: **5+7** je předpis na získání hodnoty, kterou získáme součtem jeho dvou operandů. Zápis "**5+7**" je však současně nositelem výsledné hodnoty, kterou je v tomto případě "**12**". Výrazy někdy označujeme písmenem **E** podle anglického pojmu **expression**.

Typ integer

Typ integer je datový typ, který nabývá celočíselných hodnot v intervalu (-maxint..maxint), kde hodnota **maxint** je symbolickou konstantou vyjadřující největší možnou celočíselnou hodnotu na daném počítači. Nad výrazy s prvky typu integer používáme operace:

- sečítání: + příklad $a+5 \quad i+j+67$
- odečítání: - příklad $9-b \quad c+35-j$

(Pozn. komutativní zákon platí jen v rozsahu -maxint..maxint. Je třeba mít v patrnosti, že zatímco pro kladná a,b,c může být výraz $a+c+b$ korektní, výraz $a+b-c$ může způsobit překročení hodnoty maxint při prvním součtu.)

- násobení: "*" Příklad: $(2*k) \quad (i*j*k)$
- celočíselné dělení: "**div**" Příklad: $(i \text{ div } 2), (5 \text{ div } 2)$
(* výsledek je 2 *)
- operace modulo (zbytek po celočíselném dělení): "**mod**"

(Pozn. Výsledek operace modulo je vždy v rozsahu 0..n-1, kde "n" je druhý operand operace.)

Příklad: $(k \text{ mod } 2), (11 \text{ mod } 3)$ (* výsledek je 2 *)

Multiplikativní operace (*, div, mod) mají při zápisu aritmetického výrazu vyšší prioritu (precedencí) než **operace aditivní** (+,-).

Výčтовý typ **Výčтовý typ** (*enumeration type*) se specifikuje výčtem identifikátorů, které představují hodnoty tohoto typu, uzavřeným do kulatých (výrazových) závorek. Pořadí hodnot ve výčtu stanovuje ordinální hodnotu (odpovídající pořadovému číslu počínaje nulou). Nad výčtovým typem budeme používat operace (funkce) **succ**, **pred** a **ord**. Zápis funkce succ(p) má hodnotu následujícího prvku p. Pro poslední prvek výčtu není definovaná. Zápis funkce pred(p) má hodnotu předcházejícího prvku p. Pro první prvek výčtu není definovaná. Funkce ord(p) je převod (konverze, *conversion*) výčtového typu na typ integer a má hodnotu ordinálního čísla (pořadí) prvku ve výčtu, počínaje nulou. Operace succ a pred mají smysl jen u výčtových typů s významovou ordinalitou. Jistě neočekáváme, že bychom nad výčtovým typem :

```
TStav=(svobodny,zenaty,rozvedeny,vdovec)
```

použili vztah succ(Stav), abychom z hodnoty proměnné Stav "rozvedeny" získali hodnotu "vdovec". Naopak u typu:

```
TMesic=(leden,unor,brezen,duben,kveten,cerven,cervenec,  
srpen,zari,rijen,listopad,prosinec)
```

má zápis succ(Mesic) smysluplné využití a proměnná Mesic se dá použít jako řídící proměnná počítaného cyklu procházející hodnoty měsíců v roce.

Typ znak **Typ znak - "char"** je ordinální typ.
Deklarace typu má tvar:

```
var  
    ch1, ch2:char;
```

V algoritmickém jazyce budeme předpokládat jeho "**kardinalitu**" (počet různých hodnot) danou 8 bitovým zobrazením kódu ASCII. Na rozdíl od výčtového typu je nad typem znak definovaná funkce pro převod celočíselného typu na typ znak - `chr(i)`, kde pro proměnnou typu znak `ch` platí: `ch = chr(ord(ch))`.

Pozn. Někteří nevyspělí programátoři používají pro vyjádření určitých znaků znalostí jejich ordinálního čísla (kódu). Použití podobných "magických" čísel a konstant nepatří k dobrému programovacímu stylu.

Hodnoty typu char zapisujeme znakem zapsaným mezi apostrofy. (Některé algoritmické jazyky připouštějí také použití uvozovek). Příklad:

'A' je zápis hodnoty znaku (písmene) A
'1' je zápis hodnoty znaku (číslice) 1

Rozumný programátor dává pozor na možné záměny znaků
'O' (velké písmeno O) a '0' (číslice nula)
'l' (malé písmeno l) a '1' (číslice jedna)

Pozn. V některých drobných zobrazeních na papíře i obrazovce/LCD je možná záměna u písmene 'O' a 'D' a u číslic '3' a '8'. Na tyto záměny je třeba dát pozor při tvorbě hesel (password) a různých licenčních kódů a při jejich generování se vyhýbat nevhodným a zaměnitelným znakům.

Příklad

Pro převod celočíselné hodnoty proměnné `i` z rozsahu 0 .. 9 na znak (číslici) lze zapsat výraz:

`chr(ord('0') + i)`

Typ boolean Typ boolean je ordinální typ, který nabývá dvou hodnot, jejichž zápis je `false` a `true`. Na základě ordinality typu platí, že `true=succ(false)`.

Deklarace proměnných typu boolean má tvar:

```
var
    B1, B2, B3, B4: boolean;
```

Nad typem boolean jsou definovány operace:

negace	"not"
logický součin (konjunkce)	"and"
logický součet (disjunkce)	"or"

Pozn. operátor `not` označujeme jako "**monoadický**", protože na rozdíl od "**dyadicických**" operátorů, které pracují se dvěma operandy, negace pracuje jen s jedním operandem.

Operace mají sestupnou prioritu v tomto pořadí: negace, konjunkce, disjunkce.

Příklad výrazu

`(B1 or B2) and not(B3 or B4)`

Hodnoty boolean nabývají také výrazy nad některými jinými typy. Mezi takové výrazy patří relace nad skalárními typy a typem řetězec (*string*).

Některé překladače dovolují zkratové (zkrácené) vyhodnocování booleovských výrazů.

Tento styl, který porušuje "matematickou čistotu" výrazu tím, že připouští nedefinovanost těch částí výrazu, k jejichž vyhodnocování při "zkratu" nedojde, jsou z praktického pohledu programátora výhodné a vedou v některých případech k jednoduššímu zápisu a kratší době vyhodnocování, což může mít význam zejména u cyklů

Výraz, který má tvar vícenásobné konjunkce, lze ukončit, jakmile se zleva narazí na první činitel s hodnotou "false"

B1 and B2 and B3 and and BN

V případě, že B1 je "false", má celý výraz hodnotu "false" apod.

Výraz, který má tvar vícenásobné disjunkce, lze ukončit, jakmile se zleva narazí na první činitel s hodnotou "true"

B1 or B2 or B3 or or BN

V případě, že B1 je "true", má celý výraz hodnotu "true" apod.

Pokud bude v algoritmech IAL použit zkratově vyhodnocovaný booleovský výraz, bude to vždy uvedeno komentářem.

Skalární typ

DEF

Definice:

Skalární typ je jednoduchý typ, pro jehož každé dva prvky lze stanovit, zda jsou si rovny nebo zda je jeden z nich menší nebo větší než druhý.

Pro operace relace budeme používat tyto dyadické operátory

ekvivalence (rovno) =

nerovno \neq nebo v kódu programu častěji \leftrightarrow

menší než <

menší roven \leq nebo v kódu programu častěji \leq

větší než >

větší roven \geq nebo v kódu programu častěji \geq

Pozn. Pro zápis $\text{not}(B1=B2)$ je vhodnější zápis $(B1 \leftrightarrow B2)$. Pro zápis $(B = \text{true})$ postačí zápis B a pro zápis $(B=\text{false})$ je vhodnější zápis $(\text{not } B)$.

Aritmetické operace mají vyšší prioritu než relační operace a ty mají vyšší prioritu než booleovské a proto je třeba při tvorbě složitějších výrazů správně používat závorek.

Příklad:

$((a+b) > (c-d)) \text{ or } ((d-f) \leq (g+h)) \text{ and } (i = j)$

je v konečné fázi operace disjunkce a je to jiný výraz, než

$(((a+b) > (c-d)) \text{ or } ((d-f) \leq (g+h))) \text{ and } (i = j),$

který je v konečné fázi operace konjunkce.

Strukturova Strukturované datové typy.

né datové typy

Mezi základní strukturované typy Pascalu patří **pole**, **řetězec**, **záznam**, množina a soubor. Algoritmický jazyk pro předmět IAL vystačí s prvními třemi strukturovanými typy.

Strukturovaný datový typ sestává z komponent jiného (dříve definovaného) typu, kterému říkáme **kompoziční typ**. Pokud jsou všechny komponenty strukturovaného typu stejného kompozičního typu, říkáme, že strukturovaný typ je **homogenní**. Komponentám homogenního typu se někdy říká **položky** (*item*), zatím co komponentám heterogenního typu se říká **složky** (*component*). Strukturovaný typ má **strukturovanou hodnotu**. Tato hodnota je definovaná tehdy, když jsou definované hodnoty všech jejích komponent.

Pozn. **Soubor** (*file*) je jednou z nejvýznamnějších datových struktur, protože to je jediná datová struktura, kterou program komunikuje se svým okolím. Z toho pohledu je mezi soubory nejvýznamnější tzv. textový soubor, který je základem většiny vstup/výstupních operací. Protože se algoritmy v předmětu IAL nebudu zabývat vstupními/výstupními operacemi, budou při výkladu algoritmů hrát soubory jen okrajovou roli. Také **množina** (set) je významnou datovou strukturou Pascalu. Některé její vlastnosti zavedeme do algoritmického jazyka jen účelově, při výkladu vytváření datových struktur typu množina.

Pole

Pole (array) je homogenní ortogonální (pravoúhlý) datový typ. Jednorozměrnému poli říkáme **vektor**, dvojrozměrnému poli říkáme **matice**. Lze vytvářet pole s neomezenou vícerozměrnou hierarchickou strukturou. Více než dvojrozměrná pole však nebudeme v IAL používat. Prvek vektoru je zpřístupňován jedním, prvek matice dvojicí indexů. Indexem může být libovolný ordinální typ. Pro svou homogenitu říkáme prvkům pole také **položky**. Typ pole je specifikován **rozsahem svých dimenzií** (rozměrů) a komponentním typem. Pole, jehož jméno a velikost dimenzi je pevně dána deklarací a nemohou se měnit v průběhu programu, se nazývá **statické**. Pokus o přístup k prvku pole, jehož index je mimo specifikovaný rozsah, způsobí chybu. Příklad specifikace:

```
type
  TVektor = array[1..100] of integer; (* vektor sta celých čísel *)
  TMaticeV = array[1..10] of TVektor; (* vektor deseti vektorů o 100
celých číslech, tedy matice 10*100 *)
  TMaticeP = array[1..10,1..100] (* matice o deseti řádcích a 100
sloupcích celých čísel *)
```

Na první pohled vypadá matice TMaticeV a TMaticeP stejně. Má stejný počet prvků, stejně rozměry. Také přístup k prvkům obou matic je shodný. Pro obě matice je zápis:

TMaticeV[5,8] resp. TMaticeP[5,8]

zápisem hodnoty prvku v pátém řádku, v osmém sloupci. Protože však tvar specifikací typů je odlišný, nejsou obě matice kompatibilní a po deklaraci proměnných daného typu

```
var
  MatV:TMaticeV;
  MatP:TmaticeP;
  Vekt:TVektor;

nelze zapsat přiřazovací příkaz MatV:=MatP;
```

zato lze zapsat `MatV[5] :=Vekt`, protože pátým prvkem `Mat1` je podle specifikace vektor.

Na rozdíl od jazyků blízkých strojů, které při specifikaci používají většinou jen horní hodnotu rozsahu dimenze (s implicitní spodní hranicí rovnou nule), je v jazycích vyššího typu používána i dolní hranice, nejčastěji s hodnotou 1. Uvědomme si, že dva vektory:

```
TVekt0 = array[0..9] of integer;
TVekt1 = array[1..10] of integer;
```

mají stejný počet prvků.

Algoritmu, v němž postupně projdeme (a postupně stejným způsobem zpracujeme) všechny prvky homogenní datové struktury, říkáme průchod. Pro práci s maticí jsou nejtypičtější průchody "**po řádcích**" a "**po sloupcích**". Při průchodu po řádcích se pro každý řádek postupně mění hodnoty sloupcového (druhého, pravého) indexu. Až se vyčerpají všechny jeho hodnoty, zvýší se hodnota řádkového (prvního, levého) indexu. Obecně říkáme, že **při průchodu vícerozměrným polem "po řádcích", se rychlosť změny indexu snižuje zprava doleva**. Pro průchod "po sloupcích" platí symetricky komplementární definice. Nestanovíme-li jinak, budeme považovat způsob organizace matic za odpovídající upořádání "po řádcích" a první dimenze (a první index) je index řádkový.

Pozn. V jazycích pascalovského typu je pole principiálně jednorozměrné pole (vektor). N-rozměrné pole se jeví jako jednorozměrné pole (N-1)-rozměrných polí. Matice je tedy vektor vektorů a je otázka konvence, zda je to vektor řádků nebo vektor sloupců. Evropská a pascalovská konvence považuje za standardní v upořádání "po řádcích".)

DEF Řetězec

Řetězec (string) je strukturovaný homogenní datový typ. Položkami řetězce je typ znak a řetězec má vlastnosti podobné jednorozměrnému poli znaků. Nad typem řetězec je definován bohatý repertoár operací. Popis některých operací tohoto repertoáru bude uveden až v případě jejich použití. Mezi nejvýznamnější operace patří relace uspořádání, která dovoluje, aby nad typem string byla aplikována nejen relace ekvivalence, ale také relace menší, větší, menší-roven, větší-roven. Nejtypičtější operací nad řetězcem je konatenace (spojení) dvou řetězců, která má podobu "součtu" s operátorem "+". Z hlediska praktického využití jsou nejvýznamnější operace konverze typů integer a real na řetězec, které jsou náročnou součástí všech vstup/výstupních operací. Tyto operace nebudou součástí algoritmů IAL.

Řetězec je strukturovaný datový typ, nad nímž je definována operace typu "**konstruktor**". **Konstruktor je operace, která dovoluje ustavit strukturovanou hodnotu strukturovaného typu výčtem všech jeho komponent**. Zápis konstruktoru je nositelem strukturované hodnoty. U řetězce se v konstruktoru uvede výčtem posloupnost všech jeho znaků ohraničená apostrofy. Příklad práce s řetězci:

```
var
  S1,S2,S3:string; (* deklarace tří proměnných typu řetězec *)
  ch1,ch2:char;      (* deklarace dvou proměnných typu znak *)
```

```
begin
```

```

.....
S1:='Hello_';
S2:='folks!';
S3:=S1 + S2;    (* řetězec S3 je spojením - konkatenací - řetězců S1 a S2 *)
ch1:= S3[10];   (* Znak ch1 získá hodnotu 10. znaku řetězce S3 *)
S2[5]:='A';     (* Pátý znak řetězce S2 získá hodnotu znaku 'A' *)

```

Pozn. Implementace řetězce v typických implementacích Pascalu, jakou je např. Turbo Pascal, se liší od typické implementace v jazycích odvozených od jazyka C. Zatím co v pascalovském pojetí je pro řetězec "string" vyhrazeno pole o maximální délce 256 znaků a v bytu na pozici 0 je uvedena hodnota aktuálního počtu znaků (standardní řetězec tedy může mít maximálně 255 znaků), je řetězec v jazyku C implementován neomezenou posloupností znaků zakončenou nulitním znakem (znak s ordinální hodnotou 0).

Záznam

Záznam (record) je obecně strukturovaný heterogenní datový typ. Jeho komponenty mohou být libovolného, dříve definovaného typu a často jim říkáme **složky** (*component*). Záznam je statická struktura. Jména a počet jeho komponent jsou dány při specifikaci typu a nemohou se měnit v průběhu programu. Protože komponentou záznamu může být i jiný strukturovaný typ, lze vytvářet záznamy s obecně neomezenou hierarchickou strukturou. Zatímco položka pole se zpřístupňuje indexem ordinálního typu, který je "spočitatelný" (parametrizovaný), složka záznamu se zpřístupňuje zvláštním identifikátorem - jménem složky, který nelze parametrizovat. Příklad specifikace typu osoba:

x+y

```

type
  TDatum=record    (* Záznam typu data v roce *)
    rok, den, mesic:integer;
  end;

  TOsoba=record    (* Záznam osoby se jménem a datem narození *)
    jmeno:string;
    datnar: TDatum
  end;

var
  Os1, Os2:TOsoba;
  Dat:TDatum;
  roknar:integer;
  .....
begin
  ...
    Dat.rok:=1990;  (* ustavení roku data narození *)
    Dat.mes:=2;      (* ustavení měsíce data narození *)
    Dat.den:=29;     (* ustavení dne data narození *)

  Os1.jmeno:='Novak'; (* ustavení jména osoby *)
  Os1.datnar:=Dat;    (* ustavení data narození osoby *)
  Os1.datnar.rok:=1950; (* oprava data narození osoby *)

  roknar:=Os2.datnar.rok; (* uložení roku narození osoby Os2 do roknar*)
  ...

```

DEF **Typ ukazatel** Typ ukazatel je spjat s dynamickými typy. Prvek dynamického typu vzniká v průběhu výpočtu jako výsledek operace (procedury) "new (Uk)", která v části paměti určené pro dynamické struktury vyhradí místo pro prvek (podle jeho velikosti) a ukazateli Uk přířadí hodnotu, která umožní přístup k tomuto místu. Dynamický prvek nemá jméno a jeho hodnota se tedy nezpřístupňuje identifikátorem, ale prostřednictvím ukazatele. Pro specifikaci typu ukazatel i a pro dereferenci (zpřístupnění) dynamického prvku budeme používat zápisu s pascalovským symbolem "^" (tzv. "šipka-notace"). Typ ukazatel má zvláštní hodnotu označovanou jako "nil", která má sémantiku "**ukazatel, který neukazuje na žádný prvek**" nebo "**prázdný ukazatel**". Hodnota "nil" je kompatibilní se všemi prvky typu ukazatel.

Pozn. Vytvořený (alokovaný) prvek dynamického typu zaniká (dealokuje se) operací dispose, která je komplementární k operaci new. Po operaci dispose je hodnota ukazatele rušeného prvku nedefinovaná.

Příkl.

```
x+y
type
    TUkUz=^TUzel;          (* TUzel je záznam uzlu binárního stromu *)
    TUzel=record           (* Specifikace typu Uzel *)
        DataUzlu:string;   (* data uzlu *)
        LUK, PUK:TUkUz     (* levý a pravý ukazatel na synovské uzly *)
    end;
...
var
    UkKoren:TUkUz;         (* deklarace statické proměnné typu ukazatel *)
.....
begin
...
new(UkKoren);             (* vytvoření dyn. prvku, zpřístupňovaného statickým ukazatelem UkKoren *)
UkKoren^.DataUzlu:='KAREL';
(* reference dynamického prvku s přiřazením hodnoty typu string složce DataUzlu *)
...
dispose(UkKoren); (* zrušení prvku zpřístupňovaného ukazatelem UkKoren *)
```

Pozn: Za povšimnutí stojí skutečnost, že datový typ TUzel byl ve specifikaci ukazatele TUkUz použit dříve, než byl sám specifikován. Je to jediná výjimka v povinnosti dodržet právo pořadí: "napřed specifikovat a potom použít".

2.3 Příkazy 2.3 Příkazy lze rozdělit na jednoduché, strukturované a příkazy procedury.

Jednoduché příkazy. Nejdůležitějším jednoduchým příkazem je **přiřazovací příkaz**. Přiřazovací příkaz přiřazuje proměnné na levé straně hodnotu na pravé straně přiřazovacího operátoru ":=". Proměnná i výraz musí být vzájemně **kompatibilní**.

vzhledem k přiřazení. Pro účely algoritmického jazyka postačí zjednodušená definice.

DEF

Definice (zjednodušená) :

Dva prvky jsou kompatibilní vzhledem k přiřazení, když jsou jejich typy identické (ekvivalentní).

Přiřazovací příkaz lze použít pro všechny jednoduché i strukturované typy s výjimkou typu file a strukturovaných typů, které jako svou komponentu obsahují typ file.

Prázdný příkaz má postavení příkazu a neodpovídá mu žádná činnost ve fázi výpočtu. Na základě definice středníku jako oddělovače dvou příkazu, lze zápis:

... ; ; ... interpretovat, jako prázdný příkaz oddelený dvěma středníky a zápis:
... ; end jako prázdný příkaz mezi středníkem a pravou příkazovou závorkou end.

Pozn. **Příkaz skoku** je pascalovský příkaz, který v algoritmickém jazyce nebudeme používat.

Strukturované příkazy Mezi strukturované příkazy algoritmického jazyka jsou zařazeny **složený příkaz, alternativní příkaz a příkaz cyklu**.

Složený příkaz již byl definován jako sekvence příkazů oddelených středníky, uzavřená mezi příkazové závorky begin a end.

Alternativní příkazy jsou: **podmíněný příkaz, alternativní příkaz a vícečetný alternativní příkaz case.**

Podmíněný příkaz Podmíněný příkaz je příkaz, který se provede, když má jeho booleovský řídící výraz hodnotu true.

Příkl.:

```
if B  
then S
```

Příkaz S se provede, když má booleovský výraz B hodnotu true; v jiném případě má příkaz význam prázdného příkazu. Příkazem S může být libovolný (jeden) příkaz (např. složený příkaz).

Alternativní příkaz Alternativní příkaz provede jednu ze dvou alternativ v závislosti na hodnotě řídícího booleovského výrazu B

Příkl.:

```
if B  
then S1  
else S2
```

V případě, že B má hodnotu true, provede se příkaz S1, jinak se provede příkaz S2.

Pozn. Beprostředně před "else" nesmí být uváděn středník. Středník odděluje dva příkazy a mezi "then" a "else" smí být jen jeden příkaz (byť strukturovaný). Konvence zápisu ve tvaru:

```

if B
then begin
  S (* příkaz *)
end else begin
  S (* příkaz *)
end (* if B *)

```

tomu zabraňuje.

- Příkaz case** Vícečetný alternativní příkaz **case** je řízen **výrazem** ordinálního typu. Jednotlivé alternativy (alternativní příkazy) jsou označeny výběrovým ordinárním výrazem s odpovídající hodnotou. Při provedení příkazu se vybere příkaz, označený hodnotou, která odpovídá hodnotě řídicího výrazu, jinak se provede alternativa poznačená řídicím slovem **else**. Není-li **else** uvedeno interpretuje se **case** jako příkaz prázdný. Příkaz **case** budeme používat jen okrajově.

Příklad:

```

case i of
  1: j:=j+i;
  2: j:=j+i*i;
  3: j:=j+i*i*i;
  4: j:=j+i*i*i*i;
else j:= i;

case I of
  0,2,4,6,8: EditText:= 'Even digit';
  1,3,5,7,9: EditText:= 'Odd digit';
  10..100: EditText:= 'Between 10 and 100'
else
  EditText:= 'Negative or greater than 100'
end;

```

Pozn. Častou chybou je pokus o použití řídicí proměnné typu "string". String není ordinární typ.

- Příkazy cyklu** dělíme na **počítané** (explicitní), **nepočítané** (implicitní) a **nekonečné**.

- Počítaný cyklus** se použije v případě, kdy můžeme explicitně stanovit počet jeho opakování. Řídicí proměnná (počítadlo) je ordinárního typu a pascalovské pojetí předpokládá jeho inkrementaci (zvýšení na hodnotu následníka) nebo dekrementaci snížení na hodnotu předchůdce). Vlastní průchod cyklem (jedno provedení těla cyklu, smyčka) má podobu jednoho příkazu uvedeného za vybraným slovem **do**. Po skončení příkazu cyklu provedením jeho posledního průchodu se předpokládá nedefinovaná hodnota řídicí proměnné.

Příklad:

```

for i:=1 to 100 do begin
  Sum:=Sum + Pole[i]
end; (* for *)
...

```

```

for j:= 50 down to 30 do
begin
  if Pole[i] >= 0
  then Sum:=Sum + Pole[i]
  else Sum:= Sum - Pole[i]
end;

```

Pozn. Příkazové závorky v předcházejících příkladech nejsou nutné. Anticipační styl zápisu programu je však používá za každé "then", "else" a "do". Předjímá skutečnost, že se při modifikaci programu vnitřní příkaz rozšíří a zvyšuje čitelnost začátku a konce těla cyklu. "Sebeobranný" styl zápisu přidává ke každému "end" stručný komentář označující kterému "begin" příslušný "end" patří. Často se používají dva zápisové a odrážkové styly

(indentation). V prvním cyklu je použit styl, v němž se "begin" píše bezprostředně za "do", "then", "else" a "end" se zarovnává z úvodním vybraným slovem ("for", "if" apod.) a komentářem se upřesní u čeho je odpovídající "begin". Ve druhém cyklu se "end" píše od stejné pozice jako jeho "begin" a vnitřní příkazy jsou odsazeny o dvě pozice. Existuje řada jiných stylů. Systematické používání vhodného stylu zvyšuje srozumitelnost programu a patří k "sebeobrannému" stylu programování.

Pozn. Změna (úprava) hodnoty počítadla v těle počítaného cyklu patří k nevhodným "trikům" porušujícím "sebeobranný" styl programování.

Pozn. V "blokové struktuře" výstavy programu platí, že řídicí proměnná počítaného cyklu "for" nesmí být v proceduře (funkci) globální. Pro účely našeho předmětu to znamená, že každý počítaný cyklus, který je součástí procedury, musí používat **počítadlo deklarované vnitř těto procedury**.

Nepočítaný cyklus

Nepočítané cykly se dělí podle toho, kdy se provádí test na konec cyklu. **Cyklus while** (označovaný také jako cyklus "nula nebo n"), provádí test před začátkem průchodu cyklem, na základě hodnoty booleovského výrazu B. Cyklus se provede, pokud má výraz B hodnotu "true". Má následující syntaktické schéma:

```
while B do S
```

Ve většině případů předchází cyklu typu while nastavení počátečních podmínek cyklu.

Příklad:

```

i:=1;      (* nastavení počítadla *)
F:=1;      (* nastavení sumační proměnné *)
while F < 100 do begin
(* Výpočet nejmenší hodnoty faktoriálu větší než 100 *)
  i:=i+1;
  F:=F*i;
end (* while *)
(* F = i! *)

```

Cyklus repeat - until (označovaný také jako "jedna nebo n"), provádí test na konci průchodu a jeho tělo se musí provést alespoň jednou. Cyklus se ukončí, je-li hodnota řídicí proměnné B rovna "true". V některých případech, kdy je z povahy algoritmu zřejmé, že se cyklus provede nejméně jednou, může vést použití cyklu "repeat - until" ke kratšímu a jednoduššímu zápisu. Cyklus má následující syntaktické schéma:

```

repeat
  S
until B

```

x+y

Příklad:

```
i:=1;      (* nastavení počítadla *)
F:=1;      (* nastavení sumační proměnné *)
repeat
(* Výpočet nejmenší hodnoty faktoriálu větší nebo rovné 100
*)
    i:=i+1;
    F:=F*i
until F>=100
(* F=i! *)
```

Pozn. V případě, že booleovský výraz B je složený, pak je srozumitelnější, aby v případě cyklu "while" měl podobu konjunkce (logického součinu) a v případě cyklu "repeat-until" podobu disjunkce (logického součtu). Na příklad: cyklus "while" se provede, když platí "B1 a současně B2", zatím co cyklus "repeat" se ukončí, když platí "B1 nebo B2". Pro algoritmy vyhledávání pak platí schéma "while" v podobě: "Pokračuj ve vyhledávání, když platí :<ještě jsem nenašel hledaný prvek> a současně <ještě jsem neprošel všemi prohledávanými prvky>" a schéma "repeat-until" v podobě: "Končím prohledávání, když platí :<našel jsem hledaný prvek> nebo <prošel jsem všemi prohledávanými prvky>".

Pozn. **Nekonečný cyklus** nemá v uvedeném algoritmickém jazyce speciální konstrukci. Zapsat lze např. konstrukcí: "while true do S". Algoritmy s jednou řídicí linií (sériové algoritmy), které se budou probírat, ho nepotřebují.

Příkaz with **Příkaz with** se tváří jako cyklus, má ale spíše vlastnost složeného příkazu ohraňujícího sekvenci příkazů, na něž se vztahuje zjednodušení přístupu ke složkám jednoho záznamu. Příkaz má podobu:

```
with I do
begin S
end
```

kde I je identifikátor typu záznam. Pak ve všech příkazech v rámci těla tohoto "cyklu" se nemusí složky záznamu I zapisovat s referenčním předponou "I".

x+y

Příklad: Místo zápisu

```
begin
    Osoba.jmeno:='NOVAK';
    Osoba.RokNar:=1980;
    Osoba.vyska:=180;
    Osoba.vaha:=78
end;
```

lze zapsat jednodušší zápis:

```
with Osoba do
begin
    jmeno:='NOVAK';
    RokNar:=1980;
    vyska:=180;
    vaha:=78
end;
```

a vyhnout se tak opakovanému zápisu stejného identifikátoru téhož záznamu.

- Procedury a funkce** **Procedura (funkce)** je nástroj zvyšování abstrakce příkazů (výrazů). Má formu "uzavřeného podprogramu"; to znamená, že na místě každého použití procedury (funkce) se překladem generuje skok do podprogramu, který je obecně umístěn na jiném místě paměti, než hlavní program. Po provedení podprogramu se řízení vrací za skok do podprogramu.

Pro použití procedury se v sekci deklarace uvede "předpis" na provedení procedury, který sestává z "hlavičky" a z "těla" procedury (funkce). V hlavním programu se procedura "vyvolá" příkazem procedury. Funkce se vyvolá zápisem funkce na místě výrazu.

- Hlavička procedury** Hlavička procedury sestává z vybraného slova "**procedure**" ("**function**"), identifikátoru a seznamu formálních parametrů. Formální parametry jsou odděleny (po skupinách parametrů téhož typu) středníkem. Formální parametry předávané odkazem jsou uvedeny vybraným slovem "**var**". Ostatní parametry jsou předávané hodnotou.

Příklad:

`procedure NejdNeklesPosl (var Pole:TPole; pocet:integer; var zacatek, delka:integer);`

Procedura vyhledá délku nejdělsší neklesající posloupnosti (NNP) hodnot po sobě jdoucích prvků typu integer v jednorozměrném poli Pole. Pole má "pocet" prvků ($pocet > 0$). Výsledná nejdělsší neklesající posloupnost začíná na indexu "zacatek" a má "delka" prvků. V případě více shodných NNP je výsledkem první NNP v pořadí od začátku pole.

Parametr "Pole" je vstupní parametr a předává se odkazem. Algoritmus pracuje s "originální" datovou strukturou, která je umístěna "vně" těla procedury - je ve vztahu k proceduře "globální". Parametr "pocet" udává délku pole, je vstupním parametrem a je předáván hodnotou. Existuje uvnitř těla procedury a hodnotu odpovídajícího formálního parametru získává automatickým přiřazením, před zahájením provádění těla procedury. Parametry "zacatek" a "delka" jsou výstupní parametry a jako takové jsou povinně předávány odkazem.

Pozn. Vstupní parametry se obvykle předávají hodnotou. To znamená, že se v těle procedury automaticky vytvoří "duplicát" formálního parametru, kterému se přiřadí hodnota skutečného parametru. U rozsáhlých strukturovaných typů to ale znamená spotřebu paměti a času na vytvoření duplikátu. Proto se v takových případech někdy volí předávání odkazem, které zajišťuje přístup k originální - vnější struktuře. Výstupní parametry se povinně předávají odkazem. Parametr typu "file" se nemůže předávat hodnotou.

- Tělo procedury**

Tělo procedury má tvar bloku, tzn., že sestává z nepovinné úvodní sekce dovolující specifikaci konstant, typů a deklaraci proměnných a procedur (funkcí) následovanou složeným příkazem.

Příklad (tělo procedury NejdNeklesPosl):

```
var (* deklarace lokálních proměnných *)
    PomDel, PomZac: integer; (* pomocná délka a začátek *)
    i, h: integer;           (* počítadlo a pracovní hodnota*)
```

```

begin (* vlastní tělo procedury *)
(* inicializace proměnných *)
delka:=1;
zac:=1;
PomDel:=1
PomZac:=1;
i:=1;
h:= Pole[i]; (* nastavení pracovní hodnoty*)
while i<pocet do begin
    i:=i+1;
    if h<=Pole[i]
    then begin (* posloupnost pokračuje *)
        PomDel:=PomDel+1;

    end else begin (* neklesající posl. skončila *)
        if delka<PomDel
        then begin (* korekce kandidáta na výsledek *)
            delka:=PomDel;
            zacatek:=PomZac;
        end; (* if delka *)
        PomZac:=i; (* začátek další NP *)
        PomDel:=1; (* inicializace délky další NP *)
    end; (* if h<=.. *)
    h:=Pole[i] (* nastavení nové pracovní hodnoty*)
end; (* while *)
if delka<PomDel (* test poslední NP, zakončené polem *)
then begin
    delka:=PomDel;
    zacatek:=PomZac
end
end; (* konec těla deklarace procedury *)

```

Volání procedury

Volání procedury má postavení příkazu a sestává ze jména procedury a seznamu skutečných parametrů. Skutečné parametry musí souhlasit s formálními co do počtu, pořadí i typu. Skutečné parametry odpovídající formálním parametrům předávané odkazem jsou identifikátory proměnných odpovídajícího typu. Skutečné parametry předávané hodnotou jsou výrazy odpovídajícího typu. Tato vlastnost je typická pro jazyky se **silnou kontrolou typů**.

Funkce

Funkce má vlastnosti procedury, ale má postavení výrazu s hodnotou, jejímž nositelem je vyvolání (zápis) funkce. Hlavíčka funkce je doplněna identifikátorem typu hodnoty funkce. V algoritmickém jazyce může funkce nabývat hodnot typů: "integer", "real", "char", "boolean", "výčtového typu" a typu "string". V těle funkce musí být alespoň jeden příkaz, v němž se jménu funkce přiřazuje hodnota. Hodnota, která je přiřazena při provedení těla funkce jménu funkce, je vracená výsledná hodnota funkce. Jméno funkce se smí v těle funkce použít jen na místě proměnné (na levé straně přiřazovacího výrazu nebo jako výstupní parametr procedury). Na jiném místě se interpretuje jako rekurzivní volání této funkce.

Volba, zda daný algoritmus nebo část programu má mít formu procedury nebo funkce je

dána zkušeností a stylem programátora. Funkci volíme nejčastěji tehdy, když výsledkem úseku je jedna hodnota jednoduchého typu (jakým může být funkce). Velmi častým použitím je funkce typu boolean, použitá ve strukturovaném příkaze "if", "while" nebo "repeat-until". Vícenásobné volání stejné funkce se stejným argumentem si vynucuje vícenásobný výpočet se stejným výsledkem a zvyšuje časovou náročnost programu. Pokud bude hodnota funkce voláním získávána vícekrát pro stejné vstupní argumenty (parametry), je použití funkce ve výrazu nebo v takovém úseku nevhodné a je účelné ho nahradit procedurou s vhodným výstupním parametrem, nebo pomocnou proměnnou, které se mnohanásobně vyvolávaná hodnota přiřadí.

 Příklad:

```
x := sin(alfa*alfa) + 2*(y/sin(alfa*alfa))
```

lze nahradit zápisem

```
s:=sin(alfa*alfa);
x:= s+2*(y/s);
```

Rekurze

Rekurzívní volání procedury (funkce) je konstrukce, při níž procedura (funkce) je volána v těle sebe samé. Rekurze je účinným nástrojem zápisu řady algoritmů. Rekurzívní volání funkce má vždy tvar výrazu, který je součástí (nejčastěji přiřazovacího) příkazu.

Parametry

DEF **Definice:** Vyvolání procedury (funkce) má stejný účinek, jako kdyby na tomto místě bylo tělo procedury, v němž jsou všechny formální parametry předávané odkazem nahrazeny odpovídajícími skutečnými parametry a všem formálním parametrům předávaným hodnotou jsou přiřazeny hodnoty jim odpovídajících skutečných parametrů před provedením prvního příkazu těla procedury.

Parametry jsou významným prostředkem k předávání informace mezi procedurou (funkcí) a jejím okolím. Procedury "bez parametrů" jsou v dobré postaveném programu výjimkou a je účelné se jim využívat! Seznam parametrů vytváří rozhraní (*interface*) a je jako "konektor", kterým je "černá skřínka" procedury zasazena do programu. Téměř každá komunikace procedury se svým okolím mimo seznam parametrů (mimo konektor) má potenciálně podobu "rušení" (jako když elektrické zařízení vyzařuje vně nežádoucí signály, nebo je jimi naopak zvnějšku rušeno).

 Existují případy, kdy jsou rozumné důvody pro použití globálních proměnných. Je na to ale vždy nutno upozornit v komentáři procedury! Komunikace mimo parametry má často podobu "vedlejšího jevu".

Vedlejší jevy

Vedlejší jev (*side effect*) je pojem, kterým se označuje změna hodnoty globální proměnné uvnitř těla procedury. Častým případem je vstupní parametr předávaný odkazem.

 Příklad:

```
function DET (var MatA:TMatice; N:integer):integer;
(* funkce výpočtu determinantu *)
```

pak se může stát, že následující dva výrazy nebudou mít stejnou hodnotu:

$$A[1,1] + \text{DET}(A, 10) + A[1,1]$$

$$a \\ 2 * A[1,1] + \text{DET}(A, 10)$$

protože při výpočtu determinantu mohlo dojít ke změně hodnoty prvku [1,1].

Předávání parametru matice hodnotou sice zvýší paměťovou i časovou náročnost, ale zabrání vedlejšímu jevu.

```
function DET (MatA:TMatice; N:integer):integer;
(* funkce výpočtu determinantu, matice předána hodnotou *)
```

 Řada jazyků pascalového typu nestanoví pořadí, ve kterém se zpracovávají indexy indexového výrazu, ani pořadí zpracovávání parametrů při volání procedury nebo funkce. To může být příčinou neurčitelného vlivu vedlejšího jevu !

2.4

Složitost

2.4 Základní pojmy složitosti algoritmů

Pro hodnocení kvality algoritmu i pro porovnání dvou různých algoritmů jsou zapotřebí vhodná kriteria. Účelnými parametry jsou **čas** potřebný pro provedení algoritmu a **paměťový prostor**, který zaujímá program a jeho data. Čas i prostor potřebný pro algoritmus závisí na velikosti zpracovávaných dat. Proto má složitost nejčastěji podobu funkce velikosti dat, udávané počtem položek N.

Asymptotická časová složitost je nejčastějším hodnoticím kriteriem pro algoritmy. Vyjadřuje se porovnáním algoritmu s jistou funkcí pro N blížícímu se nekonečnu. Porovnání má podobu tří různých složitostí:

- O Omikron (velké O, O, *big O*) - vyjadřuje horní hranici chování
- Ω Omega - vyjadřuje dolní hranici chování
- Θ Theta - vyjadřuje třídu chování

Pozn. Skutečnost, že asymptotická složitost jednoho algoritmu je menší než jiného nemusí znamenat, že pro každé n je první algoritmus rychlejší. Složitost algoritmu vyjadřuje obecné chování třídy algoritmů, čas algoritmu vyjadřuje konkrétní dobu pro jisté N. Pro praktické aplikace se nejčastěji algoritmy charakterizují složitostí O, která vyjadřuje "nejhorší případ" (*worst case*).

Složitost O

Pozn. V následujícím textu budeme pro kvantifikátory používat následující identifikátory:

$\exists \rightarrow Exist$

$\forall \rightarrow ForAll$

Složitost vyjádřenou zápisem **O (Big O, O)** vyjadřuje **horní hranici** časového chování algoritmu.

O(g(n)) označuje množinu funkcí f(n), pro které platí:
 $\{f(n) : \text{Exist } (c > 0, n_0 > 0) \text{ takové, že ForAll } n \geq n_0 \text{ platí}$
 $[0 \leq f(n) \leq c * g(n)]\}$

kde c a n_0 jsou jisté vhodné kladné konstanty.

DEF

Pak zápis $f(n)=O(g(n))$, označuje, že funkce $f(n)$ roste maximálně tak rychle jako funkce $g(n)$. Funkce $g(n)$ je horní hranicí množiny takových funkcí, určené zápisem $O(g(n))$.

Složitost Ω Složitost Ω vyjadřuje **dolní hranici** časového chování algoritmu.

$\Omega(g(n))$ označuje množinu všech funkcí $f(n)$ pro které platí:

{ $f(n) : \text{Exist } (c>0, n_0>0) \text{ takové, že ForAll } n \geq n_0 \text{ platí}$
 $[0 \leq c^*g(n) \leq f(n)]\}$

kde c a n_0 jsou určité vhodné kladné konstanty.

DEF

Pak zápis $f(n)=(\Omega(g(n)))$ označuje, že funkce $f(n)$ roste minimálně tak rychle, jako funkce $g(n)$. Funkce $g(n)$ je dolní hranicí množiny všech funkcí, určených zápisem $(\Omega(g(n)))$.

Složitost Θ Složitost Θ vyjadřuje **třídu** časového chování algoritmu.

$\Theta(g(n))$ označuje množinu všech funkcí $f(n)$, pro něž platí :

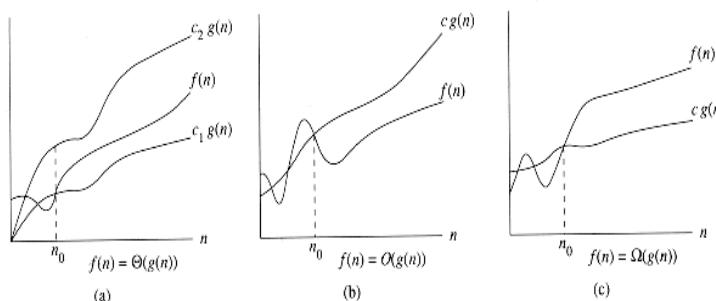
{ $f(n) : \text{Exist } (c_1>0, c_2>0, n_0>0) \text{ takové, že}$
 $\text{ForAll } (n \geq n_0) \text{ platí } [0 \leq c_1^*g(n) \leq f(n) \leq c_2^*g(n)]\}$

kde c_1, c_2 and n_0 jsou vhodné kladné konstanty.

Složitost vyjádřená zápisem Theta (Θ) označuje časové chování shodné jako daná funkce.

Pak zápis $f(n)=\Theta(g(n))$ označuje, že funkce $f(n)$ roste tak rychle jako funkce $g(n)$. Funkce $g(n)$ vyjadřuje horní a současně dolní hranici množiny funkcí, označených zápisem $\Theta(g(n))$.

Grafické znázornění složitosti O , Ω a Θ . Pro časovou složitost představuje osa y čas.


Klasifikace algoritmů

1. Theta(1) je označení algoritmů s **konstantní** časovou složitostí.
2. Theta(log(n)) je označení algoritmů s **logaritmickou** časovou složitostí. Základ logaritmu není podstatný, protože hodnoty logaritmu pro různé základy se liší pouze konstantou vzájemného převodu. Logaritmickou složitostí se vyznačují např. rychlé vyhledávací algoritmy.

3. Theta(n) je označení algoritmů s **lineární** časovou složitostí. Tuto složitost mají běžné vyhledávací algoritmy a řada algoritmů sekvenčně zpracovávajících datové struktury.
 4. Theta($n \log n$) je označení algoritmů nazývané také **linearitické**. Tuto časovou složitost mají např. rychlé řadicí algoritmy.
 5. Theta(n^2) je označení algoritmů s **kvadratickou** časovou složitostí. Takovou časovou složitostí se vyznačují algoritmy sestavené z dvojnásobného počítaného cyklu do n . Patří mezi ně řada klasických řadicích algoritmů (např. bublinové řazení).
 6. Theta(n^3) je označení algoritmů s **kubickou** časovou složitostí. Algoritmy s touto časovou složitostí jsou prakticky použitelné především pro málo rozsáhlé problémy. Kdykoli se n zdvojnásobí, čas zpracování je osminásobný.
 7. Theta(k^n), kde k je reálné kladné číslo, je označení algoritmů s **exponenciální** (**pro $k=2$ binomickou**) časovou složitostí. Existuje několik prakticky použitelných algoritmů s touto složitostí. Velmi často se označují jako algoritmy pracující s "hrubou silou" (brute-force algorithms). **Když u binomické časové složitosti n vzroste o 1, čas se zdvojnásobí.**
- Pozn. Příklady lze nalézt v grafových algoritmech, např. v problému obchodního cestujícího.

Následující příklad demonstruje vliv rádu složitosti a velikost na celkový čas potřebný pro výpočet algoritmu. V horní části tabulky jsou hodnoty času potřebné pro algoritmus o daném rádu a dané velikosti zpracovávaných dat n .

V dolní části tabulky jsou jako ukázka souvislosti rádu a času uvedeny maximální hodnoty velikosti dat n tak, aby algoritmus skončil za 1 s nebo za 1 minutu.

řad / velikost	$33n$	$46 n \log n$	$13 n^2$	$3.4 n^3$	2^n
10	0.000333 s	0.015 s	0.0013 s	0.0034 s	0.001 s
100	0.0033 s	0.03 s	0.13 s	3.4 s	4.1014 století
1000	0.033 s	0.45 s	13 s	94 hod	
10 000	0.33 s	6.1 s	22 min	39 dní	
100 000	3.3 s	1.3 min	1.5 dní	108 roků	
Maximální velikost n pro					
1 s	30 000	2 000	280	67	20
1 min	1 800 000	82 000	2 200	260	26

V následující tabulce je ukázka porovnání nejvýkonnějšího počítače a jednoho z prvních osobních počítačů z 80. let. Tabulka zdůrazňuje význam rádu algoritmu. Z tabulky vyplývá závěr, že pro daný problém může být významnější hledat zlepšení rádu algoritmu než zrychlování počítače.

	CRAY – 1 Fortran	TRS-80 Basic
n	3n3	19 500 000 n
10	3 x10-6 s	200 x10-3 s
100	3 x10-3 s	2 s
1 000	3 s	20 s
2 500	50 s	50 s
10 000	49 min	3.2 min
1 000 000	95 let	5.4 hod

2.5 Otázky a úkoly

úkoly



1. Co je to syntax?
2. Co je to sémantika?
3. Pro I a J typu integer zapište význam příkazu $I := : J$
 - a. s využitím pomocné proměnné P
 - b. bez využití pomocné proměnné P
4. V které fázi zpracování programu se zpracovává pseudopříkaz?
5. V které fázi zpracování programu se zpracovává příkaz?
6. Co je to "skalární typ"?
7. Definujte "ordinální" typ a operace nad ním definované.
8. Co je to dyadickej operátor?
9. Jak se mění rychlosť změny indexu při průchodu vícerozměrným polem "po sloupcích"?
10. Jak se říká komponentám homogenní datové struktury?
11. Jak se říká komponentám heterogenní datové struktury?
12. Jak se vymění hodnoty dvou aritmetických proměnných bez pomocné proměnné?
13. Jaká je ordinální hodnota prvního prvku výčtového typu?
14. Definujte pojem výraz v algoritmickém jazyce.
15. Napište výraz pro převod kladné celočíselné jednocyferné hodnoty na znak (číslici).
16. Co je to monoadický operátor a uveďte dva příklady takových operátorů
17. Co znamená uspořádání nebo průchod "po řádcích" u vícerozměrného pole?
18. Co je to konstruktor?
19. Který strukturovaný typ je potenciálně nehomogenní (heterogenní)?

20. Co znamená hodnota "nil" a jakým proměnným lze přiřadit?
21. K čemu slouží středník?
22. Co je to prázdný příkaz?
23. Co je to složený příkaz?
24. Na kterém místě nesmí stát středník?
25. Jakého typu smí být řídicí výraz příkazu case?
26. Jakého typu smí být řídicí proměnná (počítadlo) počítaného cyklu "for"?
27. Jaká je hodnota řídicí proměnné cyklu "for" po řádném skončení cyklu?
28. Jakého typu by měl být složený booleovský výraz v implicitním cyklu "while"?
29. Jakého typu by měl být složený booleovský výraz v implicitním cyklu "repeat-until"?
30. Z čeho sestává deklarace procedury (funkce)?
31. Z čeho se skládá hlavička procedury?
32. Z čeho se skládá hlavička funkce?
33. Jaký je rozdíl mezi procedurou a funkcí při jejich deklaraci?
34. Jaký je rozdíl mezi procedurou a funkcí při jejich volání?
35. Co jsou formální parametry a čím se oddělují skupiny stejných parametrů v seznamu formálních parametrů?
36. Co jsou skutečné parametry a čím se oddělují v seznamu skutečných parametrů?
37. Jaký je vztah mezi seznamem formálních a skutečných parametrů též procedury (funkce)?
38. Definujte účinek volání procedury (funkce).
39. Co je to rekurzívní volání procedury (funkce)?
40. Napište proceduru, která nalezně největší faktoriál, právě menší nebo roven 100.
41. Napište proceduru, která spočítá průměrnou délku a rozptyl všech neklesajících posloupností v poli čísel typu integer (nebo v řetězci string). Rozptyl je "průměrná hodnota kvadratických odchylek hodnot prvků od průměrné hodnoty všech prvků". Nalezněte algoritmus, jak spočítat rozptyl v jednom průchodu (cyklu).
42. Co vyjadřují složitosti **O**, **Ω** a **Θ**?
43. Znázorněte graficky složitost O
44. Které ze tří uvedených složitostí je z praktického hlediska nejpoužívanější?



Závěr

Závěr

Kapitola shrnuje základní vlastnosti algoritmického jazyka pascalovského typu, který bude použit při výkladu algoritmů. Je určena zejména pro studenty, kteří se se základy programování seznámili v jiném programovacím jazyku.
Kapitola je doplněna základními pojmy složitosti a příklady jejího významu.

3 ADT**3 ABSTRAKTNÍ DATOVÉ TYPY - ADT**

Učební cíle a kompetence  3. kapitola představuje nejvýznamnější část předmětu IAL. Student se v ní seznámí s jedním z nejvýznamnějších pojmu moderního programování - abstraktním datovým typem. Seznámí se s principy dynamického přidělování paměti (DPP) a naučí se implementovat jeho model polem. Naučí se využívat i navrhovat a konstruovat typické ADT. Významnou částí kursu, , je naučit se vytvářet vlastní abstraktní datové struktury a typy.

Anotace kapitoly  Principy dynamického přidělování paměti (DPP) . Abstraktní datový typ (ADT), jeho definice a nástroje syntaktické a sémantické specifikace. Typické ADT: seznamy, jednosměrný seznam, dvojsměrný seznam, kruhový seznam, zásobník, fronta, pole, řetězec, tabulka, graf.

Datová struktura  **Datová struktura** je abstraktní vyjádření zúčastněných vlastností (atributů) prvků (objektů) řešeného problému.

Je-li např. prvkem "student" a řešeným problémem "studijní informační systém", pak datovou strukturou vyjadřující zúčastněné vlastnosti může být soustava údajů např. o jménu, adrese a jiných osobních datech studenta, o datu jeho zahájení studia, o jeho zařazení do ročníku a oboru studia, o dosažených výsledcích, o absolvovaných kurzech, o zapsaných kurzech apod. Vlastnosti studenta, které jsou pro daný problém nepodstatné (irrelevantní), nejsou součástí datové struktury pro tento problém.

Struktura není vrozenou vlastností objektů reálného světa. Je to nástroj, kterým lidský intelekt snadněji ovládne rozsáhlou a složitou celistvost reálného světa tím, že ji rozdělí na menší celky - komponenty podle určitého systému pro uspořádání a vzájemné vazby těchto komponent.

3.1 DPP**3.1 Principy dynamického přidělování paměti**

Staticky se paměť přiděluje deklarovaným datovým strukturám v době překladu deklaračního úseku programu. Jednotlivé paměťové úseky jsou přístupné prostřednictvím jména proměnné, uvedeného v deklaraci. Počet ani uspořádání komponent statické struktury nelze v průběhu chodu programu měnit.

Dynamicky přiděluje paměť systém na základě požadavku vzniklého v době řešení programu. Paměťový úsek přidělený dynamicky je přístupný výhradně nepřímo, prostřednictvím ukazatele, který je součástí statické nebo dynamické struktury.

Paměť přidělovaná dynamicky se čerpá z vyhrazeného prostoru paměti počítací až do jeho vyčerpání. Při vyčerpání takto vyhrazené paměti dochází zpravidla k předčasnemu ukončení programu a v případě zdůvodněných nároků na tak velkou paměť je pro úspěšné provedení programu zapotřebí systémově vyhradit větší paměť pro DPP.

Principy dynamického přidělování paměti (*Dynamic Memory Allocation - DMA*) lze dělit podle toho, jakým způsobem se přiděluje (alokuje) paměťový úsek v operaci typu "new" a zda a jak se regeneruje při uvolnění (dealkaci) dále nepoužívaný úsek paměti při operaci typu "dispose". V souvislosti s použitým mechanismem DPP se vyhrazené paměti

v některých systémech říká také "heap" - hromada.

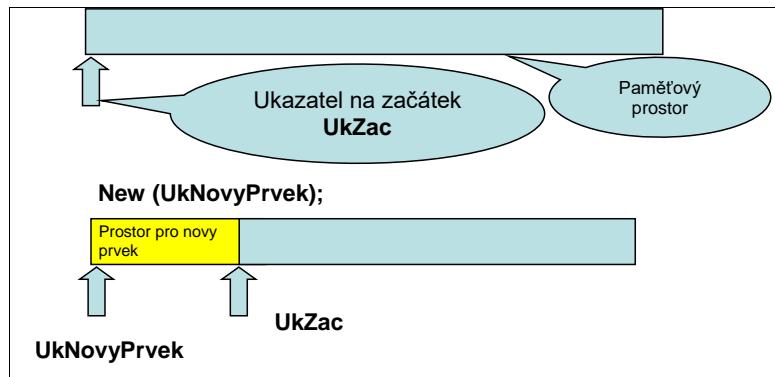
- Dynamické proměnné (struktury) vznikají i zanikají v době běhu programu.
Nemohou mít jméno (identifikátor). K obsahu dynamických proměnných (struktur) se dostáváme výhradně prostřednictvím ukazatele.
- Počet i uspořádání komponent dynamických struktur se za běhu programu mění.
- Příkladem dynamické struktury je seznam nebo stromová struktura.
- Ke tvorbě dynamické struktury je vhodné (nutné) použít datového typu „záznam“. Jeho heterogennost umožňuje, aby dynamický prvek obsahoval vedle vlastní hodnoty také ukazatel(e).

Operace new

Operace "new"

- Operace new(Uk) přiřadí ukazateli Uk hodnotu „ukazující“ na paměťový prostor, jehož velikost odpovídá datovému typu s nímž je ukazatel Uk svázán. Paměťový prostor pro operaci „new“ se vyhrazuje v paměťové oblasti, vyhrazené pro tento účel.
- Hodnota ukazatele lze přiřadit jen ukazateli téhož typu.
- "Nil" je ukazatlová konstanta s hodnotou vyjadřující, že ukazatel neukazuje na žádnou proměnnou (strukturu). Tuto konstantu lze přiřadit ukazateli libovolného typu.

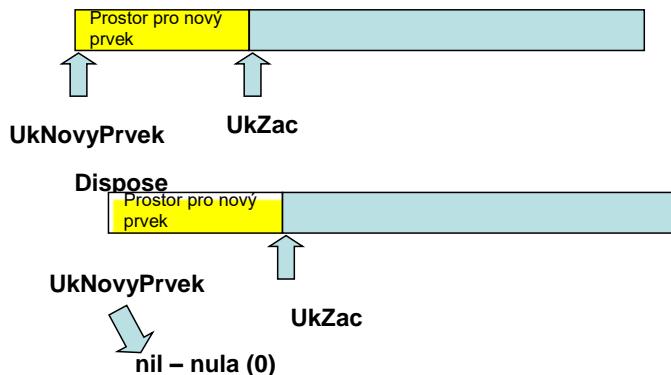
Operace new



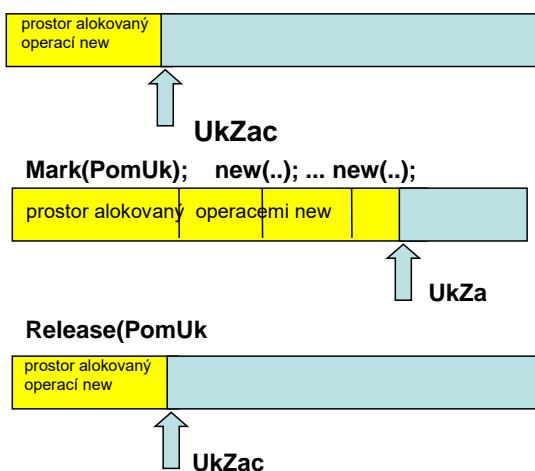
dispose

Operace "dispose"

- Operace Dispose(Uk) ruší použitelnost dynamické proměnné (struktury), na kterou ukazuje ukazatel Uk. Po této operaci je hodnota ukazatele undefined.
- Pokud se paměť, kterou zaujímá zrušená proměnná (struktura) vrátí do vyhrazené paměťové oblasti, říkáme, že DPP pracuje s **regenerací paměti**. V jiném případě jde o mechanismus **bez regenerace paměti**.



Některé systémy "bez regenerace" umožňují zjednodušenou regeneraci vracené paměti operacemi typu "**mark**" a "**release**". Operace "mark" zaznamená stav ukazatele na začátek volné paměti a operace "release" nastaví ukazatel zpět na stav zaznamenaný operací "mark" a tím uvolní úsek paměti přidělený mezi operací "mark" a "release". princip mechanismu znázorňuje následující obrázek.

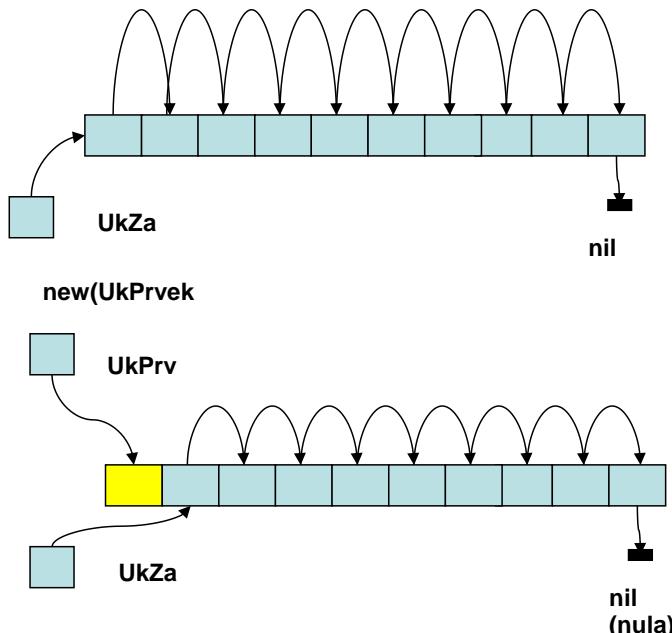


Příklad: Procedura, která vytváří, použije a dále již nepotřebuje dynamickou strukturu zanamená na začátku těla "mark" a před jeho ukončením provede "release"

```
procedure P(...);
...
begin
    mark;
    (* vlastní tělo procedury *)
    release;
end;
```

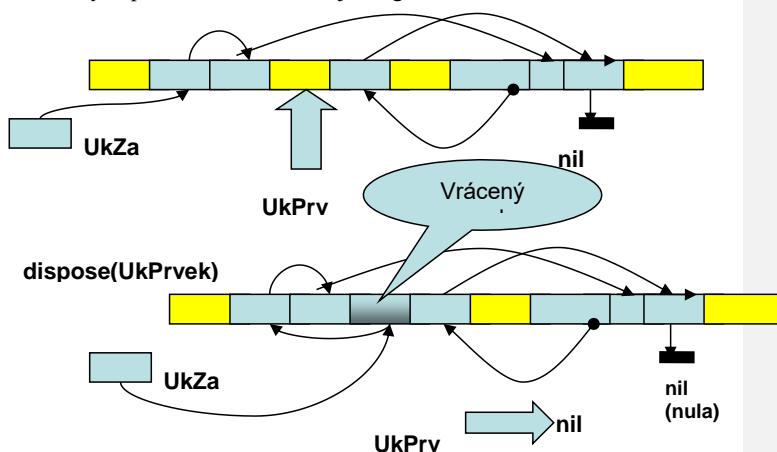
DPP s regenerací používá pole volných prvků zřetězených do jednoho seznamu pomocí ukazatelových indexů. Na počátku jsou všechny prvky volné. Pomocný

ukazatlový index ukazuje na první volný prvek, který se uvolní a přidělí operací **new** ukazateli. Situaci znázorňuje následující obrázek.



dispose

Po čase se situace opakujícími se operacemi new a dispose může vyvinout do stavu na následujícím obrázku, kdy je v poli seznam volných zřetězených prvků (nemusí to být sousedící prvky) a zbyvající (žluté) prvky jsou v užívání dynamických struktur. Při uvolňování prvku, na nějž ukazuje ukazatel **UkPrvek** se uvolňovaný prvek vloží na začátek seznamu volných prvků a tím se realizuje "regenerace".



	Implementace uživatelského modelu DPP (UDPP).
UDPP	Implementace uživatelského modelu DPP (<i>Dynamic Memory Allocation DMA</i>) používá pole. Operace pro UDPP se odlišují příponou
typy a proměnné	<pre>const HeapLength=100; (* Velikost prostoru pro UDPP *) type TDMA = record (* UDPP pro max. 100 prvků *) FreePtr, (* Ukazatel(index)prvního volného *) MarkPtr: (* Ukazatel pro „mark“*) integer; ArrItem: (* dynamický prostor*) array [1..HeapLength] of TItem; end; (* record *)</pre>
InitDMA	Pro účel implementace vytvoříme prostor pro DPP ve formě globální proměnné: <pre>var DMA:TDMA; (* globalní proměnná *)</pre>
	procedure InitDMA; (* Procedura nastaví počáteční hodnoty indexových ukazatelů. Musí se vyvolat před použitím ostatních operací nad UDPP *) <pre>begin with DMA do begin FreePtr:=1; end end;</pre>
NewDMA	procedure NewDMA(var Ptr:integer); (* Procedura vrací indexový ukazatel prvního volného prvku pole, který bude přidělen jako dynamický prostor *) <pre>begin with DMA do begin Ptr:=FreePtr; (* Vracený indexový ukazatel *) FreePtr:=FreePtr+1; (* Zvýšení hodnoty ukazatele *) end; end;</pre>
Dispose DMA	procedure DisposeDMA(Ptr:integer); (* Tato procedura je prázdná, protože nepoužíváme indikátor zrušeného prvku. Vytváříme ji pro kompatibilitu modelu s běžným DPP *) <pre>begin end;</pre>
Mark DMA	procedure MarkDMA; (* Uschování hodnoty FreePtr do MarkPtr *) <pre>begin with DMA do begin MarkPtr:=FreePtr; end; end;</pre>

```

Release DMA   procedure ReleaseDMA;
                  (* Znovuustavení uschované hodnoty do FreePtr *)
begin
  with DMA do begin
    FreePtr:=MarkPtr;
  end
end;

UDPP s regenerací Implementace UDPP s regenerací s použitím zřetězeného pole na způsob zásobníku.

typy a proměnné Pro implementaci použijeme následující typy a proměnné:
const
  LengthOfHeap=100;
  NilDMA=0;  (* Nula reprezentuje nil*)
type
  TDMA = record
    StartPtr : integer;
    ArrItem: array [1..LengthOfHeap] of TItem;
    ArrOfLinks:array[1..LengthOfHeap] of integer;
                (* pole spojovacích ukazatelů *)
  end; (* record *)

var
  DMA:TDMA; (* Globální paměťový prostor *)

InitDMA   procedure InitDMA;
(* Procedura spojuje prvky do seznamu.Ukazatel StartPtr ukazuje na první prvek. Poslední
ukazuje „nikam“ – nil reprezentovaný nulou. *)
var
  i:integer;
begin
  with DMA do begin
    for i:=1 to LengthOfHeap-1 do
      ArrOfLinks[i]:=i+1;
      (* průběžné spojení prvků *)
    ArrOfLinks[LengthOfHeap]:=NilDMA;
      (* nastavení posledního na nil *)
    StartPtr:=1;
      (* nastavení ukazatele na začátek *)
  end; (* with *)
end;

NewDMA   procedure NewDMA(var Ptr:integer);
begin
  with DMA do begin
    Ptr:=StartPtr;  (* vrací ukazatel prvního *)
    StartPtr:=ArrOfLinks[StartPtr]  (* Ukazatel prvního se
                                    posune na další *)
  end;

```

```

    end (* with *)
end;

procedure DisposeDMA(Ptr:integer);
begin
  with DMA do begin
    ArrOfLinks[Ptr]:=StartPtr; (* Ukazatel disposovaného
                                prvku je nastaven na aktuálního prvního *)
    StartPtr:=Ptr;           (* aktualizace prvního *)
  end (* with *)
end;

```

Implementace dynamických struktur v modelovém systému UDPP. Námi vytvořený modelový systém UDPP lze použít podobně jako standardní systém DPP. Namísto ukazatelů se bude UDPP pracovat s polem a indexy namísto ukazatelů. Standardní zápis úseku programu s ukazateli lze rutinně přepsat s použitím následujících transformačních vztahů:

Zápis s ukazatelem - DPP	Zápis s indexem - UDPP
Ptr je ukazatel	PtrI je index
nil	NilDMA (const NilDMA=0)
Ptr	PtrI
Ptr^	DMA.ArrItem[PtrI]
Ptr^ RPtr	DMA.ArrItem[PtrI].RPtr
Ptr^ RPtr^ LPtr	DMA.ArrItem[DMA.ArrItem[Ptr I].RPtr].LPtr

Případová studie

x+y

Případová studie (case study). Vytvořte proceduru, která v dvojsměrném seznamu seznamu znaků zadáném ukazatelem na začátek i konec seznamu zruší prvek zadáný ukazatelem. Vytvořte variantu s užitím DPP a ukazateli a variantu s užitím UDPP a indexovými ukazateli.

Typy DPP Potřebné typy a proměnné pro DPP s ukazateli:

```

type
  TListPtr=^TItem; (* typ ukazatele na prvek seznamu *)
  TItem=record          (* typ prvek seznamu *)
    data:char;
    LPtr,RPtr:TListPtr (* levý a pravý ukazatel prvku *)
  end;
  TList=record          (* type seznam - List *)
    Frst,Lst:TListPtr (* Ukazatele na první a poslední prvek seznamu *)
  end;

```

```

Verze DPP procedure deleteDMA(var L:TList; Ptr:TListPtr);
(* Ptr ukazuje na rušený prvek.Procedura s ukazateli pascalu *)
begin
  if Ptr<>nil then begin (* je ukazatel rušeného nenilový? *)
    if (Ptr=L.Frst) and (Ptr=L.Lst)
    then begin (* rušený je jediným prvkem *)
      L.Frst:=nil; L.Lst:=nil; (* rušení jediného *)
    end else begin (* rušený není jediným prvkem*)
      if (Ptr=L.Frst)
      then begin (* rušený je prvním prvkem *)
        L.Frst:=Ptr^.RPtr;
        Ptr^.RPtr^.LPtr:=Ptr^.LPtr; (* :=nil *)
      end else begin
        if (Ptr=L.Lst)
        then begin (* rušený je posledním prvkem *)
          L.Lst:=Ptr^.LPtr;
          Ptr^.LPtr^.RPtr:=Ptr^.RPtr (* :=nil*)
        end else begin (* rušený má oba sousedy *)
          Ptr^.LPtr^.RPtr:=Ptr^.RPtr;
          Ptr^.RPtr^.LPtr:=Ptr^.LPtr
        end (* Ptr=L.Lst *)
      end (* Ptr=L.Frst *)
    end; (* (Ptr=L.Frst) and (Ptr=L.Lst) *)
    dispose(Ptr);
  end (* Ptr<>nil *)
end;

```

typy UDPP Potřebné typy a proměnné pro modelové **UDPP** s polem a indexovými ukazateli:

```

const
  NilUDMA=0; (* Uživatelský nil, representovaný nulou *)
type
  TListPtr=integer; (* typ indexový ukazatel *)
  TItem=record (* typ položky dvojsměrného seznamu *)
    data:char;
    LPtr, RPtr:TListPtr
  end;
  TList=record (* typ dvojsměrný seznam definovaný ukazateli
                 na začátek a konec *)
    Frst, Lst:TListPtr
  end;

```

ArrItem – je pro zjednodušení jméno pole (DMAArrItem) , v němž je realizován dynamický prostor.

**Verze
UDPP**

```

procedure deleteDMA(var L:TList; Ptr:TListPtr);
(* Ptr ukazuje na rušený prvek. Procedura s indexovými ukazateli a UDPP *)
begin
  if Ptr<>NilUDMA then begin (* je ukazatel rušeného nenilový? *)
    if (Ptr=L.Frst) and (Ptr=L.Lst)
    then begin (* rušený je jediným prvkem *)
      L.Frst:=NilUDMA; L.Lst:=NilUDMA; (* rušení jediného *)
    end else begin (* rušený není jediným prvkem*)
      if (Ptr=L.Frst)
      then begin (* rušený je prvním prvkem *)
        L.Frst:=ArrItem[Ptr].RPtr;
        ArrItem[ArrItem[Ptr].RPtr].LPtr:=NilUDMA
      end else begin
        if (Ptr=L.Lst)
        then begin (* rušený je posledním prvkem *)
          L.Lst:=ArrItem[Ptr].LPtr;
          ArrItem[ArrItem[Ptr].LPtr].RPtr:=NilUDMA
        end else begin (* rušený má oba sousedy *)
          ArrItem[ArrItem[Ptr].LPtr].RPtr:=ArrItem[Ptr].RPtr;
          ArrItem[ArrItem[Ptr].RPtr].LPtr:=ArrItem[Ptr].LPtr
        end (* Ptr=L.Lst *)
      end (* Ptr=L.Frst *)
    end; (* (Ptr=L.Frst) and (Ptr=L.Lst) *)
    disposeDMA(Ptr);
  end (* Ptr<>NilUDMA *)
end;

```

Jak je vidět z porovnání obou verzí, text verze UDPP lze získat mechanickým přepisem s použitím transformační tabulky pro převod DPP a UDPP

Kontrolní úkoly:

- Je dán jednosměrný seznam s datovými typy pro DPP. Je dán ukazatel na první prvek jednosměrného seznamu. Vytvořte proceduru, která získá počet prvků tohoto seznamu.
- Je dán jednosměrný seznam s datovými typy pro UDPP implementovaný prvky v poli. Je dán indexový ukazatel na první prvek jednosměrného seznamu. Vytvořte proceduru, která získá počet prvků tohoto seznamu. Porovnejte zápis procedury s předcházejícím příkladem.
- Je dán jednosměrný seznam implementovaný ukazateli. Navrhněte způsob, jak zrušit prvek zadáný ukazatelem (bez průchodu seznamem). Je nějaké omezení?
- Je dán jednosměrný seznam implementovaný ukazateli. Navrhněte, jak vložit nový prvek před prvek zadáný ukazatelem. Je nějaké omezení?

3.2 ADT

3.2 Abstraktní datový typ, základní pojmy

Abstraktní datový typ (ADT) je definován množinou hodnot, jichž mohou nabývat prvky tohoto typu a množinou operací, definovaných nad tímto typem.

DEF

Abstraktní datový typ representuje třídu prvků se shodnými vlastnostmi. Jednomu konkrétnímu prvku takového typu říkáme někdy "proměnná daného typu", "prvek daného typu", "instance daného typu", "výskyt daného typu", nebo také "abstraktní datová struktura daného typu" - ADS.

Pozn. ADT zdůrazňuje vnější vlastnosti a chování svých prvků a potlačuje způsob, jakým jsou tyto vlastnosti reprezentovány na nižší (např. strojové, paměťové) úrovni a jakým způsobem je jejich chování (operace) implementováno. Připomínají tedy "černou skříňku" (black box), o níž víme, co bude mít na výstupu, je-li definováno, co má na vstupu. Přitom nevíme nebo nás ani nezajímá, jakým mechanismem se definovaná transformace vstupních hodnot na výstupní hodnoty realizují.

x+y

Standardním typem Pascalu je např. typ real. Je definován číselným rozsahem hodnot, jichž smí nabývat a množinou operací, které se s typem real mohou provádět. Přitom nás nezajímá, že typ real je složen ze dvou komponent - mantisy a exponentu, ke kterým nemáme přímý přístup, ani nepotřebujeme znát mechanismus, jakým se provádějí např. aritmetické operace nad tímto typem.

Pozn. Je zvláštní vlastností Pascalu, že dovoluje, aby operátor **+** měl vícenásobnou sémantiku a mohl být použit nejen pro součet čísel typu real, ale také pro součet čísel typu integer i k jiným aditivním operacím. Za normálních okolností má ADT svou jedinečnou množinu operací. Její operace nelze použít nad jiným typem a nad daným ADT nelze použít operací jiných ADT.

DEF

Generický abstraktní datový typ je definován pouze množinou operací nad typem definovaných. Generický ADT nespecifikuje konstituční typ. Používá se nejčastěji k vytvoření (generování) konkrétního ADT dodáním typu komponenty.

Pozn. Řekneme-li např. "vektor o 10 prvcích", hovoříme o generickém ADT, protože jsme nespecifikovali typ prvku.

DEF

Pro práci s ADT jsou důležité některé významné pojmy, jako:

konstruktor

- Konstruktor je operace, jejížmž vstupními parametry je výčet (všech) komponent struktury a výsledkem operace je datová struktura obsahující tyto komponenty. Konstruktor "vytvorí" datovou strukturu z jejich komponent. Příkladem konstruktoru v jazyce Pascal je textový řetězec v příkazu str='Textovy retezec' nebo množina [1,3,5,7,9]

selektor

- Selektor je operace, která umožní přístup k jednotlivé komponentě datové struktury na základě uvedení jména struktury a zápisu přístupu ("reference"). Příklad selektoru nad textovým řetězcem z předcházejícího odstavce je zápis str[3], kde prvek řetězce má hodnotu 'x'. Přístup k prvku datové struktury se používá za účelem změny hodnoty prvku nebo k získání jeho hodnoty.

iterátor

- Iterátor je operace, která provede zadanou činnost nad všemi prvky homogenní datové

destruktor

struktury. Příklad: většina složitých grafických útvarů je realizována seznamem grafických elementů, z nichž je útvar sestaven. Operace, která jediným příkazem typu iterátor provede operaci "vykreslit" nad všemi elementy, zobrazí útvar jako celek.

syntax
sémantika

- Destruktor je operace nad dynamickými strukturami. Tato operace zruší dynamickou strukturu a vrátí prostor jí zaujímaný systému DPP.

Při návrhu nového ADT se musí specifikovat jeho syntax a sémantika. Tím je navržený ADT zcela definován. Pokud jsou nástroje specifikace matematicky formální, mohou být použity k různým formálním matematickým postupům.

Často používaným nástrojem specifikace syntaxe je algebraická signatura a diagram signatury.

x+y

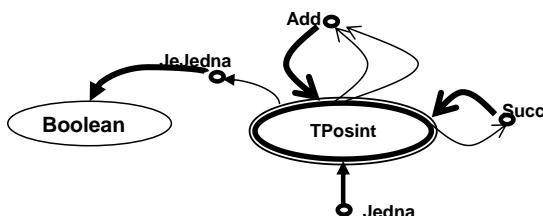
Příklad: Mějme ADT zvaný PosInt (pozitivní celočíselný typ větší než nula), nad nímž jsou definovány tyto základní operace: ustavení hodnoty "jedna" ("One"), inkrementace, ("Succ"), součet dvou prvků tohoto typu ("Add") a predikát zjištění, zda má prvek typu PosInt hodnotu "jedna" ("IsOne"). Pak algebraická signatura syntaxe operací má tvar:

algebraická
signatura

- 1) One : \rightarrow PosInt
Této operaci se říká generátor nebo "inicializace". Operace nemá žádný vstupní parametr a jejím výstupem je hodnota typu PosInt .
- 2) ADD : PosInt \times PosInt \rightarrow PosInt
Operace má dva vstupní parametry typu PosInt. Jakékoli dvě hodnoty typu PosInt na vstupu dávají na výstup jednu hodnotu typu PosInt.
- 3) SUCC : PosInt \rightarrow PosInt
Vstupní hodnota i výstupní hodnota je typu PosInt
- 4) IsOne : PosInt \rightarrow Boolean
Operace typu "predikát" má na vstupu hodnotu typu PosInt a na výstupu hodnotu typu Boolean.

Diagram
signatury

Diagram signatury je grafický přepis algebraické signatury. Každý typ je zobrazen oválem. Specifikovaný ADT je zobrazen oválem zvýrazněným tučnou čarou. Každá operace je zobrazena kroužkem. Šipky znázorňují vstupní a výstupní parametry operace. Několik operací se stejnou syntaxí může být reprezentováno jedním kroužkem. Operaci, která nemá vstupní parametr a má výstupní parametr(y) se říká "**generátor**" a používá se k prvotnímu vytvoření - inicializaci ADT. Diagram signatury pro specifikaci ADT PosInt je uveden na následujícím obrázku:



sémantika Sémantiku ADT lze specifikovat slovním popisem, operačním popisem nebo systémem

axiomů nebo jinými speciálními specifikačními nástroji a systémy.

Slovní vyjádření sémantiky ADT PosInt:

$x+y$

- Operace One ustaví hodnotu typu Posint rovnu jedné. Tato operace je inicializace typu (generátor).
- Operace ADD vytvoří aritmetický součet dvou prvků typu Posint.
- Operace Succ vytvoří hodnotu následující danou hodnotu (hodnotu o jednu větší).
- Operace (predikát) IsOne nabude hodnoty true, pokud je argument hodnota rovna jedné. V jiných případech má operace hodnotu false.

Axiomatická specifikace má tvar:

1. $\text{ADD}(X,Y) = \text{ADD}(Y,X)$
2. $\text{ADD}(\text{One},X)=\text{SUCC}(X)$
3. $\text{ADD}(\text{SUCC}(X),Y)=\text{SUCC}(\text{ADD}(X,Y))$
4. $\text{IsOne}(\text{One})=\text{true}$
5. $\text{IsOne}(\text{SUCC}(X))=\text{false}.$

Operační specifikace má nejčastěji tvar procedur/funkcí popisujících chování operace ve zvoleném algoritickém nebo programovacím jazyku.

Pozn. Nevýhodou této specifikace je, že chování ADT vyjadřuje konkrétní implementaci, která nemusí být jedinou možnou implementací takového chování a čtenář podkládá pohled na vnitřní uspořádání, které by mělo být skryto.

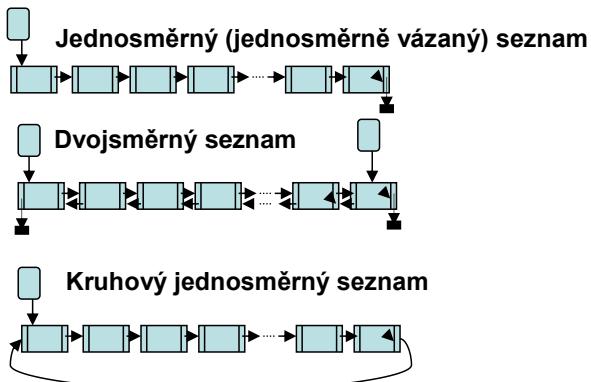
Nástroje specifikace sémantiky jsou významnou oblastí současného výzkumu.

3.2.1
Seznamy
DEF

3.2.1 Seznamy

Seznam je homogenní, lineární, dynamická struktura.

- Lineárnost znamená, že každý prvek struktury má právě jednoho předchůdce (predecessor) a jednoho následníka (successor). Výjimku tvoří první a poslední prvek.
- Prvkem seznamu může být libovolný jiný datový typ – také strukturovaný. Např. seznam seznamů.
- Seznam může být prázdný.
- Přístup k prvnímu (okrajovému) prvku je přímý, přístup k ostatním prvkům seznamu je sekvenční ve směru průchodu. Seznam, který lze procházet jen jedním směrem se nazývá "jednosměrně vázaný" zkráceně "jednosměrný", Seznam u něhož lze procházet oběma směry je "dvousměrně vázaný" - "dvousměrný".



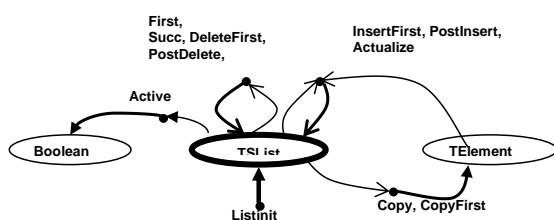
Jedno-směrný seznam

DEF

ADT jednosměrný seznam může být navržen s různými repertoáry operací. Operace seznamu v tomto učebním textu je založena na principu tzv. "aktivního prvku seznamu".

- Aktivita je vlastnost prvku seznamu
- Jen jeden prvek seznamu může být aktivní. Seznam, který obsahuje aktivní prvek je aktivním seznamem. Seznam, který neobsahuje žádný aktivní prvek, je neaktivní.
- Operace First způsobí, že první prvek neprázdného seznamu se stane aktivním, bez ohledu na předchozí stav. Tato operace nad prázdným seznamem nemá účinek.
- Operace Succ nad aktivním seznamem způsobí, že se aktivita přenese na následující prvek. Pokud byl aktivní prvek posledním ve směru průchodu, aktivita se ztratí, seznam přestane být aktivní. V případě neaktivního (prázdného) seznamu nemá operace Succ účinek.
- Diagram signatury na následujícím obrázku specifikuje syntaxi operací ADT jednosměrný seznam. Repertoár operací sestává z následujících operací: ListInit(L), InsertFirst(L,E), CopyFirst(L,E), DeleteFirst, First(L), Succ(L), Active(L), PostInsert(L,E), PostDelete(L), Copy(L,E), Actualize(L,E) (kde L je seznam a E je prvek).

Pozn. Abychom odlišili jednotlivé typy seznamů, budeme identifikátory vztahující se k jednosměrnému seznamu pro odlišení od ostatních typů seznamů uvozovat případně předponou "S" ("single-linked"), u dvojsměrného seznamu předponou "D" ("double-linked") a u kruhového seznamu předponou "C" ("circular list")



Sémantika operací

Jména dále uvedených operací mohou být pro jednosměrný seznam případně uvedena s předponou "S", např. SListInit, SInsertFirst apod. Parametr L je seznam - parametr typu TList (TSList) a E je prvek - parametr typu TEElement.

- ListInit(L) inicializace jednosměrného seznamu, vytvoření nového prázdného seznamu
- InsertFirst(L,E) Vložení prvku E na začátek seznamu L. Jediná možná vložení prvku

do prázdného seznamu. Pozn. Vložený prvek není aktivní!

- CopyFirst(L,E) Operace vrací ve výstupním parametru E hodnotu prvního prvku.
Operace způsobí chybu a ukončí program, je-li seznam prázdný! Této situaci je nutno předcházet testem na neprázdnost seznamu!
- First(L) Nad prázdným seznamem bez účinku. Nad neprázdným seznamem, ustavení prvního prvku aktivním. Pozn. Neaktivní seznam se stane aktivním!
- DeleteFirst(L) Na prázdném seznamem bez účinku. Nad neprázdným seznamem se zruší první prvek. Byl-li aktivní, zruší se i s aktivitou a seznam přestane být aktivní.
- Succ(L) Nad neaktivním seznamem bez účinku. Nad aktivním seznamem - posun aktivity na následující prvek. Byl-li aktivním poslední, ztráta aktivity seznamu.
- Active(L) Predikát aktivity seznamu L. V případě, že je seznam aktivní, vrací funkce hodnotu "true", jinak hodnotu "false"
- PostInsert(L,E) Nad neaktivním seznamem bez účinku. Nad aktivním seznamem, vložení prvku E za aktivní prvek. Pozn. Aktivita původního prvku je zachována!
- PostDelete(L) Pokud je seznam neaktivní nebo pokud je aktivní prvek poslední, nemá operace žádný účinek. V jiném případě zruší operace prvek za aktivním prvkem. **Rušená hodnota se ztrácí (nevrací se)!**
- Copy(L,E) Operace vrací ve výstupním parametru E hodnotu aktivního prvku.
Operace způsobí chybu a ukončí program, je-li seznam neaktivní! Této situaci je nutno předcházet testem na aktivnost seznamu!



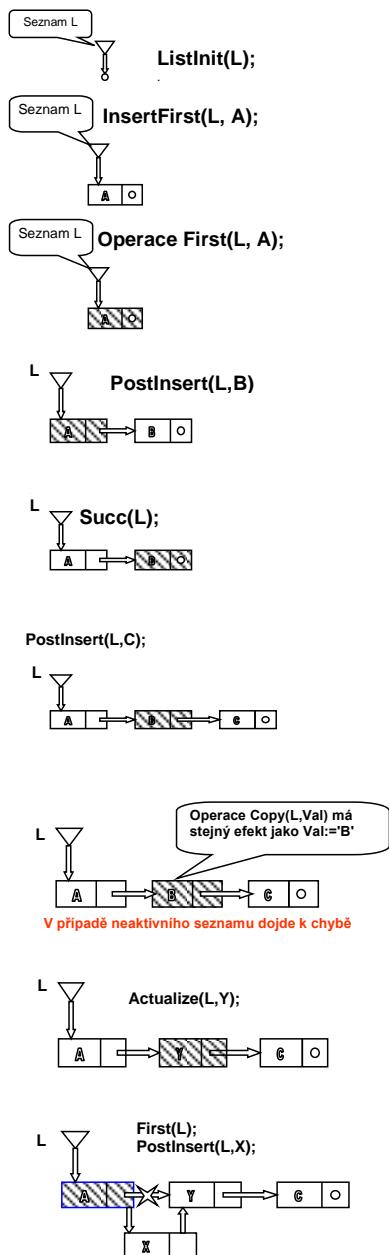
Pozn. Chybouvý stav je způsoben povinností vrátit hodnotu, která neexistuje. Podobný stav nastává u všech operací s podobnou vlastností.

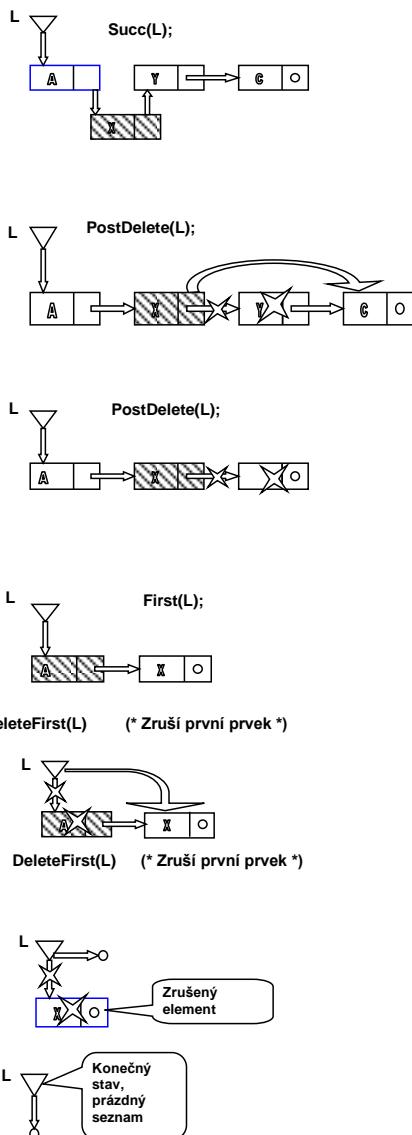
- Actualize(L,E) Operace je nad neaktivním seznamem bez účinku. V jiném případě prepíše hodnotu aktivního prvku hodnotou vstupního parametru E.

Dvojsměrný seznam **Dvojsměrný seznam je stranově symetrická datová struktura.** Operace dvojsměrného seznamu lze získat doplněním repertoáru jednosměrného seznamu o opačně stranově orientované operace se shodnou, stranově opačně orientovanou sémantikou:

SInsertFirst	→ (DInsertFirst), DInsertLast	Vlož prvek jako poslední
SFirst	→ (DFirst), DLast	Nastav aktivitu posledního
SDeleteFirst	→ (DDeleteFirst), DDeleteLast	Zruš poslední
SCopyFirst	→ (DCopyFirst), DCopyLast	Čti hodnotu posledního
SSucc	→ (DSucc), DPred	Posuň aktivitu vlevo
SPostInsert	→ (DPostInsert), DPredInsert	Vlož před aktivního
SPostDelete	→ (DPostDelete), DPreDelete	Zruš prvek před aktivním

Ilustrace účinků postupných operací:





Typické operace nad ADT seznam

1. Zjištění délky (počtu prvků) seznamu
2. Vytvoření kopie (duplicátu) seznamu
3. Zrušení seznamu
4. Ekvivalence dvou seznamů

DEF

5. Relace (lexikografická) dvou seznamů

Pozn. **Lexikografická relace** dvou lineárních struktur porovnává odpovídající dvojice prvků zleva. V případě rovnosti pokračuje, v případě nerovnosti určuje výsledek relace neshodná dvojice. V případě, že jedna lineární struktura končí dříve než druhá - při rovnosti všech předchozích prvků, je kratší struktura menší. (S použitím této relace vyhledáváme např. ve slovníku.)

6. Vkládání nových prvků do seznamu (na začátek, na pozici danou ukazatelem, pořadím, aktivitou, na konec)
7. Hledání pozice a délky nejdelší neklesající posloupnosti prvků v seznamu
8. Vkládání a rušení podseznamu v seznamu
9. Vyhledávání prvků v seznamu (pořadí prvního nalezeného)
10. Rušení prvků seznamu (prvního, prvku na a za pozici dané ukazatelem, pořadím, aktivitou, posledního)
11. Seřazení prvků seznamu podle velikosti prvků (klíče)
12. Konkatenace (zřetězení) dvou a více seznamů (podseznamů) do jednoho seznamu (např. podseznamů obsahujících textové „slovo“ do „věty“).
13. Dekatenace (rozčlenění) jednoho seznamu na podseznamy (např. rozčlenění textové „věty“ na „slova“)

Příklady

x+y

Funkce pro zjištění prázdnosti seznamu

```
function LEmpty(L:TL) :boolean;
begin
  First(L);
  LEmpty:=not Active(L)
end
```

x+y

Funkce pro zjištění délky jednosměrného seznamu.

```
function Length(L:TL) :integer; (* délka seznamu*)
var Count:integer;
begin
  Count:=0;
  First(L); (* Nastavení prvního jako aktivního *)
  while Active(L) do begin
    Count:=Count+1;
    succ(L)
  end; (* while *)
  Length:= Count
end
```

x+y

Funkce pro ekvivalenci dvou seznamů znaků:

```
function EQUALS(L1,L2:TL) :Boolean;
var
  EQU:Boolean;
  E1, E2: char;
```

```

begin
    EQU:=true;

    First(L1); First(L2); (* aktivizace *)
    (* Cyklus pokračuje, pokud se odpovídající prvky rovnají a seznamy neskončily.
    Cyklus skončí při nerovnosti, nebo při vyčerpání alespoň jednoho seznamu *)
    while Active(L1) and Active(L2) and EQU do begin
        Copy(L1, E1); Copy(L2, E2);
        if E1 = E2
        then begin
            Succ(L1); Succ(L2)
        end else begin
            EQU:= false
        end (*if*)
    end; (*while*)
    EQLists:=EQU and not Active(L1) and not Active(L2)
end;

```

Určení nejdelší neklesající posloupnosti v seznamu znaků.

```

procedure GNDS (L:TL; var Zac, N:integer);
(* Procedura pro nalezení začátku a délky nejdelší neklesající posloupnosti NP (Greatest
Non-Decreasing Sequence - GNDS) v seznamu znaků. Zac je začátek a N je délka nejdelší
neklesající posloupnosti - první v případě více posloupností se stejnou největší délkou *)
var
    PomChar, ActChar :char;
    Pocit, (*průběžné počítadlo pořadí v seznamu *)
    PomZac, (*průběžný začátek NP *)
    PomDelka (*pracovní délka NP *)
        : integer;
begin
    Pocit:=0;    (* Inicializace pracovních proměnných *)
    N:=0;
    First(L); (* Inicializace seznamu a nastavení hodnot pro neprázdný seznam *)
    if Active(L)
    then begin
        Copy(L,PomChar);
        Pocit:=1;
        PomZac:=Pocit;
        N:=1;
        PomDelka:=1;
        Succ(L);
    end; (*if*)

```



```
(* Cyklus začíná u druhého prvku seznamu a končí vyčerpáním seznamu *)
while Active(L) do begin
  Copy(L,ActChar);
  Pocit:=Pocit+1;
  if PomChar <= ActChar
    then begin (* NP pokračuje *)
      PomDelk:=PomDelk+1;
    end else begin
(* NP skončila, přečtený prvek je prvk. další posl. Je nalezená NP lepší než předchozí ? *)
      if PomDelk > N
        then begin
          N:=PomDelk; (* kandidát na největší délku *)
          Zac:=PomZac; (* kandidát na začátek nejdelší NP *)
        end; (*if PomDelk *)
        PomDelk:=1; (* Inicializace délky další posloupnosti*)
        PomZac:=Pocit (* inicializace začátku další posloupnosti *)
      end; (*if PomChar *)
      Succ(L);
      PomChar:=ActChar (* Přečtený prvek je pro další cyklus pomocný *)
    end; (* while *)
  if PomDelk > N
    then (* test na délku poslední posloupnosti *)
  begin N:=PomDelk; Zac:=PomZac; end; (* PomDelk > N *)
end
```

Varianta na předchozí příklad:

V jednom průchodu seznamem najděte průměrnou hodnotu a rozptyl (dispersi) délek všech neklesajících posloupností daného seznamu celých čísel.

Nápověda: Rozptyl D daného souboru hodnot je průměrná hodnota kvadratických odchylek všech hodnot souboru od průměrné hodnoty.

$$D = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2$$

Implementace operací nad ADT seznam.

Jednosměrný seznam

Jedno-směrný seznam : Prvek seznamu je dříve definovaného typu TElem. Kromě datové složky obsahuje ukazatel na další prvek.

Pozn. Na tomto typu závisí, zda může být operace typu Copy funkcí nebo výstup musí být předáván parametrem procedury.

Pozn. Pro ukazatele v seznamu budeme nejčastěji používat identifikátory: NaslUk, UkNasl, NextPtr. Pro ukazatelé doprava PUk nebo RPtr, doleva LUk nebo LPtr.

type

```

TUK=^TElem; (* Typ uk. na prvek jednosm. sezn. *)
TElem = record      (* Typ prvku seznamu *)
    Data:TEL;        (* Datové složky elementu *)
    UKNasl:TUK;     (* Ukazatel na následníka *)
end;
TList=record          (* ADT seznam *)
    Act,            (* Ukazatel na aktivní prvek seznamu *)
    Frst : TUK      (* Ukazatel na první prvek seznamu *)
end;

ListInit   procedure ListInit (var L:TList);
begin
    L.Act:=nil;
    L.Frst:=nil
end; (* ListInit *)

InsertFirst  procedure InsertFirst(var L:Tlist,El:TEL);
var
    UKPomEl:TUK;
begin
    new(UKPomEl);           (* Vytvoření nového prvku *)
    UKPomEl^.Data:=El;      (* Nastavení datové složky *)
    UKPomEl^.UKNasl:=L.Frst; (* Uk tam, kam začátek *)
    L.Frst:=UKPomEl;         (* Zač. ukazuje na nový prvek *)
end; (* InsertFirst *)

PostInsert  procedure PostInsert (var L:Tlist; El:TEL);
var  PomUk:Tuk;
begin
    if L.Act <> nil then begin (*jen pro aktivní seznam*)
        new(PomUk);
        PomUk^.Data:=El;
        PomUk^.UKNasl:=L.Act^.UKNasl; (* Nový ukazuje
                                         tam, kam aktivní *)
        L.Act^.UKNasl:= PomUk      (* Aktivní ukazuje na nového *)
    end;

```

Copy

```

procedure Copy (L:Tlist; var El:TEL);
(* Operace předpokládá ošetření aktivity seznamu ve tvaru: if Active(L) then begin
Copy(L,El); ... Copy v neaktivním seznamu způsobí chybu *)
begin
    El:= L.Act^.Data
end;

```

Actualize

```

procedure Actualize (var L:Tlist; El:TEL);
begin
    if L.Act<> nil
    then begin

```

```

L.Act^.Data:=El
end
end;

PostDelete procedure PostDelete(var L:Tlist);
var
  PomUk:Tuk;
begin
  if (L.Act<>nil)
  then
    if L.Act^.UkNasl <> nil
    then begin (*je co rušit*)
      PomUk:=L.Act^.UkNasl; (*Ukazatel na rušeneho*)
      L.Act^.UkNasl:=PomUk^.UkNasl; (*Překlenutí rušeného*)
      Dispose(PomUk)
    end (*if L.Act^.UkNasl<> *)
    end (*if L.Act <> *)
  end;

```

Ekvivalence dvou seznamů.



Rekurzívní definice : Dva seznamy jsou ekvivalentní, když jsou oba prázdné a nebo když se rovnají jejich první prvky a současně jejich zbývající části seznamu.



Nechť jsou dány ukazatele UK1 a UK2 na dva seznamy prvků téhož typu TUk. Napište booleovskou funkci pro ekvivalenci těchto dvou seznamů.

```

function EquLists(Uk1,Uk2:TUk):boolean;
begin
  if Uk1 = Uk2
  then begin (*oba nilové, nebo dvakrát stejný*)
    EquLists:=true
  end else begin
    if (Uk1 <> nil) and (Uk2 <> nil)
    then begin (*oba nenilové - aplikace rekurzívní definice*)
      EquLists:=(Uk1^.Data = Uk2^.Data) and
                  EquLists(Uk1^.NextPtr, Uk2^.NextPtr)
    end else begin
      EquLists:=false (* jeden nil a druhý ne! *)
    end (*if(Uk1<>nil..*)
  end (*if Uk1 = Uk2*)
end

```



Pozn. S repertoárem instrukcí nad ADT seznam by se takový program psal obtížně, protože nemáme operaci pro "zbývající část seznamu". Možné řešení by nutilo zrušit první prvky seznamů, což by destruovalo porovnávané seznamy. Předpokládejte existenci operace **Tail(L)**, která reprezentuje část seznamu L bez prvního prvku a přepište uvedený program s využitím výhradně operací nad ADT List.

Hlavička

Hlavička (headings) seznamu je nástroj ke zjednodušení zápisu některých algoritmů nad seznamy.

- Hlavička je první prvek seznamu. Má pomocnou funkci a není skutečným prvkem seznamu.
- Prázdný seznam obsahuje jeden prvek - hlavičku.
- Seznam lze dočasně opatřit hlavičkou, která se v závěru algoritmu odstraní.
- Použití hlavičky odstraňuje zbytečný prolog pro první prvek, před cyklem opakovaných operací nad ostatními prvky seznamu.

Příklad bez hlavičky Příklad: Vytvoření kopie seznamu znaků v algoritmu bez hlavičky. Algoritmus musí před cyklem ošetřit vytvoření prvního prvku kopie a pak spustit cyklus kopírování.

```
procedure CopyList( LOrig:TList; var LDupl:TList);
(* Procedura s využitím ADT TList nad prvky integer a jeho operací - bez hlavičky*)
var
  El:char;
begin
  ListInit (LDupl);
  First (LOrig);
  if  Active (LOrig)
  then begin      (* vytvoření prvního prvku se provede před cyklem *)
    Copy (LOrig, El);
    InsertFirst (LDupl, El);
    succ (LOrig);
    First (LDupl);
    (* vytvoření zbytku seznamu se provede v cyklu *)
    while Active (LOrig) do begin
      Copy (LOrig, El);
      PostInsert (LDupl, El);
      Succ (LOrig);
      Succ (LDupl);
    end (* while *)
  end (* if *)
end; (* procedure *)
```

Příklad s hlavičkou Příklad: Vytvoření kopie seznamu celých čísel v algoritmu s hlavičkou. Algoritmus musí před cyklem vytvořit hlavičku pak spustit cyklus kopírování všech prvků.

```
procedure CopyList( LOrig:TList; var LDupl:TList);
(* Procedura s využitím ADT TList s hlavičkou nad prvky integer a jeho operací *)
var
  El:char;
begin
  ListInit (LDupl); (* prázdná kopie *)
  InsertFirst (LDupl, 0); (* vložení hlavičky s hodnotou 0 *)
  First (LOrig); (* první originálu nastav na aktivní *)
  First (LDupl); (* nastav hlavičku na aktivní *)
  (* vytvoření celého seznamu se provede v cyklu *)
```

```

while Active(LOrig) do begin
  Copy(LOrig,El);
  PostInsert(LDupl,El);
  Succ(LOrig);
  Succ(LDupl);
end (* while *)
DeleteFirst(LDupl); (* odstranění hlavičky *)
end; (* procedure *)

```

Pozn. Povšimněme si, že operace InsertLast, která v našem souboru není nad jednosměrným seznamem definovaná, by vedla k významnému zjednodušení algoritmu vytvoření duplikátu.



```

procedure CopyList(LOrig:TList; var LDupl:TList);
var
  El:char;
begin
  InitList(LDupl);
  First(LOrig);
  while Active(LOrig) do begin
    Copy(LOrig, El);
    Succ(LOrig);
    InsertLast(LDupl,El)
  end;
end;

```



Pozn: Všimněme si, že bez použití aktivity v duplikátním seznamu lze následujícím cyklem vytvořit kopii, v níž jsou prvky v obráceném pořadí:

```

while Active(LOrig) do begin
  Copy(LOrig,El);
  InsertFirst(LDupl,El);
  Succ(LOrig)
end;

```

Dvojsměrný seznam

Dvojsměrný seznam Dvojsměrný seznam je pro řadu algoritmů jednodušším nástrojem. Druhý ukazatel v prvku seznamu není při současných paměťových možnostech žádnou zbytečností. Lze říci, že jednosměrný seznam použijeme jen tehdy, je-li dvojsměrnost nadbytečná.

Porovnejte následující algoritmus s předcházejícími příklady:

```

procedure CopyDoubleList (DLOrig:TDList; var DLDupl:TDList);
var
  El:TData;
begin
  DInit(DLDupl);
  DFFirst(DLOrig);
  while DActive(DLOrig) do begin

```

```

DCopy(DLOrig,El);
DSucc(DLOrig);
DInsertLast(DLDupl,El) (* Vkládáním na konec je kopírování snadné*)
end (* while *)
end; (* procedure *)

```

Pozn. Podobně snadné je rušení nalezeného prvku nebo rušení prvku před nebo za nalezeným prvkem.

Implementace operací nad dvojsměrným seznamem

typy Pro implementaci ADT dvojsměrný seznam zavedeme typy

```

type
  TUkPrv=^TPrv;
  TPrv= record
    Data:TData;
    LUk, PUk:TUkPrv
  end;
  TDList=record
    Frst, Lst, Act :TUkPrv;      (* ukazatele na začátek a
                                    konec seznamu, a na aktivní prvek seznamu *)
    (* PocPrv: integer;   Možno použít pro počítadlo prvků seznamu*)
  end;

```

DListInit **procedure DListInit(var L:TDList);**

```

begin
  L.Frst:=nil;
  L.Lst:=nil;
  L.Act:=nil;
  (* PocPrv:=0; *)
end;

```

DInsertFirst **procedure DInsertFirst (var L:TDlist; El:TData);**

```

var
  PomUk:TUkPrv;
begin
  new(PomUk);
  PomUk^.Data:=El;
  PomUk^.LUk:= nil;    (* levý nového ukazuje na nil *)
  PomUk^.PUk:= L.Frst;  (* Pravý nového ukazuje
                           na aktuálně prvního nebo nil *)
  if L.Frst<> nil
  then begin
    L.Frst^.LUk:=PomUk;    (* Aktuální první bude
                            ukazovat doleva na nový prvek*)
  end else begin          (* Vkládám do prázdného seznamu *)
    L.Lst:=PomUk;
  end;
  L.Frst:=PomUk;           (* Korekce ukazatele začátku *)

```

```

end;

DDDeleteFirst (var L:TDlist);
(* Nutno sledovat zda operace neruší aktivní prvek resp. jediný prvek *)
var
    PomUk:TUKPrv;
begin
    if L.Frst<>nil
    then begin
        PomUk:= L.Frst;
        if L.Act=L.Frst
        then L.Act:= nil; (* první byl aktivní, ruší se aktivita seznamu *)

        if L.Frst=L.Lst
        then begin (* ruší se jediný prvek sezn., vznikne prázdný*)
            L.Lst:=nil;
            L.Frst := nil
        end else begin
            L.Frst:= L.Frst^.PUk; (* Aktualizace začátku sezn.*)
            L.Frst^.LUk:= nil; (* Není-li seznam prázdný, první prvek ukazuje
doleva uk. nil *)
        end; (* L.Frst= *)
        Dispose(PomUk)
    end; (* if L.Frst <> *)
end; (* procedure *)

DPostInsert (var L:TDList; El:TData);
(* Procedura musí dávat pozor, zda nevkládá za poslední prvek *)
var
    PomUk:TUKPrv;
begin
    if L.Act<>nil
    then begin
        new(PomUk);
        PomUk^.Data:= El;
        PomUk^.PUk:= L.Act^.PUk; (* *)
        PomUk^.LUk:= L.Act;
        L.Act^.Puk:= PomUk; (*Navázání lev. souseda na vložený prv.*)
        if L.Act=L.Lst
        then L.Lst:=PomUk (* Korekce ukaz. na konec – nový poslední *)
        else PomUk^.PUk^.LUk:=PomUk (* navázání pravého
sousedu vložený prvek *)
    end; (* if *)
end; (* procedure *)

DPostDelete (var L:DList);
(* Nutno sledovat, zda neruší poslední prvek *)
var
    PomUk: TUKPrv;

```

```

begin
  if (L.Act <> nil)
  then begin
    if L.Act^.PUk <> nil
(* Je co rušit? *)
    then begin (* Rušený existuje *)
      PomUk:= L.Act^.PUk; (* ukazatel na rušený*)
      L.Act^.PUk:= PomUk^.PUk; (* překlenutí rušeného *)
      if PomUk = L.Lst
      then L.Lst:=L.Act (* je-li rušený poslední, Act bude Lst *)
      else (* ruší se běžný prvek *)
        PomUk^.PUk^.LUk:= L.Act; (* prvek za zrušeným
                                   ukazuje doleva na Act *)
        Dispose(PomUk);
    end; (*if L.Act^.Puk <> nil *)
  end; (* if L.Act<> nil *)
end; (* procedure *)

```

DPre Delete Pozn. Všimněte si stranové symetrie s PostDelete

```

procedure DPreDelete(var L:DList);
(* Nutno sledovat, zda neruší první prvek *)
var
  PomUk: TUKPrv;
begin
  if (L.Act <> nil)
  then begin
    if L.Act^.LUk <> nil (* Je co rušit? *)
    then begin (* Rušený existuje *)
      PomUk:= L.Act^.LUk; (* ukazatel na rušený*)
      L.Act^.LUk:= PomUk^.LUk;(* překlenutí rušeného *)
      if PomUk = L.Frst
      then L.Frst:=L.Act (* Je-li rušen první, stane se aktivní prvním *)
      else (* ruší se běžný prvek *)
        PomUk^.LUk^.PUk:= L.Act; (* prvek před zrušeným ukazuje
                                   doprava na aktivní prvek *)
        Dispose(PomUk)
    end (*if L.Act^.Luk <> nil *)
  end (* if L.Act<> nil *)
end; (* procedure *)

```

Kruhový seznam

Kruhový seznam Kruhový (cyklický) seznam lze vytvořit z lineárního seznamu tak, že se ustaví pravidlo, že následníkem posledního prvku je prvek první. V implementační doméně to znamená, že ukazatel posledního prvku ukazuje na první prvek.

Ze sémantického pohledu nemá kruhový seznam začátek ani konec. Musí mít ale přístup alespoň k jednomu prvku seznamu, který má pozici pracovního počátku.

Kruhový seznam může být jedno nebo dvojsměrně vázaný (jednosměrný nebo dvojsměrný). Soubor operací pro ADT kruhový seznam lze odvodit ze souboru operací nad ADT seznam.

Sémantiku operace obdobné First lze definovat jako ustavení aktivity na "první" prvek - prvek, jehož prostřednictvím je umožněn přístup ke kruhovému seznamu.

Soubor operací je nutno doplnit o možnost testu na průchod celým seznamem. Jednou z možností je zavedení počítadla prvků, které umožní, aby se kruhovým seznamem pohybovalo s využitím počítaného cyklu. Jinou možností je zavedení predikátu který vrátí hodnotu true, když se v seznamu posuneme na poslední (znova první) prvek.

 Nad seznamy se v programátorské praxi používá řada různých souborů operací. Uvedený soubor, založený na aktivitě seznamu, ilustruje jen jednu z možností. V praxi si každý programátor vytvoří takový soubor operací, jaký je nejvhodnější pro jeho potřeby, nebo doplní známý soubor o operace, které mu vyhovují.

Typickým rozšířením definice typu i souboru operací je zavedení složky pro délku seznamu, která se při inicializaci nuluje, při každém úspěšném vložení inkrementuje a při každém úspěšném rušení dekrementuje. Jiným možným rozšířením je zavedení operace InsertLast pro jednosměrný seznam apod.

Problém k zamýšlení:

Navrhněte vhodný soubor operací nad ADT kruhový seznam takový, kterým lze provádět všechny nejznámější operace nad kruhovým seznamem jako:

- vytvoření kruhového seznamu z lineárního seznamu
- průchod kruhovým seznamem
- vytvoření kopie kruhového seznamu
- zrušení kruhového seznamu
- ekvivalence dvou kruhových seznamů

(Pozor! Poloha pracovního „prvního“ prvku není pro ekvivalenci významná!

Vyřešte uvedené úkoly s využitím abstraktních operací nad ADT kruhový jedno a/nebo dvojsměrně vázaný seznam i s využitím pascalovských ukazatelů

Kontrolní otázky:

- Které operace nad seznamem mohou způsobit chybový stav a proč?
- Vysvětlete rekurzivní definici ekvivalence dvou seznamů
- Podobně vytvořte rekurzivní definici délky seznamu
- Ukažte, že lexikografickou relaci nad dvěma seznamy lze řešit rekurzivně podobně, jako ekvivalenci dvou seznamů.
- Jak lze jednoduchým způsobem vytvořit invertovanou kopii seznamu (seznam s obráceným pořadím prvků)?
- Doplňte ve funkci pro zjištění délky seznamu na místa označená XXXXX a YYYYY správný zápis z následující nabídky.
function Length (L:TL) :integer;

```

var Count:integer;
begin
    Count:=0;
    First(L);
    while Active(L) do begin
        XXXXXXXXXXXX;
        YYYYYYYYYYY
    end; (* while *)
    Length:= Count
end

First(L)
pred(L)
succ(L,Count)
copy(L)
succ(L,Count+1)
Actualize(L)
succ(L)
InsertFirst(L,Count)
PostInsert(L,Count)
PostInsert(L, Count+1);
InsertFirst(L, Count+1)
Count:=Count+1
Count:=L+1
Length:= Count+1
Length:=Length+1
Length:=Length+Count
DeleteFirst(L,Count)
PostDelete(L,Count)
succ(L):=Length+1
succ(L):= Count+1

```



Příklady souboru operací, nad jednosměrnými, dvojsměrnými a kruhovými seznamy, jejich hlavičky a implementace naleznete na webové stránce předmětu:

<https://www.fit.vutbr.cz/study/courses/IAL/private/>

3.2.2 Zásobník

3.2.2 Zásobník (stack)



Zásobník je homogenní, lineární, dynamická struktura.

Zásobník lze odvodit z lineárního seznamu, pokud přístup k prvkům omezíme na jednu stranu (na začátek).

LIFO

Zásobníku se také říká struktura typu **LIFO** z anglického "Last-In-First-Out".

Zásobník má široké aplikační použití. Mezi nejčastější patří: inverze pořadí lineárního uspořádání, mechanismus přidělování paměti (bloková struktura), konstrukce rekurzivního

zápisu algoritmů, zpracování výrazů, algoritmy s návratem (*back-tracking*) předávání parametrů aj. Zásobník patří mezi nejvýznamnější ADT.

Mezi základní operace nad zásobníkem patří:

SInit
Push

Pop
Top



SInit(S) - Inicializace. Vytvoření prázdného zásobníku. Operace typu generátor.

Push(S,El) - Vložení prvku na vrchol zásobníku

Pop(S) - Zrušení prvku na vrcholu zásobníku. Zrušený prvek se ztrácí.

Top(S,El) - Čtení hodnoty prvku z vrcholu zásobníku do výstupního parametru El. . Obsah zásobníku beze změny. V případě prázdnosti zásobníku způsobí chybu nebo předčasné ukončení programu.

Pozn. Každý výskyt operace Top(S,El) musí být předcházen kontrolou neprázdnosti seznamu, nejčastěji v podobě:

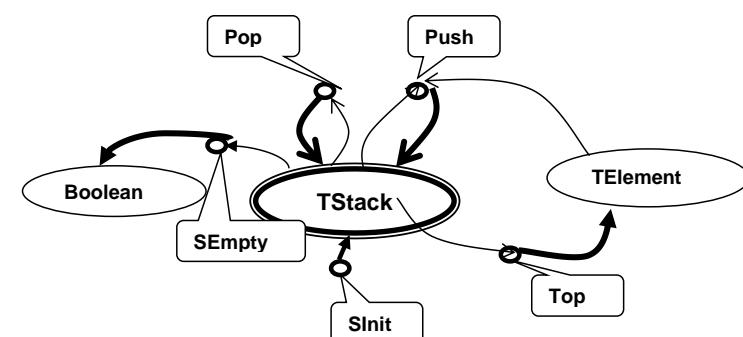
```
if not SEmpty(S)
then begin
    ...
    Top(S,El)
end ...
```

SEmpty

SEmpty(S) - Predikát prázdnosti seznamu. ve formě funkce. Vrací hodnotu "true" v případě prázdnosti zásobníku.

Pozn. V praxi se často setkáme s operací Pop, která kombinuje činnosti dvou operací: Top a Pop. Z hlediska teoretické čistoty skladby repertoáru operací je vhodné zachovat zásadu jedna operace - jedna činnost. Pro jiné případy lze v rámci tohoto předmětu použít operace TopPop(S,El), která i svým názvem jednoznačně označuje sémantiku operace.

**Diagram
sgnatury
Stack**



**Axiomat.
specifikace
Stack**

Axiomatická specifikace sémantiky ADT zásobník je srozumitelným příkladem takové specifikace. Popisuje exaktně a formálně chování ADT. Axiomatické specifikace pro složitější ADT mohou být rozsáhlé a obtížně prakticky použitelné.

1. $\text{Pop}(\text{Push}(S,El)) = S$
2. $\text{Top}(\text{StackInit}(S)) = \text{error}$
3. $\text{Top}(\text{Push}(S,El)) = El$
4. $\text{SEmpty}(\text{StackInit}(S)) = \text{true}$
5. $\text{SEmpty}(\text{Push}(S,El)) = \text{false}$

6. Pop(StackInit) = StackInit

Pozn. Axiomatické specifikace jsou uvedeny pro ilustraci. Nejsou určeny k reprodukci při zkoušce.

Pozn. Operace Top je ve specifikacích pojata jako funkce.

Vyčíslení výrazů

Použití zásobníku pro vyčíslení výrazů
Po zápis výrazů se používá **infixová, prefixová a postfixová** notace. Infixová notace uvádí (u dyadicke operace) operátor mezi dvěma operandy dvojice operandů a k zápisu větších výrazů potřebuje závorky pro potlačení vyšší priority operandů:

příkl: $3 * (5 + 7)$

Pozn. Prefixovová notace (tzv. "polská" notace byla poprvé formulována polským matematikem Janem Lukasiewiczem 1920). Prefixová notace připomíná zápis funkce v programovacím jazyku např. Add (A,B), kde operátor (jméno funkce je následováno operandy (parametry funkce).

Prefixová notace uvádí operátor dyadicke operace před dvojici operandů, nepotřebuje závorky a uvedený výraz by měl tvar:

příkl: $* 3 + 5 7$

Postfixová notace

Postfixová notace, nebo také obrácená polská notace (*Reversed Polish Notation - RPN*) uvádí operátor dyadicke operace za dvojici operandů.

příkl: $3 5 7 + *$

Postfixová notace má dvě významné vlastnosti:

- nepotřebuje závorky k potlačení vyšší priority operandů
- pořadí operátorů vyjadřuje pořadí operací, kterými se výraz vyčísluje

Příkl: $(a+b)*(c-d)/(e+f)*(g-h) \Rightarrow ab+cd-*ef+/gh-* = \Rightarrow ((ab+) (cd-) *) (ef+) / (gh-)* =$

Algoritmus pro vyčíslení výrazů v postfixové notaci s použitím zásobníku ve slovním vyjádření:

1. Zpracovávej řetězec zleva doprava
2. Je-li zpracovávaným prvkem operátor, vlož ho do zásobníku
3. Je-li zpracovávaným prvkem operátor, vyjmí ze zásobníku tolik operandů, kolika-nární je operátor (pro dyadicke operátoory dva operandy), proved danou operaci a výsledek uloží na vrchol zásobníku
4. Je-li zpracovávaným prvkem omezovač '=' , je výsledek na vrcholu zásobníku



Příklad k řešení:

Předpokládejte, že je definován ADT TStack čísel typu integer a deklarována globální proměnná S typu TStack, nad kterou smíte aplikovat operace tohoto typu.
Je dán řetězec znaků, obsahující číslice, operátory '+', '-', '*' a '/', které představují operace nad typem integer a ukončovací znak (omezovač) '=', kterým je řetězec zakončen. Číslice představují jednomístná celá čísla. Předpokládejte, že řetězec představuje syntakticky správný aritmetický výraz. Napište proceduru (funkci), která má na vstupu řetězec s postfixovým výrazem a která vrátí hodnotu vyčísleného výrazu.

Převod z

Jedním ze způsobů převodu infixové do postfixové notace je použití zásobníku.

infíkové do Algoritmus má slovní tvar:

**postfixové
notace**

1. Zpracovávej vstupní řetězec položku po položce zleva doprava a vytvářej postupně výstupní řetězec.
2. Je-li zpracovávanou položkou operand, přidej ho na konec vznikajícího výstupního řetězce.
3. Je-li zpracovávanou položkou levá závorka, vlož ji na vrchol zásobníku.
4. Je-li zpracovávanou položkou operátor, pak ho na vrchol zásobníku vlož v případě, že:
 - zásobník je prázdný,
 - na vrcholu zásobníku je levá závorka,
 - na vrcholu zásobníku je operátor s nižší prioritou.
5. Je-li na vrcholu zásobníku operátor s vyšší nebo shodnou prioritou, odstraň ho, vlož ho na konec výstupního řetězce a opakuj krok 4, až se ti podaří operátor vložit na vrchol.
6. Je-li zpracovávanou položkou pravá závorka, odebírej z vrcholu položky a dávej je na konec výstupního řetězce až narazíš na levou závorku. Tím je pář závorek zpracován.
7. Je-li zpracovávanou položkou omezovač ‘=’, pak postupně odstraňuj prvky z vrcholu zásobníku a přidávej je na konec řetězce, až se zásobník zcela vyprázdní. Na konec se přidá rovnítko, jako omezovač výrazu.

Implement. Zásobník lze implementovat v souvislé paměti po sobě jdoucích prvků (pole) nebo v paměti zřetězených položek (seznam). V prvním případě je datová struktura většinou “pseudodynamická” – tedy je dynamická potud, pokud nevyčerpá přidělený prostor (deklarované pole). Pro implementaci zásobníku polem je účelné dodefinovat predikát, která bude signalizovat naplnění vyhrazeného prostoru a tedy zákaz vkládání dalšího prvku do zásobníku. Nechť se tato operace jmenuje SFull(S) a v případě, že je zásobník plný, bude tato funkce vracet hodnotu true, jinak bude vracet hodnotu false.



Pozn. Pro implementaci polem je v jazycích jako je C nutno počítat s polem začínajícím o nulového indexu. Změní se některé inicializační hodnoty snížením o 1.

Nechť jsou definovány následující typy:

```
const SMax=1000; (* Maximální kapacita zásobníku *)
type
  TStack=record
    SPole=array[1..SMax] of TElem; (* pole pro zásobník *)
    STop: 0..SMax (* index ukazatele na vrchol *)
  end;
```

Pak operace inicializace bude mít podobu:

```
SInit
procedure SInit(var S:TStack);
begin
  S.STop:=0
end;

Push
procedure Push(var S:TStack; El:TElem);
begin
  with S do begin
```

```

    SStop:=STop + 1
    SPole[STop]:= El;
    end (* with *)
end;

Pozn: Operace Pop nevrací rušenou hodnotu.

procedure Pop (var S:TStack);
begin
  with S do begin
    if SStop > 0
    then SStop:=STop - 1
    end (* with *)
  end;

procedure Top (S:TStack; var El:TElem);
(* Před použitím musí být operace ošetřena na neprázdnost
zásobníku *)
begin
  El:= S.SPolice[S.STop]
end;

```

Pozn. Procedura ukončí program zásadní chybou, je-li při vyvolání zásobník prázdny, protože index Stop je nulový a tudíž mimo definovaný rozsah pole. Podmínu však do procedury nevkládáme. Podmínka ošetrující neprázdnost zásobníku je **povinnou součástí** použití Top. operace. Ponechme na programátorevi, co udělá v případě pokusu o čtení z prázdného zásobníku. Proceduru lze zapsat jako funkci v případě, že to dovoluje typ TElem.

```

function SEmpty(S:TStack):Boolean;
begin
  SEmpty:= S.STop=0
end;

```

```

function SFull(S:TStack):Boolean;
begin
  SFull:= S.STop=SMax
end;

```

Implementaci zásobníku jednorozměrným polem lze odvodit z operací nad jednosměrným seznamem: SInitList, InsertFirst, CopyFirst a DeleteFirst.

```

type
  TUk = ^TPrvek;

  TPrvek = record
    Data:TElem;
    UkDalsi:TUk
  end; (* TPrvek *)

```

```
  TStack=record (* typ zásobník *)
```

```

TopUk:TUk;
... (* Zde by mohly být uvedeny další atributy zásobníku, např. počet prvků *)
end; (* TStack*)

```

Operace nad zásobníkem:

```

procedure SInit(var S:TStack);
begin
  S.TopUk:=nil
  ...(* zde by byly inicializovány další atributy zásobníku, např. "pocet:=0" *)
end;

procedure Push(var S:TStack; El:TElem);
var
  PomUk:TUk;
begin
  new(PomUk);
  PomUk^.Data:=El;
  PomUk^.UkDalsi:=S.TopUk;
  S.TopUk:=PomUk
end;

procedure Pop(var S:TStack);
var
  PomUk:TUk;
begin
  if S.TopUk<>nil
  then begin
    PomUk:=S.TopUk;
    S.TopUk:=S.TopUk^.UkDalsi;
    dispose(PomUk)
  end (* if *)
end;

procedure Top(S:TStack; var El:TElem);
(* V případě prázdného zásobníku je ukazatel nilový a v pří jeho referenci dojde k chybě.
Podmínu prázdnosti však ponecháme vně procedury *)
begin
  El:=S.TopUk^.Data
end;

function SEmpty(S:TStack):Boolean;
begin
  SEmpty:=S.TopUk=nil
end;

```

 Korektní způsob práce se zásobníkem, jako s přesně definovaným abstraktním datovým typem, nepřipouští přístup k jiným prvkům struktury, než k vrcholu zásobníku.
Nedostatečný odhad kapacity zásobníku implementovaného polem vede nejčastěji k

systémové chybě a předčasnému ukončení programu.

Zásobník patří k nejdůležitějším strukturám ve výpočetní technice.

3.2.3

Fronta

3.2.3 Fronta (*queue*).

- Fronta je dynamická, homogenní a lineární struktura.
- Někdy se jí říká struktura typu "FIFO" : First-In-First-Out ("kdo dřív přijde, ten dřív mele").
- Na jednom konci fronty (obvykle ho zobrazujeme vpravo a říkáme mu "konec") se vkládají nové prvky, přidává a na druhém konci čte hodnota a prvek lze odebrat – "obsloužit" (obvykle ho zobrazujeme vlevo a říká se mu "začátek" - Pozn. zejména pro ty, kteří nevědí, jak se správně řadit do fronty).

Pozn. Použití ADT fronta souvisí s tzv. Teorie front (teorie hromadné obsluhy). K jejímu vzniku se váže historika s vodovodními kohoutky. Důstojník sledující za 2.světové války vojáky, kteří si myli esšálky pod vodovodními kohoutky tak, že umytí pod teplým trvalo 4x déle než opláchnutí pod studeným zjistil, že když přídá 3 kohoutky teplé vody, zečtyřnásobí rychlosť fronty, aniž musel čtyřnásobit všechny kohoutky.

Operace a jejich sémantika

Operace nad ADT fronta mají sémantiku podobnou operacím nad seznamem, ke kterému se přistupuje jen na obou koncích (na jednom se přidává, na druhém se odebírá):

QInit(Q) - inicializace fronty, vytvoření prázdné fronty (generátor)

QueUp(Q, El) - vložení prvku na konec fronty

Remove(Q) - rušení prvku na začátku fronty bez návratu hodnoty

Front(Q,El) - čtení hodnoty prvku na začátku bez změny fronty (definováno jen pro neprázdnou frontu)

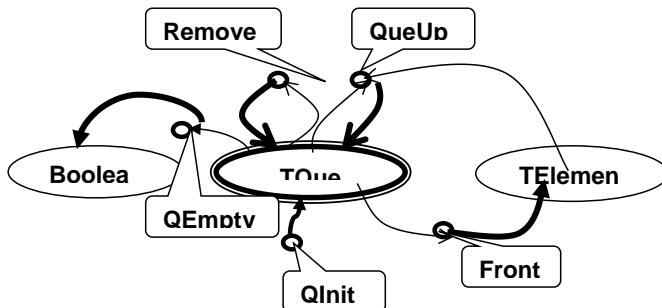
Pozn. Obdobně jako u operací Copy nad seznamem a Top u zásobníku, musí být operace Front(Q,E) ošetřena podmírkou na neprázdnost fronty.

QEmpty(Q) - predikát prázdnosti fronty.

Diagram signatury

Na následujícím obrázku je uveden diagram signatury ADT fronta.

Fronta



Axiom. specifikace

Axiomatická specifikace ADT fronta

1. $\text{QEmpty}(\text{QInit}(Q)) = \text{true}$
2. $\text{QEmpty}(\text{QueUp}(Q, E)) = \text{false}$
3. $\text{Front}(\text{QueInit}(Q)) = \text{error}$
4. $\text{Front}(\text{QueUp}(\text{QInit}(Q), E)) = E$
5. $\text{Front}(\text{Queup}(\text{Queup}(\text{QInit}(Q), E_A), E_B)) = \text{Front}(\text{QueUp}(\text{QInit}(Q), E_A))$
6. $\text{Remove}(\text{QInit}(Q)) = \text{QInit}(Q)$
7. $\text{Remove}(\text{QueUp}(\text{QInit}(Q), E)) = \text{QInit}(Q)$
8. $\text{Remove}(\text{QueUp}(\text{QueUp}(\text{QInit}(Q), E_B), E_A)) = \text{QueUp}(\text{Remove}(\text{QueUp}(\text{QInit}(Q), E_B)), E_A)$

Pozn. Axiomatické specifikace jsou uvedeny pro ilustraci. Nejsou určeny k reprodukci při zkoušce.

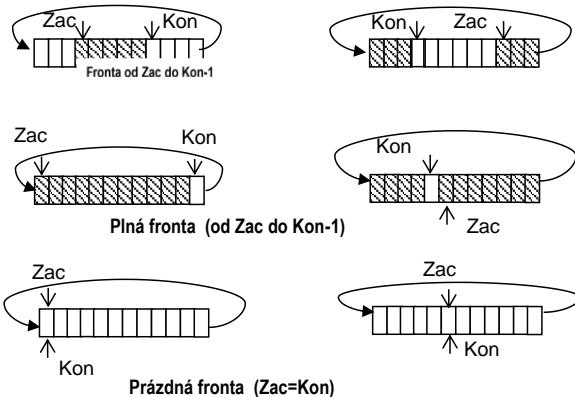
Fronta v poli

Fronta implementovaná polem má podobu kruhového seznamu v poli. Ukazatel na začátek "Zac" ukazuje na prvek, jehož hodnota je zpřístupňována operací `Front`. Ukazatel na konec "Kon" ukazuje na **první volný prvek** fronty. Na tuto pozici bude vložen nový prvek operací `QueUp`. Z toho vyplývá skutečnost, že efektivní kapacita takto implementované fronty je o 1 menší, než pole vyhrazené pro tuto implementaci.



Pro implementaci fronty polem je vhodné dodefinovat operaci `QFull`. Tento predikát vrátí hodnotu "true" v případě, že je fronta plná a nelze do ní vložit další prvek.

V následujícím vyobrazení jsou graficky znázorněny typické situace pro frontu částečně vyplňující pole, pro frontu zcela zaplňující pole (plnou frontu) a pro prázdnou frontu. Obkružující šipka znázorňuje kruhovost pole.

**Různé stavy fronty**

Typ TQue Pro frontu specifikujeme následující typy:

```
const = QMax; (* Fronta bude mít kapacitu QMax - 1 prvků !! *)
type
  TQueue = record
    QPole: array [1..QMax] of TEl;
    QZac,QKon: integer;
  end;
```

Pozn. Pro implementaci polem je v jazycích jako je C, nutno počítat s polem začínajícím o nulovém indexu.
Změní se některé hodnoty (inicializace, ošetření kruhovosti) snížením o 1.

QInit

```
procedure QInit(var Q:TQueue);
begin
  with Q do begin
    QZac:=1;
    QKon:=1
  end (* with *)
end;
```

QueUp

```
procedure QueUp (var Q:TQueue; El:TElem);
begin
  Q.QPole[Q.QKon] := El;
  Q.QKon:=Q.QKon + 1;
  if Q.QKon > QMax
  then Q.QKon := 1; (* Ošetření kruhovosti seznamu *)
end;
```

Remove

```
procedure Remove (var Q:TQueue);
begin
  if Q.QZac<>Q.QKon
```

```

    then begin
        Q.QZac:=Q.Qzac + 1;
        if Q.QZac > Q.QMax
            then Q.QZac:=1; (* Ošetření kruhovosti pole*)
        end (* if *)
    end;

```

Front **procedure Front (Q:TQueue; var El:TElem);**
(* Procedura způsobí chybu v případě čtení z prázdné fronty *)
begin
 El:= Q.QPole[Q.QZac]
end;

QEmpty **function QEmpty(Q:TQueue) : Boolean;**
begin
 QEmpty:= Q.Zac=Q.Kon
end;

QFull **function QFull(Q: TQueue) : Boolean;**
begin
 QFull:= (Q.Zac=1) and (Q.Kon=Q.QMax) or ((Q.Zac - 1) =
Q.Kon)
end;

Pozn. Ošetření kruhovosti pole lze zajistit také operací mod QMax. Druhým argumentem operace mod je vždy počet (Poc) prvků pole. Operace mod dává výsledky v intervalu 0..(Poc-1)



S ohledem na skutečnost, že pole začíná indexem 1, je nutné tuto jedničku přičíst. Pole má QMax prvků. Lze tedy inkrementaci ukazatele zajistit příkazem

Q.Zac:= Q.Zac mod QMax +1;

tzn. když je QMax = 100 má pole 100 prvků, pak
když staré Q.Zac = 99 pak po inkrementaci je:
 Q.Zac= 99 mod 100 +1 = 100
když staré Q.Zac = 100 pak po inkrementaci je:
 Q.Zac= 100 mod 100 +1 = 1

Když je pole definováno intervalom 0..QMax, má (QMax+1) prvků a inkrementaci s kruhovým převodem je možno zapsat :

Q.Zac:= (Q.Zac + 1) mod (QMax+1)

t.zn. když je QMax = 100 a pole má 101 prvků, pak
když staré Q.Zac = 99 pak po inkrementaci je

Q.Zac= (99 + 1) mod (100+1) = 100

když staré Q.Zac = 100 pak po inkrementaci je

Q.Zac= (100+1) mod (100 +1)= 0

Implementace fronty seznamem

type

```

TUk = ^TPrvek;
TPrvek = record
  Data: TElem;
  UkDalsi: TUk
end;

TQueue = record (* Typ fronty TQueue *)
  QZacUk, QKonUk:TUk;
end;

procedure QInit(var Q:TQueue);
begin
  Q.QZacUk:=nil;
  Q.QKonUk:=nil
end;

procedure QueUp(var Q:TQueue; El:TElem);
var
  PomUk:TUk;
begin
  new (PomUk);
  PomUk^.Data:= El;    (* naplnění nového prvku *)
  PomUk^.UkDalsi :=nil; (* ukončení nového prvku *)

  if Q.ZacUk = nil
  then (* fronta je prázdná, vlož nový jako první a jediný*)
    Q.QZacUk:=PomUk
  else (* fronta obsahuje nejméně jeden prvek, přidej na konec*)
    Q.QKonUk^.UkDalsi:=PomUk;
    Q.QKonUk:=PomUk (* korekce konce fronty *)
end;

procedure Front (Q:TQueue; var El: TElem);
begin
  El:=Q.QZacUk^.Data
end;

procedure Remove (var Q:TQueue);
var
  PomUk: TUk;
begin
  if Q.QZacUk <> nil
  then begin (* Fronta je neprázdná *)
    PomUk := Q.QZac;
    if Q.QZac = Q.QKon

```

```

then Q.QKon:=nil; (* Zrušil se poslední a jediný prvek fronty*)
Q.QZac:=Q.QZac^.UkDalsi;
dispose (PomUk);
end (* if Q.QZac <> *)
end;

```

DEQUE V řadě aplikací se setkáme s frontou, která umožňuje přidávat i odebírat prvky na obou koncích. Příkladem je hra "Domino". Takové struktury se říká oboustranně ukončená fronta (*double ended queue*) nebo také **DEQUE**.



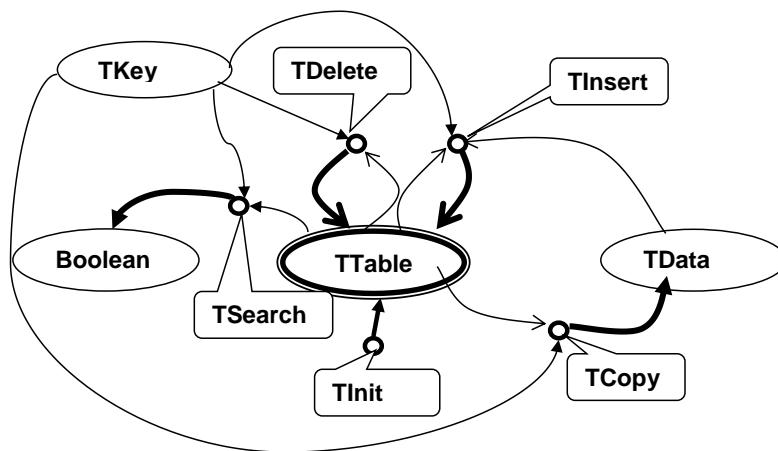
Kontrolní otázky a úkoly

- Jaký je rozdíl mezi infixovou a postfixovou notací? Uveděte přednosti postfixové notace.
- Popište postup vyčíslování výrazu v postfixové notaci.
- Popište postup převodu infixové notace na postfixovou.
- Nakreslete diagram signatury ADT zásobník
- Převeďte výraz: $(A+B) * (C-D) / ((D-F) *(E+G))^*$ H mod I = do postfixové notace.
- Nakreslete diagram signatury ADT fronta.
- Jak se zajišťuje kruhovost pole při implementaci fronty?
- Jaká je efektivní kapacita fronty implementované polem?

3.2.4 3.2.4 Vyhledávací tabulka

Vyhledávací tabulka Vyhledávací tabulka (*look-up table, search table*) - dále jen "tabulka" je homogenní, obecně dynamická struktura. Každá položka tabulky obsahuje složku, které se říká "klíč" (*key*). Klíč má v tabulce jednoznačnou hodnotu a slouží k identifikaci (vyhledávání) položky. Tabulka má vlastnosti "kartotéky". Mezi nejzákladnější operace patří operace vyhledávání "search".

Na následujícím obrázku je diagram signatury tabulky.



Sémantika operací:

TInit (T) - operace, která inicializuje (vytváří) prázdnou tabulku položek se složkami: klíč K typu TKlic a daty Data typu TData. (Složky TKlic a TData vytvárají typ položky "TElement")

TInsert(T,K,Data) - vložení položky se složkami K a Data do tabulky T. Pokud tabulka T již obsahuje položku s klíčem K dojde k přepisu datové složky Data novou hodnotou. Tato vlastnost se podobá činnosti v kartotéce, kdy při existenci staré karty se shodným klíčem se stará karta zahodí a vloží se nová (**aktualizační sémantika operace Tinsert**).

TSearch(T,K) - predikát, který vrací hodnotu true v případě, že v tabulce T existuje položka s klíčem K a hodnotu false v opačném případě.

TDelete(T,K) - operace rušení prvku s klíčem K v tabulce T. V případě, že takový prvek v tabulce neexistuje má operace účinek prázdné operace.

TCopy(T,K,El) - operace, která vrací (čte) ve výstupním parametru El hodnotu datové složky položky s klíčem K. V případě, že položka s klíčem K v tabulce T neexistuje, dochází k zásadní chybě. Proto je povinností programátora ošetřit každý výskyt operace TCopy predikátem Tsearch:

```

if TSearch(T, K)
  then begin
    TCopy(T, K, El);
  end;
  
```

Implementaci operací ADT tabulka bude vyhrazena samostatná kapitola předmětu IAL.

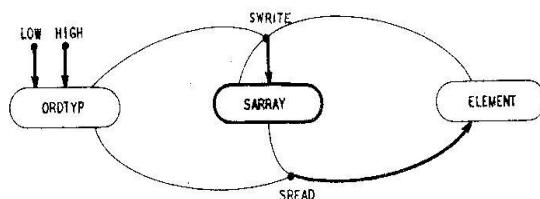
3.2.5 Pole 3.2.5 Pole (array)

je homogenní datová struktura ortogonálního (pravoúhlého) typu. V Pascalu je pole statické a je definováno jako jednorozměrné pole (vektor). N-rozměrného pole se dosahuje tím, že položkami jednorozměrného pole jsou (N-1) rozměrná pole.

Pozn. Skutečnost, že pole je v Pascalu jednorozměrné, není z deklarace ani z použití pole patrné na první pohled. Je-li matici vektor "řádků", pak lze s rádkem pracovat jako se strukturovanou hodnotou (např. v přířazovacím příkazu) a totéž nelze provádět se sloupcem.

Každá dimenze (rozměr) pole má rozsah indexů, které jsou ordinálního typu a jsou spočitatelné (vyčíslitelné). Index jednoznačně identifikuje prvek pole. Pascalovský způsob definice pole dovoluje ustavit dolní i horní hranici rozsahu indexu pro každou dimenzi.

SArray Diagram signatury ADT statické jednorozměrné pole SArray



Hlavními operacemi nad polem jsou:

- inicializace pole - vymezem intervalu indexu v dimenzi - inicializační operace operace Low(Dolni) a High(Horni),
- „zápis hodnoty do prvku pole, daného indexem“ SWrite(El) - „aktivní přístup“
- „čtení“ hodnoty prvku zadaného indexem SRead(El) – „pasivní přístup“

Pozn. Strukturovaná hodnota pole je definovaná, pokud jsou definovány hodnoty všech jejích komponent. Čtení hodnoty i-položky pole (přístup za účelem zjištění hodnoty), které nebyla dřívější operací SWrite(i,El) přiřazena hodnota, vede k chybnému stavu.

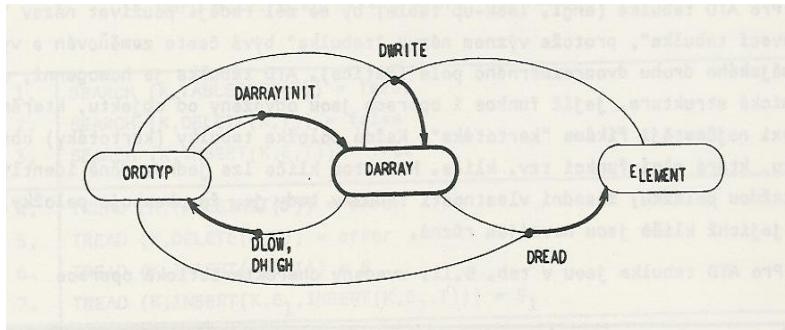
DArray Jednorozměrné dynamické pole DArray.

Dynamické pole se inicializuje v průběhu programu operací DArrayInit(Dolni,Horni), jejíž parametry vymezují rozsah indexu.

Pozn. Schéma operace DInitArray je zjednodušené. Chybí mu informace o paměťové náročnosti Elementu, zadávané explicitně, nebo odvozované od typu TElement, potřebné k alokaci paměťového prostoru..

Operace (funkce) DLow(Ind) a DHIGH(Ind) zjišťují aktuální stav nastavení rozsahu indexu.

Operace DWrite(i,El) resp. DRead(i,El) zajišťují aktivní resp. pasivní přístup k i-tému prvku pole. Pokus o čtení nedefinované hodnoty prvku pole vede k chybovému stavu.



Mapovací funkce

Mapovací funkce je transformační vztah, který převádí n-tici indexů prvku n-dimenzionálního pole na jeden index (adresu) jednorozměrného pole, jímž je n-dimenzionální pole reprezentováno v paměti.

Nechť B je k-dimenzionální pole, uložené v paměti po sloupcích.

B: array [low1 .. high1, low2 .. high2, ..., lowk .. highk] of TElement;
zápis prvku tohoto pole má tvar B[j₁, j₂, ..., j_k]

Pak mapovací funkce má tvar:

$$B[j_1, j_2, \dots, j_k] \rightarrow A\left[1 + \sum_{m=1}^{m=k} (j_m - low_m) * D_m\right]$$

kde A je jednodimenzionální pole reprezentující Pole B v paměti a jehož první index má hodnotu 1,

$$D_1 = 1$$

$$D_m = (\text{high}_{m-1} - \text{low}_{m-1} + 1) * D_{m-1}$$

Hodnota mapovací funkce se musí vyčíslovat v průběhu výpočtu při každé referenci (odkazu) na prvek pole. Jak je z tvaru výrazu vidět, může být vyčíslení, zejména u vícedimenzionálních polí, časově náročné. Době potřebné k výpočtu mapovací funkce se říká přístupová doba (*access time*).

Pozn. Znalost ani odvození tvaru mapovací funkce není předmětem zkoušení. Zkuste ale odvodit nejjednodušší případ mapovacích funkcí pro dvojrozměrné pole A[1..N,1..M] a trojrozměrné pole A[1..N,1..M,1..K].

Informační vektor

Z tvaru mapovací funkce je zřejmé, že členy low_m a D_m jsou nezávislé na momentálních hodnotách indexů indexované proměnné a jejich hodnotu lze určit v době překladu, kdy se zpracovávají deklarace. Vztah mapovací funkce lze upravit na tvar:

$$1 + \sum_{m=1}^{m=k} j_m D_m - \sum_{m=1}^{m=k} low_m D_m$$

Pak je při zpracování deklarace pole možné vytvořit pomocný informační vektor - informační záznam, (*dope vector*) do něhož se mj. uloží všechny hodnoty D_m a hodnoty $\Sigma(Low_m * D_m)$. Výpočet těchto hodnot se provede jen jednou a to v době inicializace pole při překladu. Vyčíslení mapovací funkce s použitím informačního vektoru rychlejší, za cenu dodatečné paměti potřebné pro uložení informačního vektoru. Protože většina jazyků umožňuje různé kontroly správnosti indexované proměnné (legální počet indexů, legální hodnota daného indexu), může informační vektor obsahovat i jiné pomocné údaje. Typický informační vektor obsahuje např.:

1. Počet dimenzí pole (k)
2. Dolní a horní hranice pro každou dimenzi
3. Celkový počet položek pole
4. D_m pro $m=1, 2, \dots, k$
5. $\Sigma (low_m * D_m)$
6. Adresa prvního prvku pole

Při předávání skutečného parametru procedury odkazem se u pole předává adresa informačního vektoru, který obsahuje všechny důležité informace o poli, včetně jeho adresy.

Prostorová a přístupová složitost pole.

Použití velkých a vícedimenzionálních polí provádí skutečnost, že velikost pole a doba přístupu k jednomu prvku pole mají protichůdné projevy náročnosti. Proto se setkáváme s řešeními v nichž se zdůrazňuje úspornost paměti na úkor doby přístupu nebo naopak se požaduje rychlý přístup k prvku pole i za cenu vyšší paměťové náročnosti.

Matice s nestejně dlouhými řádky (jagged-table, rag-table) je mechanismus, který šetří paměť v případech, kdy se na koncích řádků vyskytuje souvislá posloupnost nepoužívaných prvků. Takové matice se říká "matice s nestejně dlouhými řádky" a je typická pro zobrazení grafových a jiných podobných struktur.

V následujícím obrázku je uveden přístupový vektor **PV** a matice **A** s vyznačenými aktivními prvky a s prázdnými pozicemi. Výsledný vektor pro reprezentaci matice potřebuje 11 prvků. Mapování aktivního prvku pole A do výsledného vektoru **B** je dán mapovací vztahem:

$$A[i,j] = PV[i] + j$$

Hodnota přístupového vektoru na daném řádku odpovídá počtu aktivních prvků matice na všech předcházejících řádcích.

PV

Matice A

0	a ₁₁	a ₁₂							
2	a ₂₁	a ₂₂							
4			a ₃₁	a ₃₂	a ₃₃				
7			a ₄₁	a ₄₂	a ₄₃	a ₄₄	a ₄₅	a ₄₆	a ₄₇
16			a ₅₁						
17			a ₆₁						

Řídké pole

⚠ Řídké pole (*thin array*) je pole, v němž má významné množství prvků stejnou (dominantní) hodnotu. "Řídká" implementace pole snižuje paměťovou náročnost reprezentace za cenu zvýšení přístupové doby (*access time*) k prvku pole.

Pozn. Řídké pole je normální pole realizované úsporným způsobem. Příkladem je např. obrazovka bodů, na níž má většina bodů bílou barvu pozadí a jen body zobrazující text nebo čáry jsou jiné barvy. Počet bílých bodů výrazně převyšuje počet ostatních.

Řídké pole uchovává pouze hodnoty nedominantních prvků, spolu s jejich indexy. Přístup k prvku pole se realizuje vyhledáváním, kde klíčem je index (indexy) prvku. Pokud se prvek nenalezné, znamená to, že má dominantní hodnotu. Pole se tedy nejlépe implementuje vyhledávací tabulkou, v níž index je klíčem. Jednotlivé operace pak lze vyjádřit následujícími vztahy, v nichž Arr je pole určené k "řídké" implementaci.

- `InitArr(Arr)` → `TInit(T)`
- `ReadArr(Arr, Ind, El)` → `if Search(T, Ind)
then El:=TRead(T, Ind)
else El:= dominant value;`
- `WriteArr(Arr, Ind, El)` → `if El=dominant value
then TDelete(T, Ind)
else TInsert(T, Ind, El)`

Pro dvou(více)rozměrná pole lze použít mapovací funkce, která mapuje dvojici (n-tici) indexů do jednoho indexu

3.2.6 Graf**3.2.6 Graf**

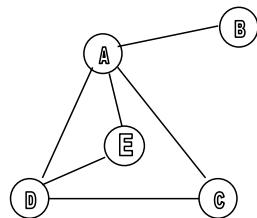
Graf je definován trojicí $G=(N,E,I)$

kde

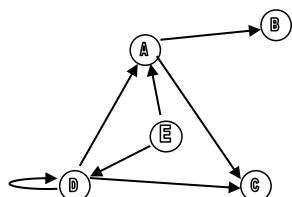
- N je množina uzlů, jimž lze přiřadit hodnotu
- E je množina hran, kterým lze přiřadit hodnotu. Každá hrana spojuje dva uzly a může být orientovaná. Je-li hrana orientovaná, pak jejímu grafu se říká orientovaný graf.

- I je množina spojení, která jednoznačně určuje spojení dvojic uzlů daného grafu.

Příklad neorientovaného grafu



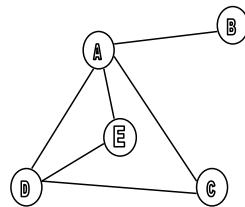
Příklad orientovaného grafu



DEF

- Průchodem** se nazývá posloupnost všech uzlů grafu.
- Průchod** je operace nad grafem, která provádí transformaci nelineární struktury na lineární.
- Cesta** z uzlu A do uzlu B je posloupnost hran, po nichž se dostaneme z uzlu A do uzlu B, aniž bychom šli po některé hraně dvakrát.
- Cyklický graf** je graf, který obsahuje „cyklus“. Cyklus je neprázdná cesta, která končí v téže uzlu, v němž začíná.
- Acyklický graf** neobsahuje cyklus.

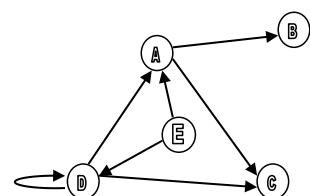
**Matice
koincidence**



Neorientovaný graf a jeho implementace maticí koincidence (spojení). Matice je symetrická podle hlavní diagonály.

	A	B	C	D	E
A	0	1	1	1	1
B	1	0	0	0	0
C	1	0	0	1	0
D	1	0	1	0	1
E	1	0	0	1	0

Matice koincidence pro orientovaný graf

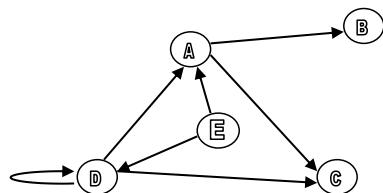


	A	B	C	D	E
A	0	1	1	-1	-1
B	-1	0	0	0	0
C	-1	0	0	-1	0

D	1	0	1	1	-1
E	1	0	0	1	0

Implementace orientovaného grafu maticí sousednosti

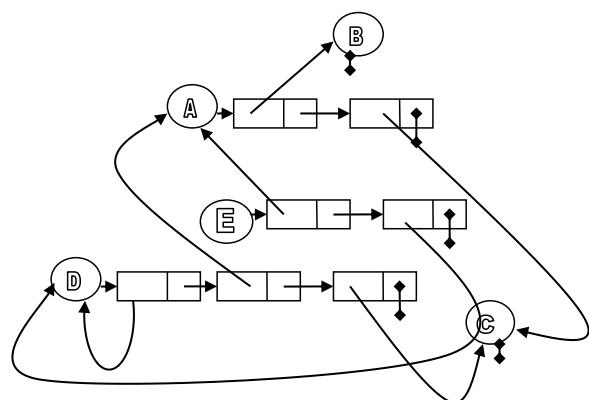
Matice sousednosti



A	B	C	-	-	-
B	-	-	-	-	-
C	-	-	-	-	-
D	A	C	D	-	-
E	A	D	-	-	-

Dynamická implementace orientovaného grafu seznamy hran

Dynamická implement. grafu



3.2.7 Strom 3.2.7 Strom (tree)

DEF

Kořenový strom (*root tree*) je acyklický graf, který má jeden zvláštní uzel (*node*), který se nazývá kořen (*root*).

- Kořen je uzel, pro nějž platí, že z každého uzlu stromu vede jen jedna cesta do kořene.
- Z každého uzlu vede jen jedna hrana (*edge*) směrem ke kořeni do uzlu, kterému se říká „otcovský“ uzel a libovolný počet hran k uzelům, kterým se říká „synovské“.
- Výška prázdného stromu je 0, výška stromu s jediným uzlem (kořenem) je 1. V jiném případě je výška stromu dána počtem hran od kořene k nejvzdálenějšímu uzlu + 1.

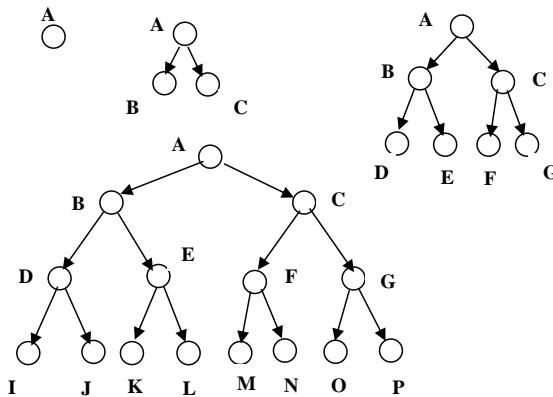
Binární strom**DEF**

Rekurzivní definice binárního stromu (*binary tree*):

Binární strom je buď prázdný, nebo sestává z jednoho uzlu zvaného kořen a dvou binárních podstromů – levého a pravého. (Oba podstromy mají vlastnosti binárního stromu.)

- Binární strom sestává z **kořene, neterminálních uzelů**, které mají ukazatel na jednoho nebo dva uzly synovské a **terminálních uzelů** (listů), které nemají žádné „potomky“.
- Každý uzel je kořenem svého „podstromu“. Podstromu (*subtree*) se říká také větev (*branch*). Uzly na cestě vycházející doleva z kořene vytvářejí "hlavní (levou) diagonálu", doprava "vedlejší (pravou) diagonálu".
- **Binární strom je váhou vyvážený (*weight balanced*), když pro všechny jeho uzly platí, že počty uzelů jejich levého podstromu a pravého podstromu se rovnají, nebo se liší právě o 1.**
- **Stromu, počet jehož uzelů je $2^n - 1$, pro $n \geq 0$ a jehož výška je právě n říkáme absolutně váhou vyvážený strom.**

Příklady absolutně vyvážených stromů:

**DEF**

- **Binární strom je výškově vyvážený (*height balanced*)**, když pro jeho všechny uzly platí, že výška levého podstromu se rovná výšce pravého podstromu, nebo se liší právě o 1.

Váhová vyvázenost

Binární strom (dále jen BS) je jednou z nejvýznamnějších datových struktur. Většina algoritmů nad BS může mít rekurzivní nebo nerekurzivní zápis. Existuje velký počet "cvičných" příkladů (stromových etud), jejichž zvládnutí patří k základům řemesla profesionálního programátora. Mnohé z nich jsou založeny na průchodu stromem

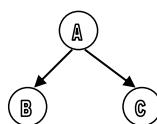
Výšková vyvázenost



Průchod stromem je posloupnost všech uzelů stromu, v níž se žádný uzel nevyskytuje dvakrát. Průchod transformuje nelineární strukturu stromu na lineární.

Mějme binární strom:

Průchody stromem



Pak průchod (pass) **PreOrder** má tvar A,B,C
 průchod **InOrder** má tvar B,A,C
 průchod **Postorder** má tvar B,C,A

Inverzní průchody mají obrácené pořadí synovských uzlů:

InvPreOrder má tvar A,C,B

InvInOrder má tvar C,A,B

InvPostOrder má tvar C,B,A

Průchod PostOrder je invertovaný (obrácený) průchod InvPreOrder

Algoritmy průchodů

Pro implementaci stromu budeme používat typy podobné typům používaným u dvojsměrného seznamu.

```
type
  TPtr=^TNode; (* ukazatel na typ TUzel *)
  TNode=record (* typ uzlu *)
    Data:Tdata;
    LPtr,RPtr:TPtr;
  end;
```

Algoritmus průchodu má rekurzivní podobu a hodnoty uzlů stromu při průchodu vkládá na konec dvojsměrného seznamu (zde označeného identifikátorem L, a jeho operace bez předpony "D").

```
procedure PreOrder(var L:TList;RootPtr:TPtr);
(* Seznam L byl inicializován před voláním *)
begin
  if RootPtr <> nil
  then begin
    DInsertLast(L,RootPtr^.Data);
    PreOrder(L,RootPtr^.LPtr);
    PreOrder(L,RootPtr^.RPtr)
  end;
end; (* procedure *)
```

Záměnou pořadí rekurzivního volání v podmíněném příkazu if získáme průchody InOrder a PostOrder

```
(* Inorder *)
  Inorder(L,RootPtr^.LPtr);
  DInsertlast(L,RootPtr^.Data);
  Inorder(L,RootPtr^.RPtr);

(* Postorder *)
  Postorder(L,RootPtr^.LPtr);
  Postorder(L,RootPtr^.RPtr);
  DInsertlast(L,RootPtr^.Data).
```

Stromové study Mezi typické stromové studie patří:

Stromové etudy - implementace některých operací nad BS :

- průchody BS
- ekvivalence struktur dvou BS
- ekvivalence dvou BS
- kopie BS
- destrukce BS
- počet listů BS,
- výška BS
- nalezení nejdelší cesty od kořene k listu
- váhová/výšková vyváženosť stromu).

**Nerekurz.
průchod
Preorder** Nerekurzívny průchod BS typu preorder lze zapsat řadou způsobů. Vždy však bude zapotřebí zásobník nad typem "ukazatel na uzel", který zajišťuje "návrat" k otcovskému uzlu. V uvedeném příkladu (a v řadě následujících ukázek) bude využita procedura, která svou činnost provádí při cestě levou diagonálou BS.

```
procedure NejlevPre(Uk:TUk; var L:TList);
(* Procedura Nejlev pro Preorder. Používá globální ADT zásobník ukazatelů "S" *)
begin
    while Uk<>nil do begin
        Push(S,Uk);
        InsertLast(L,Uk^.Data) (* vložení na konec seznamu*)
        Uk:= Uk^.LUk
    end
end;

procedure PreOrder(Uk:TUk; var L:TList);
begin
    ListInit(L); SInit(S);(* inicializace seznamu a zásobníku *)
    NejlevPre(Uk,L);
    while not SEmpty(S) do begin
        Top(S,Uk); Pop(S);
        NejlevPre(Uk^.PUk,L)
    end
end;
```

Upravte předcházející algoritmus tak, aby realizoval průchod "InOrder". Všimněte si, že půjde jen o přesun operace "InsertLast".

**Nerekurz.
průchod
PostOrder** Průchod PostOrder od předcházejících průchodů liší tím, že se k otcovskému uzlu vrací dvakrát. Poprvé zleva (aby šel doprava) a podruhé zprava, aby „zpracoval“ otcovský uzel. Zásobník booleovských hodnot provede rozlišení těchto "návratů". Procedura NejlevPost prochází levou diagonálou a naplňuje zásobník ukazatelů ukazateli, k nimž se bude vracet a zásobník pro příznak návratu hodnotou true, což bude znamenat návrat "zleva". Proto se průchodu PostOrder také říká, že je "dvouzá sobníkový".

Pozn. Pracujeme se dvěma různými typy zásobníků - se zásobníkem ukazatelů S a zásobníkem Booleovských hodnot SB. Proto se liší i jejich operace příponou "B".

```

procedure NejlevPost(Uk:TUk);
(* Globální ADT zásobník ukazatelů S a zásobník Booleovských hodnot SB *)
begin
    while Uk<>nil do begin
        Push(S,Uk);
        PushB(SB, true);
        Uk:=Uk^.Luk
    end
end;

procedure PostOrder(Uk:TUk; L:TList);
(* Vlastní průchod PostOrder se dvěma zásobníky *)
var
    Zleva:Boolean; (* indikátor návratu zleva *)
begin
    ListInit(L); SInit(S); SInitB(SB);
    NejlevPost(Uk);
    while not SEmpty(S) do begin
        Top(S,Uk);
        TopB(SB, Zleva); PopB(SB);
        if Zleva
            then begin (* přichází zleva, půjde doprava *)
                PushB(SB, false);
                NejlevPost(Uk^.Puk)
            end else begin (* přichází zprava, odstraní a zpracuje otcovský uzel *)
                Pop(S);
                InsertLast(L,Uk^.Data)
            end (* if *)
        end (* while *)
    end;

```

Rušení BS Jednoduchým způsobem jak zrušit všechny uzly BS je průchod PostOrder, v němž se při každém návratu zprava k otcovskému uzlu tento uzel zruší. Existují i jiné možnosti:

Algoritmus pro zrušení binárního stromu bez průchodu postorder.

```

procedure ZrusStrom(var Kor:TUkUz);
(* předpokládá se rušení neprázdného stromu, tedy kor<>nil *)
begin
    InitStack(S); (* inicializace zásobníku S pro ukazatele na uzel *)
    repeat (* cyklus rušení *)
        if Kor=nil
        then begin
            if not SEmpty(S)
            then begin
                Kor:=TOP(S);
                POP(S);

```

```

    end; (* if not SEmpty *)
end else begin (* jde doleva, likviduje a pravé strká do zásobníku *)
    if Kor^.PUK <> nil
        then PUSH(S,Kor^.PUK);
    PomPtr:=Kor; (* uchování dočasného kořene pro pozdějsí zrušení *)
    Kor:=Kor^.LUK; (* posud doleva *)
    dispose(PomPtr); (* zrušení starého uchovaného kořene *)
    end; (* Kor = nil *)
until (Kor=nil) and SEMPTY(S)
end

```

Obdobný algoritmus s využitím mechanismu Nejlev

```

procedure ZrusStrom2 (var kor:TUkUz);
(* tato verze s cyklem repeat předpokádá rušení neprázdného BS *)
var
    UkUz,UkUz2:TUkUz;

procedure nejlev (kor:TUkUz);
begin
    while kor<>nil do begin
        Push(S,kor);
        kor:=kor^.LUK;
    end
end;

begin;
    SInit(S);

    nejlev(UkUz); (* cestou doleva plní zásobník *)
repeat
    UkUz:=Top(S); Pop(S); (* uzel na vrcholu se ruší *)
    UkUz2:=UkUz;
    if UkUz^.Puk <> nil
        then nejlev(UkUz^.Puk); (* cestou doleva plní zásobník *)
        dispose(UkUz2);
    until SEmpty(S);
end;

```

Pozn. Příklady slouží k základní orientaci. V této podobě nejsou odladěny.

Duplikát BS s ukazateli - nerekurzívní zápis.

Binární strom je implementovaný pomocí ukazatelů. Napište nerekursívní zápis procedury pro vytvoření kopie (duplikátu) daného stromu. Zásobník nemusí implementovat.

type

```

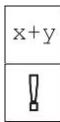
TUk  =^TUzel;  (* typ uzlu *)
TUzel=record
  Data:Tdata;
  LUk, PUK:TUk;
end;

procedure Nejlev (UkOr:TUk; var UkKo:TUk);
(* Pomocná procedura pro proceduru TreeCopy. Procedura pracuje se dvěma zásobníky
ukazatelů StackOr a StackKo prochází levou diagonálou originálu a souběžně vytváří
kopii *)
var PomUk:TUk; (* lokální pomocný ukazatel pro kopii *)
begin
  if UkOr<>nil then begin (* ošetření prvního uzlu diagonály před cyklem *)
    new(UkKo);           (* vytvoření prvního diagonály (kořene) *)
    UkKo^.Data:=UkOr^.Data;(* a jeho naplnění *)
    Push(StackOr, UkOr);   (* uložení do zásobníku pro pozdější návrat *)
    Push(StackKo, UkKo);
    PomUk:=UkKo;          (* umístění pomoc. uk. na "kořen" *)
    UkOr:=UkOr^.LUk;       (* posun v originálu doleva *)
    while UkOr<>nil do begin (* průchod druhým a dalšími prvky diagon. *)
      new(PomUk^.LUk);     (* vytvoření levého prvku kopie *)
      PomUk:=PomUk^.LUk;   (* posun v kopii doleva *)
      PomUk^.Data:=UkOr^.Data;(* naplnění uzlu kopie *)
      Push(StackOr, UkOr);   (* uk. uzlu originálu do zásobníku *)
      UkOr:=UkOr^.LUk;       (* posun v originálu doleva *)
      Push(StackKo, PomUk);   (* uk. uzlu kopie do zásobníku *)
    end; (* while *)
    PomUk^.LUk:=nil;        (* nilování nejlevějšího na diagonále kopie *)
  end else begin            (* originál prázdný, kopie taky... *)
    UkKo:=nil;
  end; (* if *)
end; (* procedure Nejlev *)

procedure TreeCopy (UkOr:TUk; var UkKo:TUk);
(* procedura vytvoří kopii - duplikát zadaného binárního stromu procedura pracuje se
dvěma zásobníky téhož typu: StackOr a StackKo *)
var PomUk:TUk; (* pomocný ukazatel pro kopii *)
begin
  SInit(StackOr);
  SInit(StackKo);
  Nejlev(UkOr, UkKo);

  while not SEmpty(StackKo) do begin
    Top(StackOr, UkOr); Pop(StackOr);
    Top(StackKo, PomUk); Pop(StackKo);
    Nejlev(UkOr^.PUk, PomUk^.PUk);
  end; (* while *)
end; (* procedure *)

```



Duplikát BS implementovaný v poli pomocí UDPP - nerekurzívní zápis.

Binární strom implementovaný (s využitím "uživatelské DPP") v poli DPP je dán indexem. Napište nerekusívní zápis procedury pro vytvoření kopie (duplikátu) daného stromu, který nebude obsahovat terminální uzly (listy) originálního stromu. Zásobník nemusíte implementovat.

```

type
  TUk=0..1000;      (* hodnota 0 je užívána ve významu nil *)
  TUzel= record      (* typ uzlu *)
    Data:Tdata;
    LUk, PUk:TUk;
  end;

  TDPP=array[1..1000] of TUzel;      (* pole pro UDPP *)

var  DPP:TDPP;

procedure Nejlev(UkOr:TUk; var UkKop:TUk);
(* procedura pracuje se dvěma zásobníky ukazatelů *)
var PomUkKop:TUk;
begin
  if (UkOr=0) or (DPP[UkOr].LUk=0) and (DPP[UkOr].PUk=0)
  then
    (* POZOR! funguje jen při zkráceném vyhodnocování booleovských výrazů. Jinak je
index UkOr mimo rozsah. Nad úpravou se zamyslete v rámci domácí přípravy. *)
    UkKop:=0          (* kořen resp. pravý syn nilový nebo list *)
  else begin
    newDPP(UkKop);  (* vytvoř kořen nebo pravého syna - první uzel diagonály *)
    DPP[UkKop].Data:=DPP[UkOr].Data;      (* kopírování dat      *)
    PushOr(UkOr);           (* uzel v originálu do zás. *)
    UkOr:=DPP[UkOr].LUk;       (* posun v originále doleva *)
    PomUkKop:=UkKop;

    while (DPP[UkOr].LUk<>0) or (DPP[UkOr].PUk<>0) do begin
      (* cyklus diagonály nezpracuje nejlevější uzel, je-li to list *)
      PushKop(PomUkKop);        (* uzel kopie do zásobníku  *)
      newDPP(DPP[PomUkKop].LUk); (* vytvoř neterminální řadový *)
                                  (* uzel diagonály kopie      *)
      UkKop:=DPP[PPomUkKop].LUk; (* posun v kopii doleva      *)
      DPP[PomUkKop].Data := DPP[UkOr].Data;  (* kopírování dat *)
      PushOr(UkOr);           (* uzel originálu do zásobníku *)
      UkOr:=DPP[UkOr].LUk;       (* posun v originálu doleva *)
    end (* while *)
    PushKop(PomUkKop);        (* uložení nejlevějšího uzlu diagonály kopie *)
    DPP[PomUkKop].LUk:=0;(* nilování nejlevějšího uzlu diagonály kopie *)
  end; (* if *)
end; (* procedure Nejlev *)

```

```

procedure RedukovanaKopie(KorOr:TUk; var KorKop:TUk);
(* procedura pracuje se dvěma zásobníky ukazatelů; každý je realizován samostatným
souborem operací; všimněme si, že má-li každý zásobník "své" operace, které se liší i
jménem (pak operace nemusí mít parametr, rozlišující o který zásobník v operaci jde;
takovému parametru se říká "parametr instance" *)
var
    UkOr,UkKop:TUk;
begin
    SInitOr; SInitKop;
    Nejlev(KorOr,KorKop);
    while not SEmptyOr do begin
        TopOr(UkOr); TopKop(UkKop);
        PopOr; PopKop;
        Nejlev(DPP[UkOr].PUk, DPP[UkKop].PUk);
    end; (* while *)
end; (* procedure *)

```

**Váhové
vyvážení z
pole**

x+y

Vytvoření váhově vyváženého binárního stromu ze seřazeného pole. (rekurzívni zápis)

```

procedure StromZPole(var Kor:TUk; LevyInd,PravyInd:integer;
Pole:TPole);
var Stred:integer;
begin
    if LevyInd <= PravyInd
    then begin
        Stred:=(LevyInd+PravyInd) div 2;
        Vytvor(Koren, Pole[Stred]);
        StromZPole(Koren^.Luk, LevyInd, Stred-1,Pole);
        StromZPole(Koren^.Puk, Stred+1, PravyInd,Pole);
    end else begin
        Koren:= nil
    end (* if *)
end

```

Výška BS

x+y

!

Procedura pro zjištění výšky BS - rekurzívni zápis

```

procedure VyskaBS(Kor:TUk; var Max:intger);
var Pom1, Pom2:integer;
begin
    if Kor <> nil
    then begin
        VyskaBS(Kor^.LUk, Pom1);
        VyskaBS(Kor^.PUk, Pom2);
        if Pom1> Pom2
        then Max := Pom1 + 1
        else Max := Pom2 + 1
        end else Max:= 0
    end; (* procedure *)

```

Výška BS - funkce

```

function Vyska (Uk:TUk) :integer;
function Max(N1,N2) :integer;
(* funkce vráti hodnotu většího ze dvou vstupních parametrů. Funkce Max je vnitřní
funkcí funkce Vyska *)
begin
    if N1 > N2
        then Max:= N1
        else Max:= N2
end;
begin
    if Uk=nil
        then Vyska:= 0
        else Vyska:= Max(Vyska(Uk^.Luk), Vyska(Uk^.Puk)) + 1
end; (* function*)

```

Ekvivalence struktur dvou stromů – rekurzívni zápis

```

function EQTS(Kor1, Kor2:Tuk) :Boolean;
begin
    if (Kor1=nil) or (Kor2=nil)(* alespoň jeden je nil *)
        then EQTS:= (Kor1=nil)and(Kor2=nil) (* oba jsou nil*)
        else EQTS := EQTS(Kor1^.LUk, Kor2^.LUk) and
                           EQTS(Kor1^.PUk, Kor2^.PUk)
            (* and (Kor1^.Data = Kor2^.Data) pro ekvivalenci BS*)
end;

```

Test váhové vyváženosti BS - rekurzívni zápis.

```

procedure TESTBVS (var Kor:TUk; var Balanced:Boolean;var
Pocet:integer);
var
    VYVL, VYVP:Boolean;
    PocL, PocP:integer;
begin
    if Kor <> nil
        then begin
            TESTBVS(Kor^.Luk, VYVL, PocL);
            TESTBVS(Kor^.Puk, VYVP, PocP);
            Pocet:=PocL + PocP + 1;
            Balanced:= VYVL and VYVP and (abs(PocL-PocP) <= 1);
        end else begin
            Pocet:=0;
            Balanced:= true
        end (*if*)
    end (* procedure *)

```

3.2.8 Jiné ADT

Jiné ADT

Mezi významné ADT patří také řetězec (*string*) a množina (*set*). V předmětu IAL se o nich zmíníme jen okrajově.

Řetězec

Řetězec je homogenní, lineární dynamická struktura, jejíž komponentou je typ znak (char). Řetězec má podobné vlastnosti jako pole znaků. Na rozdíl od pole však dovoluje relaci ekvivalence ("=","<>") a relaci uspořádání ("<","<=",">",">="). Nad typem **string** je v různých jazycích a knihovnách definováno velké množství různých procedur a funkcí. Mezi nejvýznamnější patří konatenace a dekatenace (zřetězení a rozdělení na podřetězce), vkládání a vyjímání podřetězců a vyhledávání podřetězců. Řetězec může být prázdný a pro práci s řetězem je typické použití konstruktoru, vytvářejícího hodnotu typu string výčtem jeho neoddělovaných komponent uzavřených do apostrofů nebo uvozovek např. 'ABBA'.

Implementace ADT řetězec má dvě typické podoby. Pascalovské pojednání implementovalo řetězec polem o 256 bytů [0..255], v němž byty 1..255 je určeno pro znaky a byte na indexu 0 obsahuje skutečný počet znaků (délku) řetězce.

Pojetí typické pro jazyky odvozené od C je pole bytů, obsahující znaky řetězce zakončené znakem s ordinální hodnotou "0".

Množina

Množina je homogenní, lineární, dynamická struktura nad ordinárním typem. Vyjadřuje "členství" hodnoty ordinálního typu. Typické operace nad množinou jsou převzaty z teorie množin. Množina může být implementovaná různými datovými strukturami. Nejjednodušší, ale neefektivní implementace množiny je polem prvků typu Boolean. Výhodnější je z hlediska paměťové náročnosti i rychlosti operací sekvence bitů. V obou případech (hodnota "true" prvku, nebo jedničková hodnota bitu) vyjadřuje "existenci" prvku, jehož ordinální hodnota je stejná jako pořadí prvku pole nebo bitu v sekvenci. Implementace množiny polem bitů umožňuje využít registrových logických instrukcí k urychlení některých množinových operací.

3.2.9

Kontrolní otázky



1. Nakreslete diagram signatury ADT zásobník.
2. Čím musí být ošetřena operace Top nad ADT zásobník?
3. Vysvětlete rozdíl mezi zápisem výrazu v infixové a postfixové notaci.
4. Napište algoritmy všech operací nad ADT zásobník, implementované polem i seznamem.
5. Popište přesně algoritmus převodu výrazu z infixové do postfixové notace.
6. Popište přesně mechanismus vyčíslení výrazu v postfixové notaci.
7. Nakreslete diagram signatury ADT fronta.
8. Napište algoritmy všech operací nad ADT fronta, implementované polem i seznamem.
9. Jak se zajišťuje kruhovost pole při implementaci pole? Popište oba možné způsoby.
10. Co to je oboustranně ukončená fronta a jaké má operace?
11. Nakreslete diagram signatury ADT vyhledávací tabulka.
12. Co je to "aktualizační sémantika" operaci Insert?
13. Co se stane, když se do tabulky vkládá prvek s hodnotou klíče, který je již v tabulce

obsažen?

14. Co se stane při operaci, která čte hodnotu prvku s klíčem, který v tabulce není obsažen?
15. Popište vlastnosti ADT pole.
16. Co je to mapovací funkce?
17. Co je to přístupová doba a na čem závisí?
18. Co je to matice s nestejně dlouhými řádky, k čemu se používá?
19. K čemu slouží přístupový vektor při práci s maticí s nestejně dlouhými řádky?
20. Napište transformační vztah pro přístup k prvku pole implementovaného maticí s nestejně dlouhými řádky.
21. Co je to řídké pole?
22. Co je to dominantní hodnota v řídkém poli?
23. Jak se implementují operace pro zápis a získání hodnoty z pole implementovaného jako řídké?
24. Jak je definován graf?
25. Jaký je rozdíl mezi orientovaným a neorientovaným grafem?
26. Jakými způsoby se v paměti reprezentuje graf?
27. Co je to acyklický graf?
28. Definujte rekurzivně binární strom.
29. Definujte váhovou vyváženosť BS.
30. Definujte výškovou vyváženosť BS.
31. Definujte pojem "výška stromu".
32. Vytvořte průchody typu PreOrder, InOrder a PostOrder pro stromy uvedené v příkladu absolutně vyvážených stromů.
33. K čemu slouží zásobník v nerekurzívním zápisu algoritmu průchodů InOrder a PreOrder?
34. K čemu slouží druhý zásobník v nerekurzívním zápisu algoritmu PostOrder?
35. Jaká je výška absolutně vyváženého binárního stromu, který má N uzlů?
36. Kolik uzlů má maximálně binární strom, jehož výška je M ?
37. Kolik zásobníků je zapotřebí k nerekurzívnímu zápisu průchodu PostOrder?

3.2.10

Kontrolní příklady



1. Vytvořte nerekurzívní proceduru, která vhodným parametrem určí, zda se zadaný průchod binárním stromem do zadанého seznamu uloží v podobě PreOrder, InOrder neb PostOrder.
2. Napište nerekurzívní proceduru PostOrder pomocí inverzního PreOrderu s „jedním“ zásobníkem.
3. Napište proceduru, která do jednosměrného seznamu uloží obrácené pořadí

průchodu BS typu PreOrder.

4. Vytvořte rekurzívni/nerekurzívni proceduru, která zruší všechny uzly zadaného BS.
5. Vytvořte rekurzívni/nerekurzívni proceduru, která vytvoří kopii (duplicát) zadaného zdrojového BS.
6. Vytvořte rekurzívni/nerekurzívni proceduru pro ekvivalenci struktur dvou stromů. (Procedura neporovnává hodnoty uzelů, jen to, zda oba mají stejnou konfiguraci synů).
7. Vytvořte rekurzívni/nerekurzívni proceduru pro ekvivalenci dvou binárních stromů. (Procedura na rozdíl od předchozí procedury porovnává i hodnoty uzelů)
8. Vytvořte rekurzívni/nerekurzívni proceduru/funkci, která zjistí výšku binárního stromu.
9. Vytvořte proceduru/funkci, která spočítá průměrnou vzdálenost a rozptyl vzdáleností listů od kořene BS.
10. Vytvořte proceduru, která naleze a do výstupního seznamu uloží nejdélší cestu od kořene k listu (od listu ke kořeni). Pozn. Můžete využít procedury z příkladu 7.
11. Je dáno seřazené pole prvků typu integer. Vytvořte Binární vyhledávací strom, který je váhově vyvážený

Závěr

Závěr

Třetí kapitola pokrývá stěžejní část látky předmětu - abstraktní datové typy, jejich význam, specifikaci a implementaci.

4 Vyhledávání

4 VYHLEDÁVÁNÍ

Cíl kapitoly

Cílem kapitoly je popis metod implementace ADT "vyhledávací tabulka". Bude zaměřena na vyhledávání ve vnitřní (operační) paměti s přímým přístupem a s využitím dynamických struktur a DPP.

V této kapitole uvedeme metody, které se liší způsobem přístupu k paměťovému prostoru, v němž je vyhledávací tabulka umístěna a to:

- sekvenční vyhledávání
- nesekvenční vyhledávání v poli
- vyhledávání v binárním vyhledávacím stromu (*binary search tree*)
- vyhledávání v tabulkách s rozptýlenými položkami (*hash-table*)

Nejvýznamnějším kriteriem pro hodnocení vyhledávacích tabulek je "přístupová doba" (*access time*). Je to doba potřebná k zajištění přístupu (k vyhledání) položky s hledaným klíčem. Pro hodnocení se používá několik časových vlastností vyhledávání. Zvlášť se označují doby pro úspěšné a pro neúspěšné vyhledání, které se u některých metod liší:

- minimální doba úspěšného a neúspěšného vyhledání
- maximální doba úspěšného a neúspěšného vyhledání
- průměrná doba úspěšného a neúspěšného vyhledání

Průměrná doba úspěšného vyhledání je teoretický parametr, který je dán podílem součtu dob úspěšného vyhledávání klíčů všech položek tabulky a počtu

položek.

Vlastnosti klíče Pro implementaci tabulky v konkrétním prostředí je nutné znát vlastnosti množiny hodnot klíče, podle kterého se bude vyhledávat. Je významné rozlišit případy, kdy nad typem klíč je definována pouze relace ekvivalence, nebo kdy je navíc definována i relace uspořádání.

Klíč může být jednoduchý nebo strukturovaný. Priority složek strukturovaného klíče určují váhu (význam) jednotlivých položek. Pro ekvivalence dvou strukturovaných klíčů musí být ekvivalentní všechny odpovídající si složky klíče. Pro relaci uspořádání dvou strukturovaných klíčů se porovnávají postupně odpovídající se složky klíče se srovnávají se prioritou (vahou).

x+y

Příklad: Položka "TOsoba" má jako klíč "datum narození", které sestává ze tří složek: Rok, Mesic a Den. Priorita složek pro relaci uspořádání je:

1. Rok, 2. Mesic, 3. Den.

Pro relaci uspořádání v seznamu osob uspořádaných podle pořadí narozenin v roce (s tím, že starší bude v den stejných narozenin uveden dříve) je:

1. Mesic, 2. Den, 3. Rok.

Schéma vyhledávání

Základní schéma vyhledávacího algoritmu má tvar:

```
Nasel:= false;
while not Nasel and <množina prvků není vyčerpána> do begin
    < prozkoumej další prvek. Je-li to hledaný, nastav Nasel na true >
end; (* while *)
Search:= Nasel;
```

Nevhodný zápis dvou podmínek v cyklu while může způsobit chybu při pokusu o vyhledání neexistujícího prvku. V případu práce s polem by zápis:

```
i:=1;
while (K<> Pole[i]) and (i <= max) do begin
    i:=i+1
end;
Search:= K= Pole[i]
```

vedl k referenci na neexistující prvek Pole[max+1].

Je-li vyhledávací tabulka v poli o max prvcích zcela zaplněná a přitom prvek s hledaným klíčem neexistuje dojde k pokusu o referenci neexistujícího prvku Pole[max+1]!

Vhodným řešením je použití booleovské proměnné:

```
Nasel:=false;
i:=1;
while not Nasel and (i<=max) do begin
    if K=Pole[i]
    then Nasel:=true
```

```

        else i:=i+1
end;
Search:=Nasel;
```

Lze také použít zápisu se "zkratovým vyhodnocením booleovského výrazu" (*short-cut evaluation of Boolean expression*). Pokud vzroste i nad horní hodnotu, booleovský výraz se vyhodnotí jako false dříve, než dojde k nekorektní referenci neexistujícího prvku Pole[i+1]:

```

i:=1;
(* zkratové vyhodnocení Booleovského výrazu *)
while (i <= max) and (K<> Pole[i]) do begin
    i:=i+1
end;
Search:= i <= max;
```

x+y

Pozn. V algoritmech IAL se "zkratově vyhodnocované booleovské výrazy" používají spíše výjimečně. V případě použití, jsou vždy uvedeny komentářem.

Podobně chybná situace může nastat v seznamové nebo souborové struktuře:

```

Uk:=UkZac;
while (K<>Uk^.Klic) and (Uk<>nil) do begin (* chybná
reference *)
    Uk:=Uk^.Puk
end;
```

Řešení zkratovým vyhodnocením
.....while (Uk<>nil) and (K<>Uk^.Klic) do ...

nebo raději:

```

Nasel:=false;
while not Nasel and Uk<>nil do begin
    if K=Uk^.Klic
    then Nasel:= true
    else Uk:= Uk^.Puk
end;
```

Dynamika tabulky

Vyhledávací tabulka se z pohledu dynamiky může chovat zcela staticky (seznam obcí v republice), staticky po etapách (papírový telefonní seznam aktualizovaný po roce), dynamicky s ohledem na operaci insert (změna jen přidáváním nových položek - tabulka identifikátorů v překladači), nebo plně dynamicky (dovoluje rušení položek). Některé implementace tabulek nedovolují nejvyšší stupeň dynamiky, nebo je jeho provedení těžkopádné a neefektivní. Rušení položky v každé tabulce je možné tzv. "**zaslepěním**".

Zaslepení

Zrušení položky **zaslepěním** se provede tak, že se klíč rušené hodnoty přepíše takovou hodnotou klíče, o níž je jisté, že nebude nikdy vyhledávána.

V rámci této kapitoly budou uvedeny následující metody. Některé z nich budou

uvedeny jen pro ilustraci zajímavých přístupů a nejsou předmětem zkoušení,

- Sekvenční vyhledávání v neseřazeném poli
- Sekvenční vyhledávání v neseřazeném poli se zarážkou
- Sekvenční vyhledávání v seřazeném poli
- Sekvenční vyhledávání v seřazeném poli se zarážkou
- Sekvenční vyhledávání v poli seřazeném podle pravděpodobnosti vyhledání klíče
- Sekvenční vyhledávání v poli s adaptivním uspořádáním četnosti vyhledání
- Binární vyhledávání v seřazeném poli
 - normální binární vyhledávání
 - Dijkstrova varianta binárního vyhledávání
- Uniformní binární vyhledávání
- Fibonacciho vyhledávání
- Binární vyhledávací stromy (BVS)
- AVL stromy
- Tabulky s rozptýlenými položkami (Hashing tables) - TRP

4.1 Sekvenční vyhledávání v poli

4.1 Sekvenční vyhledávání v poli bude používat následující datové typy:

```
const
  max=...; (* maximální kapacita tabulky *)
type   (* typ položky tabulky *)
  TPol=record
    Klic:TKlic;
    Data:TData
  end;
  TTab=record (* typ tabulka implementovaná polem*)
    Tab: array[1..max] of TPol; (* pole tabulky *)
    N: integer; (* aktuální počet prvků v tabulce*)
  end;
```

Pozn. Pro typ klíče budeme používat nejčastěji identifikátor TKlic, pro název klíčové složky položky tabulky identifikátor Klic a pro hodnotu vyhledávaného klíče identifikátor K.

- Inicializace tabulky – nulování aktuálního počtu prvku N
- Operace vyhledávání "Search" :

```
function Search(T:TTab; K:TKlic):Boolean;
(* funkce vrací hodnotu "true" v případě nalezení prvku s hledaným klíčem *)
var
  i:integer; Nasel:Boolean;
begin
  Nasel:=false;
  with T do begin
    i:=1;
    while not Nasel and (i<=N) do begin
      if K = Tab[i].Klic
```

x+y

```

    then Nasel:=true
    else i:=i +1
  end; (* while *)
  Search:=Nasel
end (* with *)
end; (* function*)

```

- Varianta operace Search v podobě procedury s parametrem vrácení polohy (indexu) nalezeného prvku:

```

procedure SearchIns(T:TTab; K:TKlic;
                     var Nasel:Boolean;
                     var Kde:integer);
var i:integer;
begin
  Nasel:=false;
  with T do begin
    i:=1;
    while not Nasel and (i<=N) do begin
      if K = Tab[i].Klic
      then Nasel:=true
      else i:=i +1
    end; (* while *)
    kde:=i; (* pro Nasel=false je i nedefinováno*)
  end (* with *)
end; (* procedure *)

```

- Operace Insert používá proceduru Search pro účely vrácení polohy nalezeného prvku:

```

procedure INSERT(var T:TTab; Pol:TPol; max:integer;
                 var OverFlow:Boolean);
var Nasel:Boolean;
    Kde:integer;

begin
  Overflow:=false; (* počáteční nastavení příznaku plné tabulky *)
  SearchIns(T,Pol.Klic,Nasel,Kde); (* vyhledání za účelem
  vkládání *)
  if Nasel
  then begin
    T.Tab[Kde]:= Pol; (* přepsání staré položky *)
  end else begin (* má se vložit nový prvek *)
    Kde:=T.N+1;
    if Kde <=max (* je v tabulce místo ? *)
    then begin (* lze vložit *)
      T.Tab[Kde]:=Pol; (* vkládání *)
      T.N:=Kde; (* aktualizace počítadla *)
    end else begin
  end
end

```

```

Overflow := true (* nelze vložit – přetečení *)
end (* if Kde <> max *)
end (* if Nasel *)
end; (* procedure *)

```

- Operace Delete se provede výměnou rušeného prvku s posledním a zrušením posledního:

```

procedure Delete(var T:TTab; K:TKlic);
var
  kde:integer;
  Nasel:Boolean;
begin
  with T do begin
    SearchIns(T, K, Nasel, Kde);
    if Nasel then begin
      Tab[Kde]:=Tab[N] (* rušený je přepsán posledním *)
      N:=N-1;
    end; (* if *)
  end; (* with *)
end; (* procedure *)

```

Pozn. Operaci Delete lze také implementovat zaslepením: Klíč rušené položky se přepíše hodnotou, která se nikdy nebude vyhledávat. Tento způsob ale snižuje efektivní kapacitu tabulky!

Hodnocení metody



Hodnocení metody sekvenčního vyhledávání

- minimální čas úspěšného vyhledání = 1
- maximální čas úspěšného vyhledání = N
- průměrný čas úspěšného vyhledání = N/2
- čas neúspěšného vyhledání = N
- nejrychleji jsou vyhledány položky, které jsou na počátku tabulky

Sekvenční vyhledávání lze snadno realizovat i v typických sekvenčních strukturách, jako je seznam nebo soubor. Operaci insert (s aktualizační sémantikou) přepíšeme v případě úspěšného nalezení a nový vložíme na kterékoli místo (začátek u seznamu, konec i souboru). Operace Delete přináší nutnost rušit v seznamu (snadnou o dvousměrného, méně snadnou u jednosměrného seznamu). U sekvenčního souboru je rušení prvku problematické. Řešení s pracovním souborem, do kterého se přepíše vše kromě rušeného a přepis zpět může mít nepřijatelnou časovou náročnost.

4.2 Rychlé sekvenční vyhledávání



4.2 Vyhledávání se zarážkou – tzv. **rychlé sekvenční vyhledávání** používá zarážku, což je přidaná položka na konec obsazené části tabulky, do níž se vloží hledaný klíč. Vyhledání tak vždy skončí nalezením, a úspěšnost se pozná podle indexu, na němž vyhledání skončilo. Rychlosť spočívá ve zkrácení booleovského výrazu.

Zarážka (*sentinel, guard, stop-point*) je často používaná technika, která umožní

vynechat test na konec pole (sekvence, průchodu). Nutností rezervovat místo pro zarážku snížuje efektivní kapacitu tabulky o 1 položku.

```
function SearchG(T:TTab; K:TKlic):Boolean;
var
  i:integer;
begin
  i:=1;
  T.Tab[T.N+1].Klic := K; (* vložení zarážky *)
  while K<>T.Tab[i].Klic do i:=i+1;
  Search:=i<>(T.N+1)(* když našel až zarážku, tak vlastně nenašel*)
end; (* function *)
```

4.3 Sekvenční vyhledávání v seřazeném poli

4.3 Sekvenční vyhledávání v seřazeném poli (*ordered array*)

- Podmínkou vyhledávání v seřazeném je definovaná relace uspořádání (dříve stačila relace rovnosti) nad typem klíč.
- Pole je seřazeno podle velikosti klíče.
- Operace Search skončí neúspěšně, jakmile narazí na položku s klíčem, který je větší, než je hledaný klíč!
- Operace Search se urychlí jen pro případ neúspěšného vyhledávání. To je jediný význam vyhledávání v seřazeném poli!! Jinak se věci jen komplikují...!
- Operace Insert musí najít správné místo, kam vloží nový prvek, aby zachovala seřazenost pole. Segment pole od nalezeného místa se musí posunout o jedničku. (N se zvýší o jedničku).
- Operace Delete posune segmentem pole napravo od vyřazovaného doleva o jednu pozici a tím se vyřazováný přepíše. (Počet prvků N se sníží o jedničku.)



Pozn. Pozor: Posun segmentu pole doprava se dělá cyklem zprava a naopak! Posun segmentu [Dolni..(Horni-1)] o jednu pozici doprava:

```
for i:= Horni downto (Dolni + 1) do Pole[i] := Pole[i-1]

function Search(T:TTab; K:TKlic):Boolean;
(* funkce vrací hodnotu "true" v případě nalezení prvku s hledaným klíčem *)
var
  i:integer; Konec:Boolean;
begin
  Konec:=false;
  with T do begin
    i:=1;
    while not Konec and (i<=N) do begin
      if K <= Tab[i].Klic
      then Konec:=true
      else i:=i +1
    end; (* while *)
    if i<= N
    then Search:=Tab[i]=K
    else Search:= false
```

```
end (* with *)
end; (* function*)
```



Upravte algoritmus sekvenčního vyhledávání v seřazeném poli tak, aby pracoval se zarážkou.

4.4 Vyhledávání s rekonfigurací pole

4.4 Sekvenční vyhledávání s adaptivní rekonfigurací pole podle četnosti přístupů.

Nejvhodnější by bylo uspořádání pole podle četnosti vyhledání tak, aby nejčastěji vyhledávané položky byly na počátku pole. To lze realizovat občasným seřazením položek podle počítadla, které se aktualizuje po každém přístupu k položce.

Vyhodnější variantou je vyhledávání s adaptivní rekonfigurací pole podle četnosti vyhledání. Po každém přístupu k položce se položka vymění se svým levým sousedem, pokud sama již není na první pozici:

Součástí vyhledávacího cyklu této metody je příkaz:

```
if Kde>1
then T.Tab[Kde] :=: T.Tab[Kde-1]
```

Tato metoda je velmi elegantní a účinná všude tam, kde se spokojíme se sekvenčním vyhledáváním a lineární složitostí.

4.5 Binární vyhledávání



4.5 Binární vyhledávání v seřazeném poli se provádí nad seřazenou množinou klíčů s náhodným přístupem (v poli). Metoda připomíná metodu plnění intervalu pro hledání jediného kořene funkce v daném intervalu. Hlavní vlastností binárního vyhledávání je jeho složitost, která je v nejhorším případě logaritmická $\lg_2(n)$.

K samostatné úvaze. Porovnejte zaokrouhlenou hodnotu pro nejhorší případ binárního vyhledávání v poli o 1000 prvcích pro nejhorší případ sekvenčního vyhledávání v tomto poli. (Jinými slovy, kolikrát se vám podaří rozpůlit interval o 1000 položkách, aby již nebylo co půlit?)

Pro pole tabulky implementované binárním vyhledáváním platí:

```
Tab[1].Klic < Tab[2].Klic < ... < Tab[N].Klic
a pro vyhledávaný klíč musí platit:
(K >= Tab[1].Klic) and (K <= Tab[N].Klic)
```

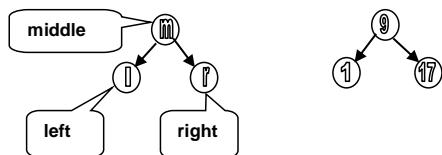
pak algoritmus vyhledávání tvoří sekvence příkazů:

```
...
left:=1; (* levý index *)
right:=n; (* pravý index *)
repeat
    middle:=(left+right) div 2;
    if K < Tab[middle].Klic
```

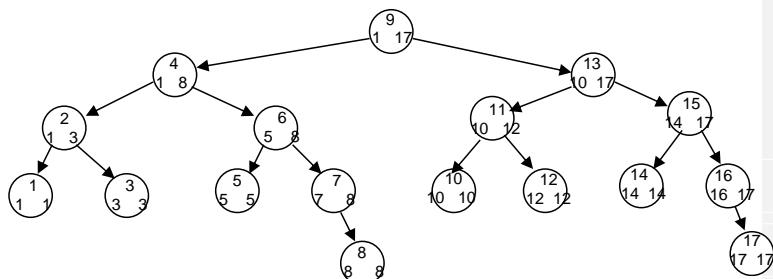
```

then right:=middle-1 (* hledaná položka je v levé polovině *)
else left:= middle+1; (* hledaná položka je v pravé polovině *)
until (K=Tab[middle].Klic) or (right < left);
Search:= K=Tab[middle].Klic;
Mechanismus výpočtu středu je middle=(left+right) div 2

```



Mechanismus postupného rozdělování intervalu a určování středu zobrazuje rozhodovací strom binárního vyhledávání.



4.6 Dijkstrova varianta binárního vyhledávání

- E.W.Dijkstra – významný teoretik programování druhé poloviny minulého století.
- Dijkstrova varianta binárního vyhledávání vychází z předpokladu, že v poli může být více položek se shodným klíčem. (To se neočekává u vyhledávací tabulky. Dijkstrova varianta se používá pro účely řazení.)

Polohu které z několika položek se shodným klíčem má vrátit mechanismus Search? Obvyklým požadavkem je některý z krajních (nejčastěji poslední ze shodných). Tomuto požadavku odpovídá algoritmus, který nekončí tím, že najde shodu s klíčem, ale tím, že se dalším dělením dostane až na dvojici sousedních prvků, o nichž platí:

$$\text{Tab}[i].\text{Klic} = K \text{ a současně } \text{Tab}[i].\text{Klic} < \text{Tab}[i+1].\text{Klic}$$

To zaručuje, že i-tá položka je nejpravější se shodných klíčů rovnajících se hledanému klíči K.

Pro tuto možnost (hledání nejpravějšího) pak platí:

$\text{Tab[1].Klic} \leq \text{Tab[2].Klic} \leq \dots \leq \text{Tab[N-1].Klic} < \text{Tab[N].Klic}$

(* všechny klíče pole mohou být shodné, kromě nejpravějšího *)

a také

$(\text{K} \geq \text{Tab[1].Klic}) \text{ and } (\text{K} < \text{Tab[N].Klic})$

(* hodnota klíče musí být uvnitř intervalu mezi levým a pravým okrajem*)

Pak Dijkstrova varianta pro hledání nejpravějšího ze shodných má podobu sekvence příkazů:

```
....  
left:=1;  
right:=n;  
while right <> (left+1) do begin  
    middle:=(left+right) div 2;  
    if Tab[middle].Klic <= K (* hledá nejpravější *)  
        then left:=middle  
        else right:=middle  
    end;  
Search:= K=Tab[left].Klic;
```

Příklad:

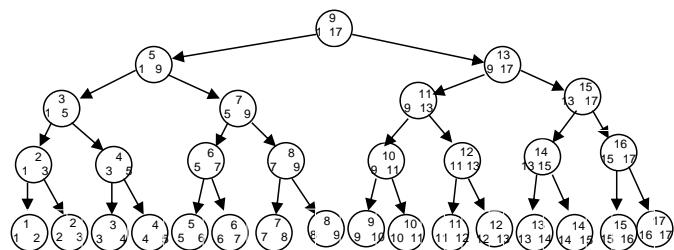
V poli: 1,2,3,4,5,5,6,6,6,8,9,13 najde algoritmus Dijkstrovy varianty klíč K=6 na 9. pozici:

V poli: 1,1,1,1,1,1,1,1,1,1,2 najde algoritmus Dijkstrovy varianty klíč K=1 na 10. pozici.

Vyhledáváním nejlevější a nejpravější položky mezi položkami se stejným klíčem je komplementární.

```
if Tab[middle].Klic => K (* hledá nejlevější *)  
then right:=middle  
else left:=middle  
a levým okrajovým prvkem, který nesmí být vyhledáván a musí být menší než ostatní pro verzi nejlevější.
```

Rozhodovací strom Dijkstry varianty pro pole [1..17] má tvar:



Σ Hodnocení binárního vyhledávání

- a) Vyhledávání má logaritmickou složitost
- b) Je zvlášť výhodné pro statické tabulky, kde není nutný potenciálně časově náročný posun segmentu pole
- c) Operace Insert a Delete mají stejný charakter jako u sekvenčního vyhledávání v seřazeném poli.
- d) U Dijkstry varianty má doba pro úspěšné i neúspěšné vyhledávání stejnou velikost $\lg_2 N$.

4.7 Uniformní a Fibonacciho vyhledávání

Uniformní a Fibonacciho vyhledávání nabízejí řešení pro případ, že operace půlení intervalu je operací časově náročnou. K takovému případu může dojít, pokud např. prostředí programového systému nepoužívá k zobrazení čísel dvoujmovkového kódu.

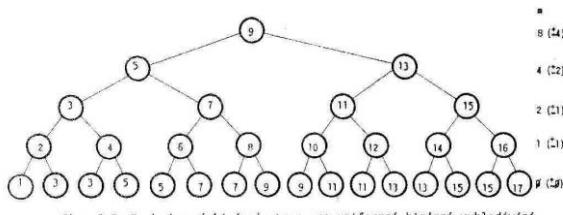
Pozn. Počítače specializované na masové použití vstup/výstupních operací číselných hodnot (orientované např. na hromadné zpracování dat při ekonomických výpočtech) mohou používat specializovanou aritmetiku, založenou na binárně kódované desítkové soustavě (*binary coded decimal*) - BCD aritmetice. V prostředí binární aritmetiky je půlení realizováno jedním posunem bitu doprava a je velmi rychlou operací. V prostředí aritmetiky BCD může být operace půlení mnohem pomalejší a v rozsáhlém problému může výrazně snížit efektivnost metody založené na půlení. Metody uniformního binárního a Fibonacciho vyhledávání se snaží tuto nevýhodu odstranit.

Uniformní binární vyhledávání je založeno na principu určení hranic intervalu odchylkou od středu. Pro danou tabulku se spočítá pole odchylek a kraje intervalu se určí z hodnoty odchylky levého a pravého okraje od středu. V následujícím rozhodovacím stromu je vidět, že na dané úrovni jsou odchylky krajů intervalu od středu vždy stejné. proto se metoda jmenuje "uniformní".

Na nejnižší úrovni jsou kraje intervalu 1 a 3 vzdáleny od středu 0 o hodnotu 1. (v rozhodovacím stromu se např. k uzlu 5 mohu dostat doleva od středu 6 nebo doprava od středu 4. Pro danou tabulku lze odchylky stanovit jednorázově dopředu a v průběhu vyhledávání je již opakováně nevyčíslovat. Tím se zabrání zdlouhavému půlení. Tabulky, které mají počet prvků o hodnotě $2^n - 1$, mají pole odchylek se společným základem. Pro takové tabulky lze připravit univerzální pole odchylek. Použití tabulky s libovolnou velikostí řeší **Sharova metoda**,

která bude dále uvedena.

Rozhodovací binární strom pro uniformní binární vyhledávání.



Obr. A.3. Rozhodovací binární strom pro uniformní binární vyhledávání

Fibonacciho strom

Fibonacciho posloupnost

DEF

Definice Fibonacciho stromu

DEF

Tak, jako je binární vyhledávání v poli úzce spjato s binárním rozhodovacím stromem, který znázorňuje postup dělení intervalu v poli, je Fibonacciho vyhledávání úzce spjato s tzv. **Fibonacciho stromem**.

Základem Fibonacciho stromu je **Fibonacciho posloupnost** 1. rádu, definovaná vztahy:

$$F_0=0, F_1=1, \text{ pro } i>1 \quad F_i=F_{i-1}+F_{i-2}$$

Následující prvek posloupnosti je dán součtem aktuálního prvku a jednoho jeho předchůdce. Pro počáteční hodnoty 0, a 1 má posloupnost tvar:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 atd.

Fibonacciho strom (F-tree) je definován pravidly:

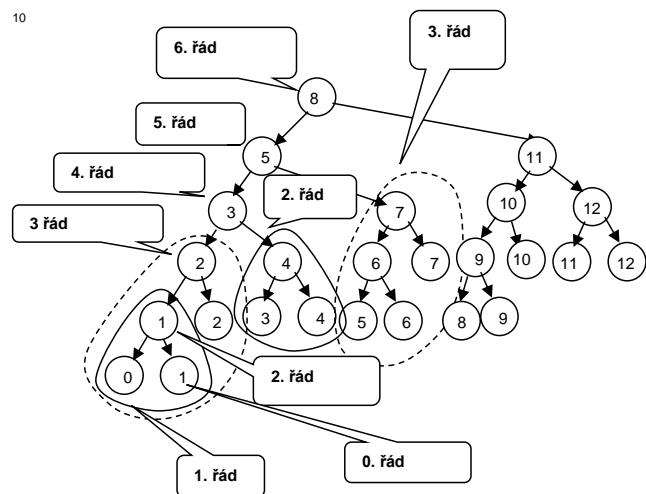
- F-tree i-tého rádu sestává z $F_{i+1}-1$ non-terminálních uzlů a z F_{i+1} terminálních uzlů.
- Je-li $i=0$ nebo $i=1$ je strom reprezentován pouze kořenem a současně terminálním uzlem [0].
- Je-li $i>1$, pak je kořen stromu reprezentován hodnotou F_i , jeho levý podstrom je F-tree rádu $i-1$ (F_{i-1}) a pravý podstrom je F-tree rádu $i-2$ (F_{i-2}), v němž všechny hodnoty uzlů jsou zvýšeny o hodnotu kořene F_i .
- Celkový počet uzlů Fibonacciho stromu i-tého rádu $PocFtree(i)$ je dán vztahy:

$$PocFtree(0)=PocFtree(1)=1$$

$$PocFtree(i)=PocFtree(i-1)+PocFtree(i-2)+1.$$

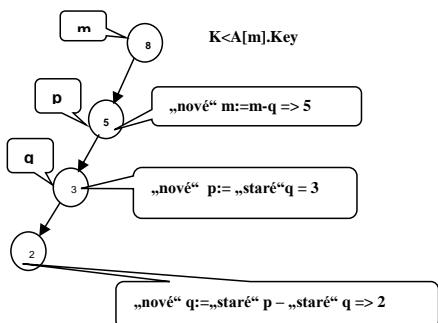
Zatímco binární vyhledávání spočívá na půlení, jehož obrazem je binární strom, Fibonacciho vyhledávání spočívá na rozdelení intervalu na dva nestejně podintervaly, jehož obrazem je Fibonacciho strom. Na obrázku je první interval 0..12, daný nejlevějším a nejpravějším uzlem, rozdelen kořenem a představuje index pole 8. Pokud je hledaný klíč v levém podintervalu, jde o posun doleva a další vyhledávání bude v levém podstromu. V opačném případě se bude vyhledávat v pravém podstromu. Situace znázorňují další obrázky.

Neúspěšné vyhledání končí vždy na levém nebo na pravém terminálu nejnižšího stromu, který je vždy 2. rádu. To poznáme podle toho, že levý uzel ("oproštěný" od přidané hodnoty kořene) - algoritmu q - má hodnotu nula, nebo že pravý uzel ("oproštěný" od hodnoty kořene) - v algoritmu p - má hodnotu 1.

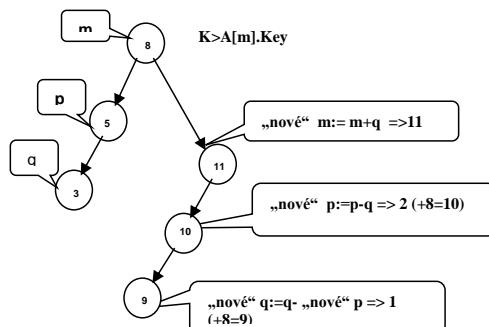
Fibonacciho


Posun doleva

Další vyhledávání bude v levém podstromu. Pomocné ukazatele m, p a q při tomto "posunu" po diagonále dolů dostanou nové hodnoty, jak je naznačeno v obrázku.

**Posun doprava**

Při vyhledávání v pravém podstromu dojde k následující změně indexových ukazatelů:



Vyhledávací algoritmus má tvar:

```

m:=F(1);
p:=F(1-1);
q:=F(1-2);
TERM:=false;
while (K<>A[m].Key) and not TERM do begin
  if K<A[m].Key
  then (* hledání pokračuje v levém podstromu *)
    if q=0
    then TERM:=true (* search končí na nulovém terminálu *)
    else begin (* posun levého syna po diagonále *)
      m:=m-q; p1:=q;

```

```

q1:=p-q;  p:=p1;
q:=q1;
end else (* search pokračuje v pravém podstromu *)
if p=1
then TERM:=true (* search končí na pravém terminálu 1. řádu *)
else begin (* nové hodnoty m, p a q v pravém podstromu *)
    m:=m+q;
    p:=p-q;
    q:=q-p
end; (* if
end
end; (* while *)
Search:= not TERM;

```

Sharova metoda

Sharova metoda řeší případ, kdy skutečná velikost (počet prvků) tabulky je jiná, než je hodnota vhodná pro Uniformní binární nebo Fibonacciho vyhledávání. Metoda postupuje ve dvou krocích:

- V prvním kroku provede rozdělení na největším indexu, který vyhovuje metodě a který je menší než daná velikost.
- Ve druhém kroku zjišťuje, zde je hledaný klíč nalevo nebo naprav od dělicí hodnoty. Když je nalevo, postupuje jako by tabulka měla počet prvků daný rozdělovací (a pro metodu vhodnou) hodnotou. Když je napravo, provede transformaci tabulky posunem začátku pole doprava tak, aby prohledávaná část tabulky měla opět vyhovující počet prvků.



Uniformní binární i Fibonacciho vyhledávání je zvláštním a okrajovým příkladem v oblasti vyhledávání. Ilustruje způsoby hledání efektivních algoritmů ve speciálních případech.

Fibonacciho vyhledávání je ukázkovým příkladem toho, že krátký a přehledný algoritmus je bez znalosti jednoduchého, ale málo známého principu Fibonacciho stromu, zcela nesrozumitelný i pro jinak zkušeného programátora. Je to podnět k tomu, že správná dokumentace musí kromě základní navigace v programu obsahovat i popis méně známých principů.

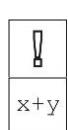
Kontrolní otázky a příklady

- Co je to strukturální ekvivalence?
- Co je to vyhledávání?
- Co je to přístupová doba?
- Má smysl použít cyklus FOR pro vyhledávání?
- Co je to zkratové vyhodnocování?
- Vysvětlete pojmen zaslepení.
- Co je to zarážka?
- Jaká je výhoda vyhledávání v seřazené posloupnosti?
- Jaká je nevýhoda operací insert a delete v tabulce seřazených položek oproti

neseřazeným položkám?

- Jaká je maximální doba přístupu k prvku tabulky s binárním vyhledáváním?
- Jak se liší v dob přístupu normální a Dijkstrova varianta binárního vyhledávání?
- Kdy je výhodné použití Uniformního nebo Fibonacciho vyhledávání?
- Čím se liší Fibonacciho vyhledávání od metod binárního vyhledávání?
- Nakreslete Fibonacciho strom 5. řádu.
- Jak se pozná konec neúspěšného vyhledávání ve Fibonacciho vyhledávání.
- V čem spočívá princip vyhledávání s adaptivní rekonfigurací položek podle četnosti přístupů a jaké má přednosti?

4.8 Binární vyhledávací stromy

DEF

x+y

4.8 Binární vyhledávací strom - BVS (*binary search tree*) je jednou z nejpoužívanějších implementací pro plně dynamické vyhledávací tabulky.

Uspořádaný strom je kořenový strom, pro jehož každý uzel platí, že n-tice kořenů podstromů uzlu je uspořádaná.

Binární vyhledávací strom je uspořádaný binární strom pro jehož každý uzel platí, že klíče všech uzlů levého podstromu jsou menší než klíč v uzlu a klíče všech uzlů pravého podstromu jsou větší než klíč v uzlu.

Rekurzivní definice:

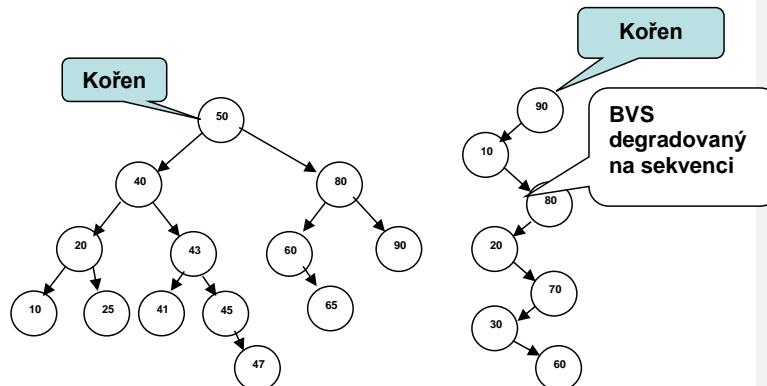
Binární vyhledávací strom je buď prázdný nebo sestává z kořene, hodnota jehož klíče je větší než hodnota všech klíčů levého binárního vyhledávacího podstromu a je menší než hodnota všech klíčů pravého binárního podstromu.

Pozn. Rekurzivní definice BVS vede k rekurzivním zápisům řady algoritmů nad BVS

Průchod InOrder binárním vyhledávacím stromem dává posloupnost prvků seřazenou podle velikosti klíče.

Cesta k terminálnímu uzlu váhově vyváženého BVS s N uzly prochází $\lg_2 N$ uzly.

Příklady BVS



Průchod InOrder BVS stromem dává seřazenou posloupnost:

1 strom: 10,20,25,40,41,43,45,47,50,60,65,80,90

2 strom: 10,20,30,60,70,80,90

Vyhledávání v BVS

Vyhledávání v BVS je podobné binárnímu vyhledávání v seřazeném poli.

Je-li vyhledávaný klíč roven kořeni, vyhledávání končí úspěšným vyhledáním.

Je-li klíč menší než klíč kořene, pokračuje vyhledávání v levém podstromu, je-li větší, pokračuje v pravém podstromu. Vyhledávání končí neúspěšně, pokud je prohledávaný (pod)strom prázdný.

Datové typy

Datové typy používané pro BVS jsou obdobné jako pro dvojsměrný seznam

resp. binární strom:

```
type
  TUk=^TUzel;
  TUzel=record
    Klic:TKlic;
    Data:TData;
    LUK, Puk:TUk
  end;
```

Rekurzívní zápis Search

x+y

Rekurzívní zápis funkce vyhledávání v BVS

```
function Search(UkKor:TUk; K:TKlic):Boolean;
(* UkKor je ukazatel kořene BVS *)
begin
  if UkKor<>nil
  then (* Strom je neprázdný *)
    if UKKor^.Klic=K
    then (* Našel hledaný klíč *)
      Search:=true
    else (* Nenašel *)
      if UKKor^.Klic>K
      then (* hledání pokračuje v levém podstromu *)
        Search:=Search(UkKor^.LUk, K)
      else (* hledání pokračuje v pravém podstromu *)
        Search:=Search(UkKor^.Puk, K)
      else (* cesta končí na termin. uzlu – nenašel *)
        Search:=false
  Search:=false
end; (* function *)
```

Search jako procedura

x+y

Varianta vyhledávání v BVS jako procedura, která vrací odkaz na nalezenou položku ve výstupním parametru Kde. V případě nenalezení je Kde=nil.

```
procedure SearchTree(UkKor:TUk; K:TKlic;
                     var Kde:TUk);
begin
  if UkKor = nil
  then
    Kde:=nil
  else
    if UKKor^.Klic <> K
    then
      if UKKor^.Klic > K
      then
        SearchTree(UkKor^.LUk, K, Kde)
      else
        SearchTree(UkKor^.Puk, K, Kde)
    else Kde:=UkKor (* našel a nastavuje výstupní parametr Kde *)
end; (* procedure *)
```

Nerekurzívní Search



Nerekurzívní zápis funkce Search.

```

function Search(UkKor:TUk; K:TKlic):Boolean;
(* Nerekurzívní zápis vyhledávání v BVS *)

var Fin:Boolean (* Řídicí proměnná cyklu *)
begin
    Search:=false;
    Fin:=UkKor=nil;
    while not Fin do begin
        if UkKor^.Klic=K
        then begin
            Fin:=true;
            Search:=true;
        end else
            if UkKor^.Klic > K
            then UkKor:=UkKor^.LUk (* Pokračuj vlevo *)
            else
                UkKor:=UkKor^.PUk; (* Pokračuj v pravém podstromu *)
        if UkKor=nil
        then Fin:=true
    end (* while *)
end; (* procedure *)

```

Insert v BVS

Operace Insert aplikuje „aktualizační sémantiku“, tzn., že v případě, že uzel s daným klíčem existuje, přepíše operace stará data aktuálními. V případě, že uzel s daným klíčem neexistuje, vloží operace nový uzel jako terminální uzel tak, aby se zachovala pravidla BVS.

Pro zkrácení zápisu použijeme tuto pomocnou procedura pro vytvoření uzlu:

```

procedure VytvorUzel(var UkUzel:TUk; K:TKlic;
D:TData);
begin
    new(UkUzel);
    UkUzel^.Klic:=K;
    UkUzel^.Data:=D;
    UkUzel^.Luk:=nil;
    UkUzel^.Puk:=nil
end;

```

Rekurzívní Insert



Rekurzívní zápis operace Insert.

```

procedure Insert (var UkKor:TUk; K:TKlic; D:TData);
begin
    if UkKor=nil
    then
        VytvorUzel(UkKor, K, D) (* vytvoření kořene či term. uzlu *)

```

```

else
  if K<UkKor^.Klic
  then
    Insert(UkKor^.LUk,K,D) (*pokračuj v levém podstromu *)
  else
    if K>UkKor^.Klic
    then
      Insert(Ukkor^.PUk,K,D) (* pokračuj vpravo *)
    else UkKor^.Data:=D (* přepiš stará data novými *)
end; (* procedure *)

```

**Nerekurzívní
Insert**

x+y

Nerekurzívní zápis operace Insert využívá operace Search upravené tak, aby vracela polohu nalezeného uzlu pro přepis starých dat novými, nebo polohu uzlu, ke kterému se připojí nový vkládaný terminální uzel.

```

procedure SearchIns(UkKor:TUk;K:TKlic;
                     var Found:Boolean; var Kde:TUk);
(* Vyhledání za účelem nerekurzívního vkládání *)
var PomUk:TUk;
begin
  if UkKor=nil
  then begin
    found:=false; Kde:=nil; (* prázdný BVS *)
  end else
    repeat
      Kde:=UkKor; (* uchování výstupní hodnoty Kde *)
      if UkKor^.Klic > K
      then
        UkKor:=UkKor^.Luk (* posun doleva *)
      else
        if UkKor^.Klic < K
        then
          UkKor:=UkKor^.Puk (* posun doprava *)
        else
          found:=true (* našel, vrací hodnotu Kde *)
      until found or (UkKor=nil)
  end; (* procedure *)

```

Procedura Insert v sobě vyvolá pomocnou proceduru Search a přepíše nebo vloží data.

```

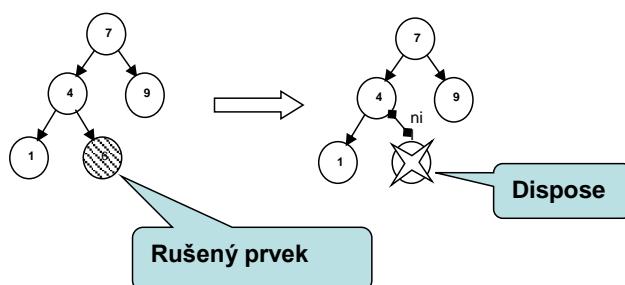
x+y
procedure Insert (var UkKor:TUk; K:TKlic; D:TData);
var
  PomUk, Kde:TUk;
  Found:Boolean;

begin
  SearchIns (UkKor, K, Found, Kde) ;
  if Found
  then
    Kde^.Data:=D; (* přepsání starých dat novými *)
  else
    VytvorUzel (PomUk, K, D) ;
    if Kde =nil
    then UkKor:=PomUk; (* prázdný strom, nový uzel je kořen *)
    else (* neprázdný strom, nový uzel se připojí k uzlu jako terminál *)
      if Kde^.Klic>K
      then (* nový uzel se připojí vlevo *)
        Kde^.LUk:=PomUk
      else (* uzel se připojí vpravo *)
        Kde^.PUk:=PomUk
  end; (* procedure *)

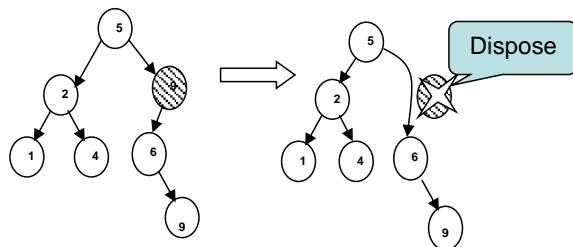
```

Operace Delete Rušení uzlu – operace Delete, je vždy složitější, než operace Insert. Rušení terminálního uzlu je snadné.

Příklad rušení terminálního uzlu

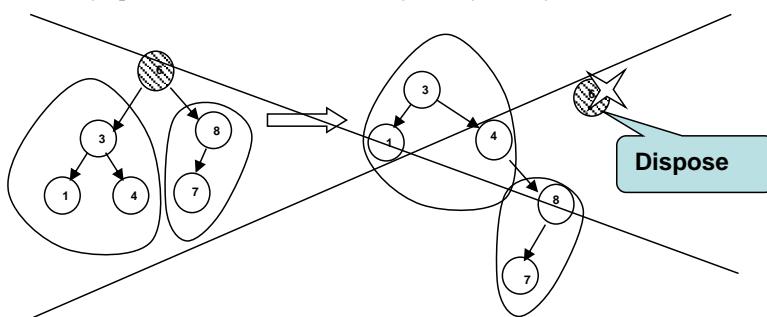


Zrušení uzlu, který má jen jednoho syna, je také snadné. Synovský uzel se připojí na nadřazený uzel rušeného uzlu.



Zrušit uzel, který má dva podstromy lze také tak, že levý podstrom připojíme na nejlevější uzel pravého podstromu nebo tak, že pravý podstrom rušeného uzlu připojíme na nejpravější uzel levého podstromu, jak je to uvedeno na následujícím obrázku. Toto řešení je však pro vyhledávací stromy nepřijatelné, protože zbytečně zvyšuje výšku stromu a tím i maximální dobu vyhledávání.

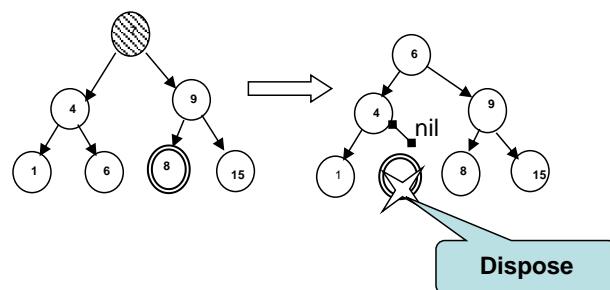
Nevhodný způsob rušení uzlu se dvěma synovskými uzly.



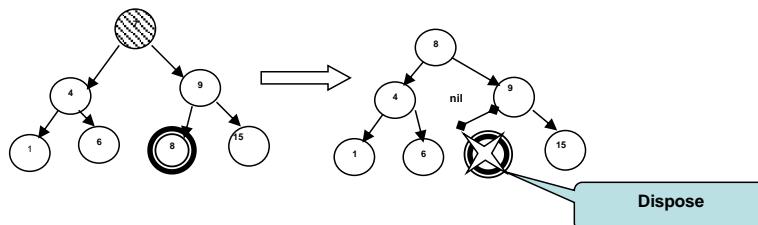
Rušení uzlu se dvěma synovskými uzly:

Rušený uzel se dvěma syny se nezruší „fyzicky“, ale jeho hodnota se přepíše hodnotou takového uzlu, který lze zrušit snadno a přitom při přepisu nesmí dojít k porušení uspořádání (pravidel) BVS. Takovým uzlem je nejpravější uzel levého podstromu rušeného uzlu nebo symetricky nejlevější uzel pravého podstromu rušeného uzlu.

Příklad rušení uzlu se dvěma podstromy. Rušený uzel 7 se přepíše uzlem 6 a ten se zruší.



Symetrické řešení



Rekurzívní Delete Rekurzívní zápis procedury Delete obsahuje pomocnou rekurzívní proceduru Del, která prochází po pravé diagonále levého podstromu a hledá prvek, jehož hodnotou se přepíše rušený uzel a který je následně rušen.

Algoritmus rekurzívní procedury je netriviální a u písemných zkoušek se neočekává jeho „rekonstrukce“. U závěrečné zkoušky lze žádat vysvětlení předloženého algoritmu.



```

procedure Delete(var UkKor:TUk; K:TKlic);
var
    PomUk:TUk;

procedure Del(var Uk:TUk);
(* Pomocná procedura Del se pohybuje po pravé diagonále levého podstromu rušeného uzlu a
hledá nejpravější uzel. Když ho najde, je globální proměnná - ukazatel na uzel PomUk -
přepsán ukazatelem na uzel Uk a Uk je připraven pro následnou operaci dispose.*)
begin
    if Uk^.PUk <>nil
    then
        Del(Uk^.PUk) (* pokračuj v pravém podstromu *)
    else begin (* nejpravější uzel je nalezen, přepsání a uvolnění uzlu *)
        PomUk^.Data:=Uk^.Data;
        PomUk^.Klic:=Uk^.Klic;
        PomUk:=Uk;
        Uk:=Uk^.LUk (* Uvolnění uzlu Uk! Pozor! V proceduře Del je Uk ukazatelová
složka uzlu nadřazeného k uzlu Uk *)
    end
end; (* Konec pomocné procedure Del *)

begin (* tělo hlavní procedury *)
    if UkKor <> nil
    then (* vyhledávání neskončilo; hledaný uzel může stále být v BVS *)
        if K < UkKor^.Klic
        then
            Delete (UkKor^.LUk,K) (* pokračuj v levém podstromu *)
        else
            if K > UkKor^.Klic
            then Delete(UkKor^.PUk,K) (* pokračuj v pravém podstromu *)
            else begin (* uzel UkKor se má rušit *)
                PomUk:=UkKor;

```

```

if PomUk^.PUk=nil
then(* uzel nemá pravý podstrom; levý podstrom se připojí na nadřazený uzel *)
    UkKor:=PomUk^.LUk
else (* rušený uzel má pravý podstrom; bude přepsán nejpravějším uzlem
levého podstromu uvnitř procedury Del. Je-li levý podstrom prázdný, připojí se pravý
podstrom na nadřazený uzel *)
    if PomUk^.LUk=nil
    then
        UkKor:=PomUk^.PUk (* připojení pravého podstromu *)
    else
        Del(PomUk^.LUk); (* Pomocná rekurz. procedura Del *)
        dispose (PomUk); (* uvolnění uzlu *)
    end (*if K>UkKor^.LUK,K*)
else (* zde může být akce, když uzel nebyl nalezen; normálně se neděje nic *)
end (* delete *).

```

Nerekurzívní zápis procedury Delete obsahuje několik pomocných procedur. Procedura DeleteSearch vyhledává rušený prvek. Vrací polohu rušeného prvku, polohu uzlu jemu nadřazenému a boolevskou hodnotu o straně, ke které je rušený uzel k nadřazenému uzlu připojen. Procedura rušení musí v počáteční fázi ošetřit případ rušení kořene a rušení jediného uzlu.



```

procedure Delete(var UkKor:TUk; K:TKlic);
var
    UkOtce,UkPraOtce:TUk;
procedure DeleteSearch(UkKor:TUk;K:TKlic;
    var Found,OtecZleva:Boolean;
    var UkOtce,UkPraOtce:TUk);
(* Pomocná procedura DeleteSearch je uvedena později. Procedura hledá klíč K v BVS
zadaný ukazatelem UkKor. Parametry Found, OtecZleva, UkPraOtce a UkOtce jsou výstupní
parametry. Je-li strom prázdný pak Found=false and ukazatele UkOtce a UkPraOtce nejsou
definovány. Je-li Found=true a nalezený uzel je kořen, pak UkPraOtce=nil, UkOtce=UkKor
and OtecZleva je nedefinovaný. Když nalezený uzel je vnitřní uzel stromu, tak Found=true,
UkPraOtce je uzel nadřazený otci, OtecZleva=true je v případě, že otec je levým synem
practce. Když je uzel nenalezen, je Found=false a ostatní parametry jsou nedefinované.*)
begin
    (* tělo procedury bde uvedeno později *)
end;

procedure Nejprav(var UkOtce:TUk);
(* Procedura najde Nejpravější uzel levého podstromu, přepíše data otce nalezeným uzlem,
uvolní nejpravější uzel a vrátí ho v parametru pro pozdější dispose *)
begin
    (* tělo procedury bde uvedeno později *)
end;
begin
    DeleteSearch(UkKor,K,Found,OtecZleva,UkOtce,UkPraOtce);
    if Found
    then begin

```



```

if UkOtce^.PUk=nil
then (* Nalezený nemá pravého syna, jde zrušit snadno *)
    if UkPraOtce=nil
        then UkKor :=UkOtce^.LUk      (* Rušený je kořen s jediným levým
synem; syn (může být i prázdný) se stane kořenem ,je-li levý prázdný, zůstane jen kořen *)
        else (* Připoj levého syna (může být i prázdný) na praoctee *)
            if OtecZleva
                then UkPraOtce^.LUk:=UkOtce^.LUk
                    (* připoj syna (nil) k praoctci zleva *)
            else UkPraOtce^.PUk:=UkOtce^.LUk
                    (* připoj syna (nil) zprava *)
        else (* nalezený má pravého syna *)
            if UkOtce^.LUk=nil
                then (* nalezený nemá levého syna *)
                    if UkPraOtce=nil (* Rušený je kořen s jediným pravým synem; syn (může
být i prázdný) se stane kořenem *)
                        then UkKor:=UkOtce^.PUk
                    else
                        if OtecZleva
                            (* připoj pravého syna (může být i prázdný) k praoctci *)
                            then UkPraOtce^.LUk:=UkOtce^.PUk
                                (* připoj syna (nil) zleva *)
                            else UkPraOtce^.PUk:=UkOtce^.PUk
                                (* připoj syna (nil) zprava *)
                        else Nejprav(Otec);
                        dispose(UkOtce) (* fyzické rušení uzlu *)
                    end (* if Found *)
                end; (* procedure *)
end;

procedure DeleteSearch(UkKor:TUk;K:TKlic; var
Found,OtecZleva:Boolean; var UkOtce,UkPraOtce:TUk);
(* Procedura hledá rušený prvek. Když ho najde, pak není-li to kořen, vrací také ukazatel na
nadřazený uzel a informaci o tom, je-li k nadřazenému uzlu připojen zleva nebo zprava *)
var
    Fin:Boolean (* řídící proměnná cyklu *)
begin
    Found:=true;
    UkOtce:=UkKor;
    if UkKor<>nil
    then
        if UkKor^.Klic=K
        then begin
            Found:=true; (* končí se, nalezený je kořen *)
            UkPraOtce=nil; (* kořen nemá nadřazený uzel *)
        end else begin
            Fin:=false;
            while not Fin do begin
                if UkOtce^.Klic=K
                then begin

```



```

        Found=true;
        Fin:=true;
    end else begin
        UkPraOtce:=UkOtce;
        (* Uchování ukazatele na nadřazený uzel *)
        if Otec^.Klic > K
        then begin
            (* uchování směru připojení k nadřazenému uzlu *)
            OtecZleva:=true;
            UkOtce:=UkOtce^.LUk
        end else begin
            OtecZleva:=false;
            UkOtce:=UkOtce^.PUk
        end;
        end; (* if *)
        if UkOtce=nil
        then Fin:=true
    end (* while *)
end; (* if *)
end (* procedure *)

```

```

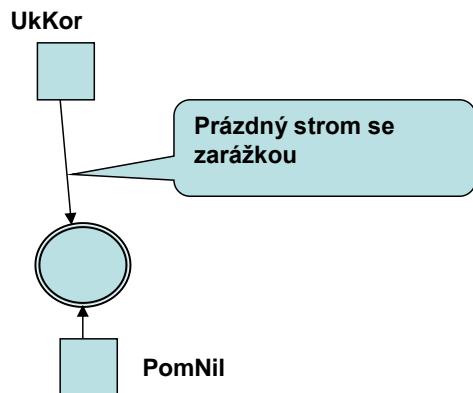
procedure Nejprav(var UkOtce:TUk);
var
    Nejpr,OtecNejpr:TUk;
begin
    Nejpr:=UkOtce^.LUk;
    if Nejpr^.PUk<>nil
    then begin (* hledej nejpravější uzel *)
        repeat
            OtecNejpr:=Nejpr;
            Nejpr:=Nejpr^.PUk
        until Nejpr^.PUk=nil;
        OtecNejpr^.PUk:=Nejpr^.LUk
    end else begin (* Nejpr je Nejprav sám *)
        UkOtce^.LUk:=Nejpr^.LUk;
    end;
    UkOtce^.Klic:=Nejpr^.Klic; (* přepis klíče *)
    UkOtce^.Data:=Nejpr^.Data; (* přepis dat *)
    UkOtce:=Nejpr
    (* Otec, ve skutečnosti nejpravější, bude uvolněn operací dispose *)
end;

```

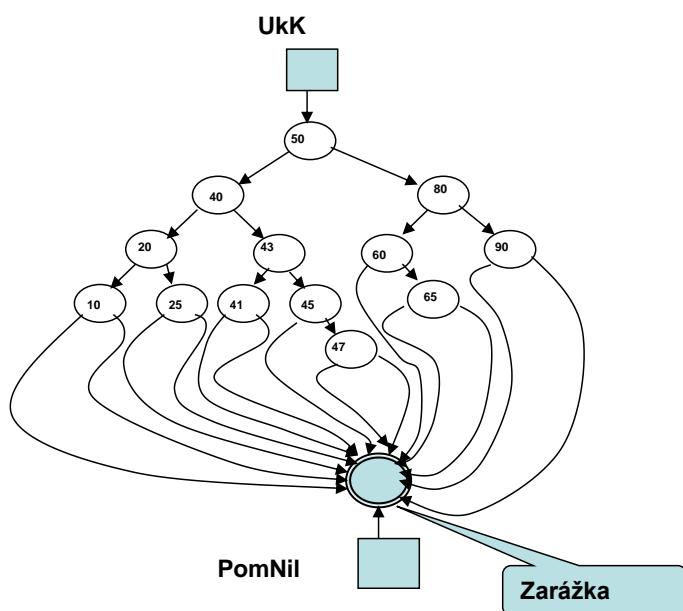


4.9 Binární vyhledávací strom se zarážkou používá pomocný uzel jako zarážku, do které vloží hledaný klíč. Tento uzel vždy najde, ale podle ukazatele pozná, za jde o skutečný uzel nebo o zarážku. Ukazatel na tento uzel slouží jako "pomocný nil"

Prázdný BVS se zarážkou



BVS se
zarážkou



Inicializace tabulky implementované BVS se zarážkou.

```

procedure TInit(var UkKor:TUk);
(* proměnná PomNil je globální *)
begin
  new (PomNil);

```

```

UkKor:=PomNil
end;

Fukce Search

function SearchTree(Uk:TUk; K:TKlic) :Boolean;
(* proměnná PomNil je globální *)
begin
  PomNil^.Klic:=K;
  while Uk^.Klic <> K do begin
    if Uk^.Klic>K
    then Uk:=Uk^.Luk
    else Uk:= Uk^.Ruk
  end; (* while *)
  SearchTree:= Uk<>PomNil
end; (* procedure *)

```

BVS se zarázkou je příklad použití zarážky v nelineární struktuře. Zarážka umožňuje vynechat opakující se test na konec a urychluje algoritmus.

4.10 BVS se zpětnými ukazateli



4.10 Binární vyhledávací strom se zpětnými ukazateli má význam pouze tehdy, chceme-li se při průchodu InOrder vyhnout rekurzi nebo použítí zásobníku.

Operace **Search** je stejná, jako u normálního BVS.

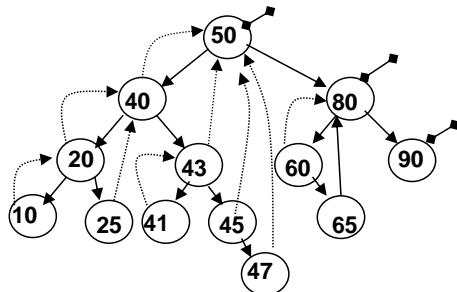
Operace **Insert** musí při vkládání nového terminálního uzlu respektovat následující pravidla:

- Zpětný ukazatel kořene ukazuje na nil (všechny uzly vedlejší diagonály ukazují na nil...)
- Zpětný ukazatel levého syna ukazuje na svého otce
- Zpětný ukazatel pravého syna dědí ukazatele od otce (ukazuje tam kam otec).

Operace **Delete** se provádí stejně jako u normálních BVS, ale v případě, že se ruší prvek s jedním a to s levým podstromem, musí se udělat korekce zpětných ukazatelů na pravé diagonále podstromu tak aby ukazovaly na uzel nadřazený rušenému.

DEF

Ukázka BVS se zpětnými ukazateli.



Definujme typy:

```

TUk=^TPol;
TPol=record
  Klic:TKlic;
  Data:TData;
  LUk,PUk,ZpetUk:TUk;
end;

procedure Nejlev(UkKor:TUk; var UkNaNejlev:TUk);
(* procedura vrátí ukazatel na nejlevější uzel stromu *)
begin
  UkNaNejlev :=UkKor;
  while UkKor <> nil do begin
    UkNaNejlev :=UkKor;
    UkKor:=UkKor^.LUk
  end (* while *)
end
  
```

InOrder

Pak procedura InOrder bude mít tvar:

```

 procedure Inorder(UkKor:TUk; var DL:TDlist);
(* Průchod Inorder vkládá data do dvojsměrného seznamu *)
var
  Fin:Boolean;
  UkNaNejlev:TUk;
begin
  DListInit (DL); (* Inicializace dvojsměrného seznamu *)
  Nejlev(UkKor,UkNaNejlev);
  Fin:=UkNaNejlev=nil;      (* řídící proměnná cyklu *)
  while not Fin do begin
    DInsertLast(DL, UkNaNejlev^.Data); (*vkládání do sezn.*)
    if UkNaNejlev^.PUk<>nil
      then Nejlev(UkNaNejlev^.PUk, UkNaNejlev)
    else
  
```

```

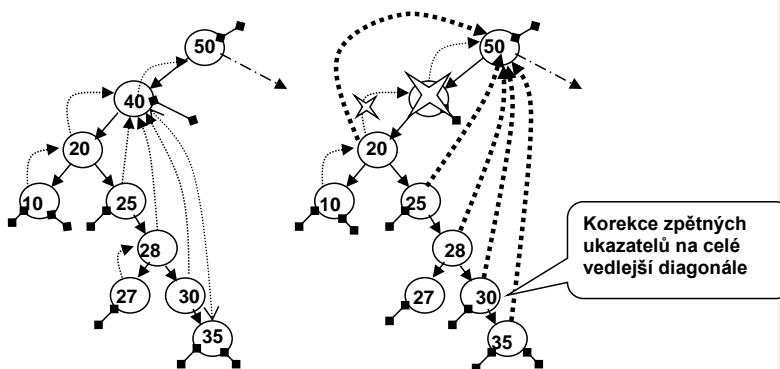
if UkNaNejlev^.ZpetUk=nil
  then Fin:=true
  else UkNaNejlev:=UkNaNejlev^.ZpetUk (* posun zpět *)
end;
end; (* procedure *)

```

Delete

Operace Delete se provádí stejně, jako u normálního BVS, ale v případě rušení uzlu s jediným a to levým podstromem se musí provést korekce zpětných ukazatelů na celé pravé diagonále podstromu rušeného uzlu tak, aby ukazovaly na uzel nadřazený rušenému.

Situaci znázorňuje následující obrázek.



4.11 Kontrolní otázky a úlohy

- Vytvořte následující nerekurzívní varianty zápisu procedury pro vyhledávání v BVS:
 - procedura vrátí Booleovský parametr Search a ukazatel UkKde pro nalezený uzel (v případě Search=false je UkKde nedefinováno)
 - procedure InsertSearch1(UkKor:TUK; K:TKlic;
var Search:Boolean; var UkKde:TUK);
 - procedura vrátí ukazatel UkKde na nalezený uzel a nil v případě neúspěšného vyhledávání.
 - procedure InsertSearch2(UkKor:TUK; K:TKlic; var UkKde:TUK);
- Je dán BVS (nevývážený). Je zadán (maximální) počet jeho uzelů. Vytvořte jeho váhově vyváženou verzi. Zapište řešení ve formě rekurzívní i nerekurzívní procedury. Zvolte vhodnou datovou strukturu uzelů a definujte potřebné typy.
Pozn. Řešte s použitím pomocného pole, do kterého vložíte všechny hodnoty nevyváženého stromu. Z pole pak vytvořte nový, váhově vyvážený strom.
 - Provedte diskuzi, za kterých okolností je výhodnější použít binární vyhledávání než vyhledávání v BVS.
 - Čím je dán nejhorší případ přístupové doby v (nevýváženém) BVS?
 - V čem spočívá princip a BVS jakou má výhodu BVS se zarážkou?
 - K čemu se používá se zpětnými ukazateli?
 - Čím se liší BVS se zpětnými ukazateli od normálního BVS?

4.12 AVL strom

AVL stromy

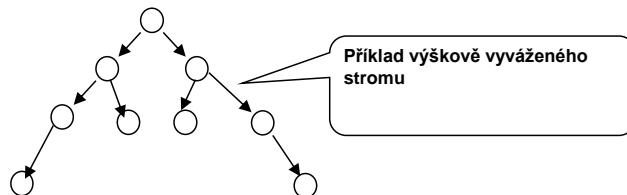
DEF

- **AVL strom** je výškově vyvážený strom, nazvaný podle ruských matematiků Adelson-Velskij a Landis, kteří dokázali, že jeho výška je maximálně o 45% vyšší než váhově vyvážený strom. To znamená, že složitost "O" vyhledávání v binárním stromu, založená na nevětší vzdálenosti od kořene k listu je ve srovnání s váhově vyváženým BVS 1.45 krát vyšší - tedy $1.45 \cdot \lg_2 N$, pro strom o N uzlech.
- Výškově vyvážený binární vyhledávací strom je strom, pro jehož každý uzel platí, že výška jeho dvou podstromů je stejná nebo se liší o 1.
- Ustavení nové vyváženosti při poruše váhově vyvážených stromů, způsobené vložením nového nebo zrušením stávajícího uzlu stromu je pracné a obtížné.
- Znovuustavení výškově vyváženosti AVL stromu lze provést nepříliš pracnou rekonfigurací několika uzlů v okolí tzv. kritického uzlu.
- Kritický uzel je nejvzdálenější uzel od kořene, v němž je v důsledku vkládání nebo rušení porušená rovnováha.



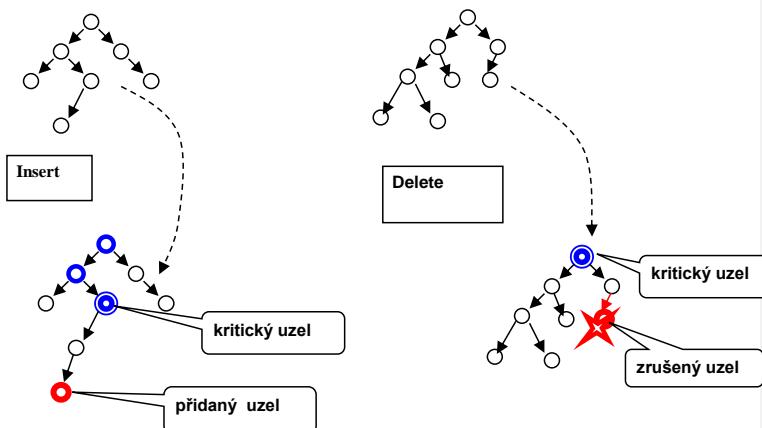
Výškově
vyvážený
strom

$x+y$



Příklad porušení výškové vyváženosti v důsledku operace Insert a Delete a ukázka vzniku **kritického uzlu**.

$x+y$



Rotace

Mechanismu znovuustavení právě porušené výškově vyváženosti se říká **rotace**. Za účelem ustavení vyváženosti se do uzlu stromu AVL přidává složka, které se říká "váha". Budeme ji označovat symbolem " h ". Složka uzlu h , může nabývat hodnot:

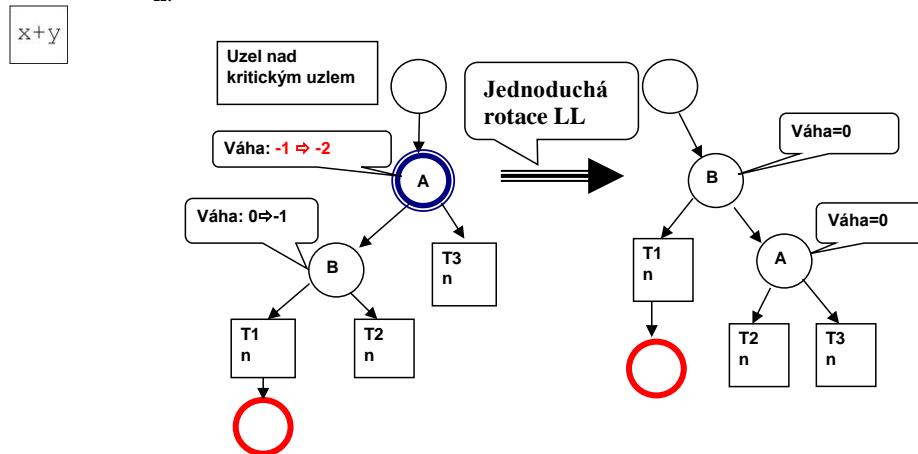
- $h = -2$ uzel má porušenou výškovou vyváženosť doleva
- $h = -1$ uzel je výškově vyvážený, ale je těžký vlevo (levý podstrom je o 1 vyšší než pravý)
- $h = 0$ uzel je absolutně výškově vyvážený
- $h = 1$ uzel je výškově vyvážený, ale je těžký vpravo (pravý podstrom je o 1 vyšší než levý)
- $h = 2$ uzel má porušenou výškovou vyváženosť doprava

Existují čtyři základní rotace, které znovuustavují vyváženosť. Každá z nich má ještě stranově symetrickou variantu. Dvě dvojice rotací jsou pro korekci po operaci Insert - dvojice jednoduchých rotací (LL a RR) a dvojice dvojitých rotací (DLR a DRL). Podobně jsou dvě další dvojice pro operaci Delete. Následující obrázky ilustrují účinek rotací. Pro ukázky budeme používat následující typy:

```
type
  TKlic=string;
  TData=String;

  TUkUz=^TUz;
  TUz=record
    LUK, PUK:TUkUz;
    Vaha:-2..2;
    Klic:TKlic;
    Data:TData;
  end; (* record *)
```

Jednoduchá rotace LL Jednoduchá rotace LL koriguje porušení vyváženosťi vložením nového (tučně orámovaného uzlu). Vznikne kritický uzel (dvojitě orámovaný). Kritický uzel i jemu podřízený levý uzel se posunou zleva doprava. Obdélníčky Ti představují podstromy s výškou označenou symbolem **n**.



```

Algoritmus procedure RotLL(var Nadkrit, Krit:TUKUz);
rotace LL
(* Rekonfigurace uzelů AVL stromu po operci INSERT v okolí kritického
uzlu typu jednoduchá rotace LL *)
var A,B:TUKUz;
begin
  (* Rekonfigurace uzelů *)
  A:=Krit;
  B:=A^.LUk;
  A^.LUk:=B^.PUk;
  B^.PUk:=A;
  (* Napojení na nadkritický uzel *)
  if NadKrit^.LUk=Krit
    then NadKrit^.LUk:=B
    else NadKrit^.PUk:=B;

  (* Korekce váhy zúčastněných uzelů pro operaci INSERT *)
  A^.Vaha:=0;
  B^.Vaha:=0;
end;

```

Překreslete rotaci LL na její stranově symetrickou variantu RR. Upravte symetricky váhy uzelů výchozí i korigované konfigurace.

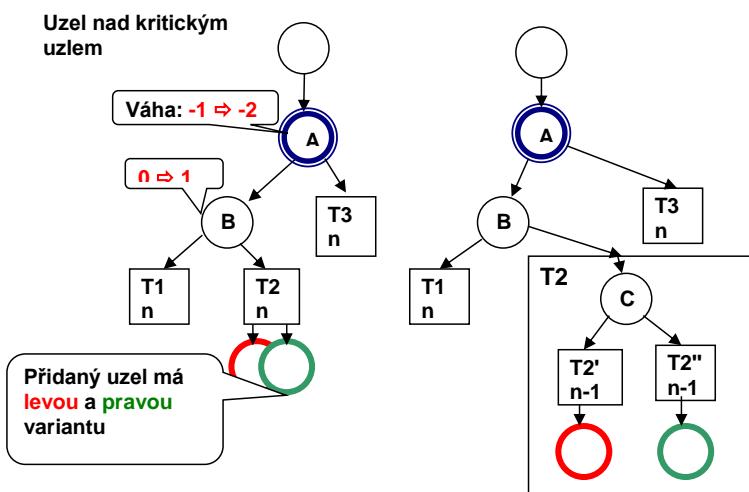


Dvojitá rotace DLR pro Insert

Dvojitá rotace DLR je znázorněna na následujícím obrázku. Uzel, přidaný k podstromu T2, může zvýšit výšku vlevo (po rozkreslení se přidá k podstromu T2') nebo zvýší výšku vpravo (po rozkreslení se přidá k podstromu T2''). V obrázku jsou tyto dvě alternativy (nastane jen jedna z nich) odlišeny barevně.

x+y

Konfiguraci rotace DLR lze překreslit do tvaru uvedeného vpravo



```

Algoritmus procedure RotDLR(var Nadkrit, Krit:TUKUz);
rotace DLR (* Rekonfigurace uzlů AVL stromu po operaci INSERT v okolí kritického
uzlu typu dvojitá rotace LR *)
var
  A,B,C:TUKUz;
begin
  A:=Krit;
  B:=A^.LUk;
  C:=B^.PUk;

  (* Rekonfigurace *)
  B^.PUk:=C^.LUk;
  A^.LUk:=C^.PUk;
  C^.LUk:=B;
  C^.PUk:=A;

  (* Napojení na nadkritický uzel *)
  if NadKrit^.LUk=Krit
    then NadKrit^.LUk:=C
    else NadKrit^.PUk:=C;

  (* Korekce váhy zúčastněných uzlů po operaci INSERT *)
  if C^.Vaha=1
    then begin
  
```

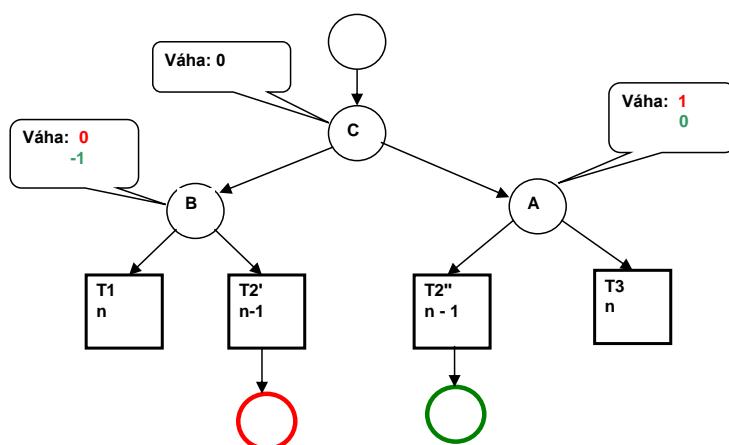
```

B^.Vaha:=-1;
A^.Vaha:=0
end else if C^.Vaha==1
then begin
  B^.Vaha:=0;
  A^.Vaha:=1
end else begin (*pokud je C^.Vaha=0*)
  B^.Vaha:=0;
  A^.Vaha:=0
end; (* if *)
end; (* procedure *)

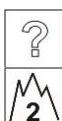
```

Výsledná rotace s barevně odlišenými alternativami je uvedena v následujícím obrázku.

x+y



Nakreslete výchozí situaci, její rozkreslení a výslednou rotaci pro symetrickou operaci DRL.



Algoritmus operace
operace
Insert pro
AVL strom



```

(*****)
(*      InsertAVL      *)
(*****)

procedure InsertAVL(var Kor:TUkUz; K:TKlic; Dat:TData);
(* procedure Insert je zapsána schematickým způsobem; některé příkazy mají tvar komentářů
popisujících jejich sémantiku *)

var
  Opakuj,Nasel:boolean;
  ActUz,KritUz,NadKrit,NadUz:TUkUz;

```

```

begin
    KritUz:=Kor;
    ActUz:=nil; NadKrit:=nil; NadUz:=nil; (* Inicializace *)
    if Kor=nil
    then (* "Vytvoř vyvážený kořen" *)
    else begin
        Opakuj:=true;      (* Inicializace cesty stromem *)
        KritUz:=Kor;
        ActUz:=KritUz;
        NadUz:=nil;

        while Opakuj do begin
            if ActUz^.Klic=K
            then begin          (* Našel shodu *)
                ActUz^.Data:=Dat; (* přepis dat *)
                Nasel:=true;
                Opakuj:=false
            end else

                if ActUz^.Vaha <> 0
                then begin (* Zaznamenání potenciálního krit. uzlu *)
                    NadKrit:=NadUz;
                    KritUz:=ActUz
                end;

                if ActUz^.Klic > K
                then (* jdi doleva *)
(****** stranově symetrická levá verše *****)
                if ActUz^.LUk = nil
                then begin (* bude vkládat vlevo *)
                    Nasel :=false;
                    Opakuj:=false
                end else begin
                    NadUz:=ActUz; (* posun doleva *)
                    ActUz:=ActUz^.LUk
                end;
                end else (* jdi doprava *)
(****** stranově symetrická pravá verše *****)
(* zde by měla být stranově symetrická pravá verše *)
(****** ***** ***** ***** ***** ***** ***** *****)

(****** ***** ***** ***** ***** ***** ***** *****)

end; (* while *)

(* Vkládání nového uzlu *)
if not Nasel
then begin
    Opakuj:=true;
    ActUz:=KritUz;

```

```

while Opakuj do begin
  if K < ActUz^.Klic
  then begin
    (***** stranově symetrická levá verze *****)
    ActUz^.Vaha:=ActUz^.Vaha-1; (* Korekce váhy *)
    if ActUz^.LUk=nil
      then begin
        (* "Připoj na ActUz^.LUk nový vyvážený uzel" a ukonči cyklus *)
        opakuj:=false;
      end else
        ActUz:=ActUz^.LUk (* posun doleva *)
    (***** konec symetrické verze *****)
  end else begin
    (***** stranově symetrická pravá verze *****)
    (* zde by měla být stranově symetrická pravá verze *)

  (*****)
  end (* if k < ... *)
end; (* while opakuj *)

(* Rekonfigurace uzelů pomocí rotací *)

```



Zde se provádí rotace

```

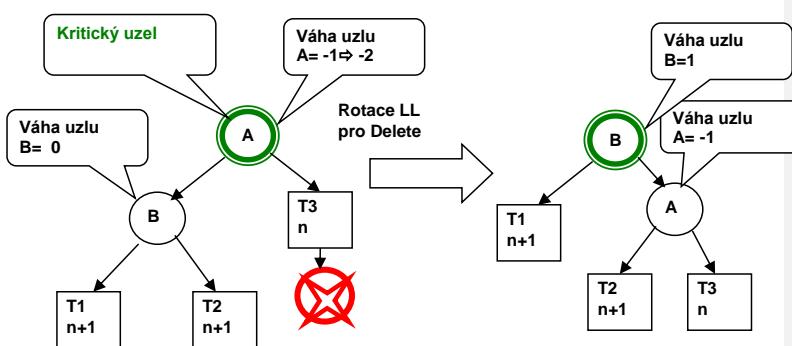
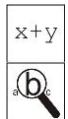
if (NadUz<>nil) and (NadKrit<>nil)
then begin
  if Krituz^.Vaha = -2
  then
    if KritUz^.LUk^.Vaha = -1
    then RotLL(NadKrit,KritUz)
    else RotDLR(NadKrit,KritUz);

  if KritUz^.Vaha = 2
  then
    if KritUz^.PUk^.Vaha = 1
    then RotRR(NadKrit,KritUz)
    else RotDRL(NadKrit,KritUz)
  end; (* if not Nasel *)
end; (*(NadUz<>nil) and (NadKrit<>nil)*)

end (* if Kor=nil *)
end; (* procedure InsertAVL *)

```

Jednoduchá rotace LL pro Delete Řeší porušení rovnováhy po zrušení terminálního uzlu. Situaci určenou pro tuto rotaci ukazuje následující obrázek. Dvojitě orámovaný uzel je vzniklý kritický uzel.



```

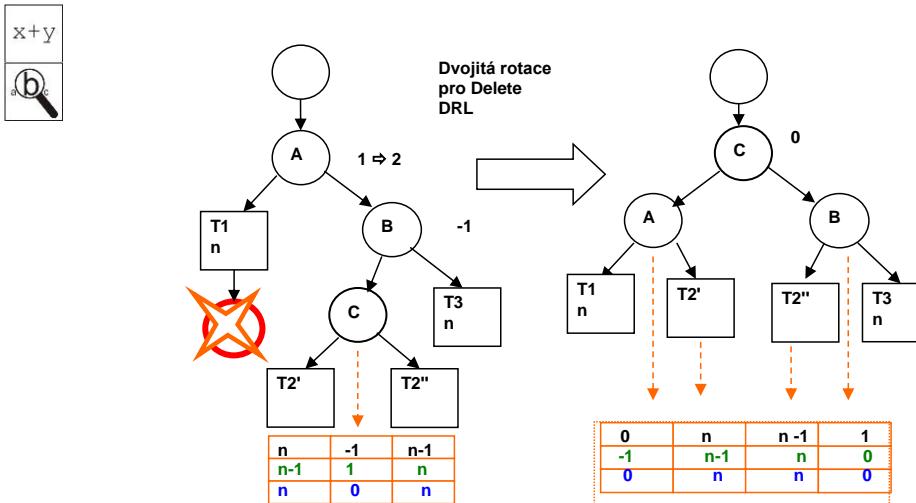
procedure DelRotLL(var nadKrit,KritUz:TUkUz);
(* Jednoduchá rotace LL pro nerekursívní zápis operace DELETE
nad AVL stromem *)
var
  A,B:TUkUz;
begin
  (* Rekonfigurace uzelů *)
  A:=KritUz;
  B:=A^.LUk;
  A^.LUk:= B^.PUk;
  B^.PUk:=A;

  (* Napojení na nadkritický uzel *)
  if NadKrit^.LUk=KritUz
  then NadKrit^.LUk:=B
  else NadKrit^.PUk:=B;

  (* Vyházení uzelů *)
  if B^.Vaha=0
  then begin
    B^.Vaha:=1;
    A^.Vaha:=-1
  end else begin
    B^.Vaha:=0;
    A^.Vaha:=0
  end; (* if B^ *)
end; (* procedure DelRotLL *)

```

Dvojitá rotace DRL Dvojitá rotace DRL je nejsložitější operací. Po zrušení uzlu se váha uzlu A změní z 1 na 2, váha uzlu B bude rovna -1. Váha uzlu C má tři alternativy, v obrázku rozlišené barevně. **pro Delete** Situaci po rotaci znázorňuje pravá část obrázku, níž jsou tabulkou vyznačeny alternativy vah uzlů A a B a výšek stromů T_2' a T_2'' .



```

procedure DelRotDRL(var nadKrit,KritUz:TUkUz);
(* Dvojitá rotace DRL pro nerekursívní zápis oprace DELETE nad AVL stromem *)
var
  A,B,C:TUkUz;
begin
  (* Inicializace *)
  A:=KritUz;
  B:=A^.PUk;
  C:=B^.LUk;
  (* Rekonfigurace *
  A^.PUk:=C^.LUk;
  C^.LUk:=A;
  B^.LUk:=C^.PUk;
  C^.PUk:=B;

  (* Připojení na nadkritický uzel *)
  if NadKrit^.LUk = KritUz
  then NadKrit^.LUk:=C
  else NadKrit^.PUk:=C;

  (* Vyvážení uzlů *)
  if C^.Vaha = 1
  then A^.Vaha:=-1
  else A^.Vaha:=0;

  if C^.Vaha = -1
  then B^.Vaha:=1
  else B^.Vaha:=0;

  C^.Vaha:=0;

```

```
end; (* procedure *)
```

Schema operace Delete pro AVL strom.

```
(* *****)
(* Schema operace DeleteAVL *)
(* *****)

procedure DeleteAVL( var Koren:TUkUz; Klic:TKlic);
(* procedure je zapsána zcela schematicky s využitím textové a pseudopascalovské notace *)
type
  TStackEl=record
    Uk:TUkUz;
    Zleva:boolean
  end;
var
  Konec:boolean;
  A,B:TukUz;
  Otec,PomEl:TStackEl;

(* Sada operací nad zásobníkem, jehož elementem je typ TStackEl *)

procedure SINIT; begin end;
procedure POP; begin end;
procedure TOP(E1:TStackEl); begin end;
procedure PUSH(UkUz:TStackEl); begin end;
function SEMPTY:boolean; begin end;

var
  KritUz, NadKrit:TUkUz;
begin
  SINIT; (* inicializace zásobníku *)

  (*
    1: Při vyhledávací cestě za rušeným uzlem ukládej do zásobníku ukazatele uzlů (počínaje kořenem) a směr, jímž se z uzlů postupuje dále.

    2: Jestliže má nalezený rušený uzel dva syny, přejdi na provádění bodu 3 a 4; jinak zruš uzel způsobem obvyklým pro případ listu nebo uzlu s jedním synem.

    3: Vyhledej nejpravější uzel v levém podstromu rušeného uzlu; cestou ukládej do zásobníku ukazatele na uzly ze směru postupu z uzlu.

    4: Přepiš hodnotu rušeného uzlu nalezeným nejpravějším uzlem a nejpravější uzel zruš.

    5: Postupuj cestou v zásobníku zpět od zrušeného uzlu takto:
  *)

```

```

Konec:=false;
while not Konec do begin
    TOP(Otec); POP;
    if Otec.Zleva
    then begin
        (***** symetrická levá verze *****)
        (* při návratu zleva zvyš váhu otce o 1 *)
        Otec.Uk^.Vaha:=Otec.Uk^.Vaha + 1;
        case Otec.Uk^.Vaha of
            0: if Otec.Uk=Koren
                then Konec:=true
                else (* pokračuj *);
            1: Konec:=true;
            2: (* "Otec je kritický uzel;
                  Pokračuj na úseku 6A, t.j. rotace RR nebo DRL " *);
        end; (* case *)
        (*****)
        end else begin
            (***** symetrická pravá verze *****)
            (* v této verzi se váha otce snižuje o 1 !! *)
            (*****)
        end; (*if*)
    end; (* while *)

```



Zde se provádějí rotace

```

(* 6A: *)
begin
    A:=KritUz;
    B:=A^.PUk;
    TOP(PomEl); POP;
    NadKrit:=PomEl.Uk;

    if B^.Vaha=-1
    then DelRotDRL(NadKrit,A)
    else DelRotRR(NadKrit,A);
    Konec:=true (* konec cyklu while *)
end;

(* 6B: *)
begin
    A:=KritUz;
    B:=A^.LUk;
    TOP(PomEl); POP;

```

```

NadKrit:=PomEl.Uk;

if B^.Vaha=1
then DelRotDLR(NadKrit,A)
else DelRotLL(NadKrit,A);
Konec:=true; (* konec cyklu while *)
end;

end; (* procedure DeleteAVL *)
begin
end.

```

Z hlediska praktického využití je AVL strom nejvýznamnější a nejpoužívanější stromová struktura používaná pro vyhledávání.

- Zaručuje logaritmickou složitost vyhledávání
- Zajišťuje dynamiku operace Delete (snadné rušení v tabulce)
- Umožňuje poměrně snadné udržování vyváženosti stromu a tím zabraňuje potenciální degradaci stromu jeho rozsáhlým porušením vyváženosti.

4.13 TRP

4.13 Tabulky s rozptýlenými položkami - TRP (hash table, hashing table) jsou založeny na použití pole, které je primárním prostorem pro práci s tímto typem tabulek.

Tabulka s přímým přístupem

Základem TRP je princip **tabulky s přímým přístupem**.

Definice:

DEF

Nechť existuje množina klíčů **K** dané tabulky a množina sousedních míst (adres) v paměti **H** které realizují vyhledávací tabulku. Existuje-li jedno-jednoznačná funkce mapující každý prvek první množiny do druhé množiny a naopak, pak hovoříme o tabulce s přímým přístupem. Této funkci se říká **mapovací funkce**.

x+y

Příklad: Kdyby množina klíčů byla dána intervalem 1..100 a tabulku reprezentovalo pole $T[1..100]$, pak mapovací funkce pro klíč K je $T[K]$. Každý prvek pole musí mít navíc Booleovskou složku „obsazeno/volno“, která určuje, zda daný prvek na své adrese je, či není. Při inicializaci se nastaví všechny prvky tabulky na „volno“.

Pak vyhledání spočívá v přímém zjištění, zda na pozici klíče (indexu) dané tabulky je obsazeno a pak tam prvek je nebo je volno a pak prvek v tabulce není.

Jinými slovy - každý prvek má své místo, to místo je přímo určitelné z klíče a na tomto místě prvek je, či není.

!

Časová složitost přístupu v přímé tabulce je 1.

Skutečnost, že se tabulky s přímým přístupem nepoužívají všeobecně spočívá v tom, že nalezení vhodné mapovací funkce je obtížné.

Mapovací funkce

Hledání vhodné **mapovací funkce** je obtížné. Uvádí se následující příklad: Pro 31 prvků, které se mají zobrazit do 41 prvkové množiny existuje 41^{31} tj cca 10^{50} různých možných

mapovacích funkcí. Přitom jen (41!/31!) z nich dává odlišné hodnoty pro různé klíče (jsou jedno-jednoznačné). Poměr „vhodných“ funkcí ku všem možným je tedy asi 1:10 000 000.

x+y

Situaci ilustruje příklad nazvaný "**Paradox společných narozenin**".

Je dobrá naděje, že mezi 23 osobami, které se sejdou ve společnosti, se najdou dvě osoby, které mají narozeniny ve stejný den.

Jinými slovy: Najdeme-li náhodně funkci, která mapuje 23 klíčů do tabulky o 365 prvcích, je pravděpodobnost, že se žádné dva klíče nenamapují do stejného místa rovna 0.4927. Naděje, že se do stejného místa namapují právě překročila hodnotu 0.5.

DEF



Jevu, kdy se dva různé klíče mapují do stejného místa říkáme **kolize**.
Dvěma nebo více klíčům, které se namapují do téhož místa říkáme **synonyma**.

Dobrá mapovací funkce musí splňovat dva požadavky:

- rychlost mapování
- vytváření co nejméně kolizí

Mapovací funkce transformuje klíč na index do primárního (rozptylového, hashovacího pole). Hodnota indexu musí být v daném intervalu rozsahu pole. Nejčastěji se mapovací funkce dělí do dvou etap:

- převod klíče na přirozené číslo ($N > 0$)
- převod přirozeného čísla na hodnotu spadající do intervalu indexu pole (nejčastěji s použitím operace modulo).

Nechť K je celé číslo větší než nula. Pak funkce

$$h(K) = K \bmod (Max+1)$$

získá hodnoty z intervalu 0..Max a funkce

$$h(K) = K \bmod Max + 1$$

získá hodnoty z intervalu 1..Max

Princip TRP

Princip tabulky s rozptýlenými položkami je obdobný principu **index-sekvenčního přístupu**.

Operace Insert:

Při vkládání nové položky se položka vloží na mapovací funkci stanovený index primárního pole (nebo na místo stanovené položkou na tomto indexu), pokud je toto místo volné. Pokud je místo obsazené, jde o vkládání synonymního klíče. Synonyma se vkládají do sekvenčně uspořádané struktury (seznamu), jejímž prvním prvkem je první z vložených synonym.

Operace Search:

Při vyhledávání začíná mechanismus na položce pole, ke které se dostane indexem určeným mapovací funkci. Je-li položka volná, znamená to "neúspěšné vyhledání" - položka neobsazená. Je-li položka obsazena, pak mechanismus začne prohledávat seznam synonym, začínající prvním prvkem. Pokud je hledaný prvek v seznamu synonym nalezen, je vyhledávání úspěšné, v opačném případě je neúspěšné.

Pozn. Mechanismus je "index-sekvenční", protože v prvním kroku "index" se dostává na začátek sekvence a ve druhém kroku "sekvenční" - se zahajuje sekvenční vyhledávání v posloupnosti synonym.

Explicitní a implicitní zřetězení synonym

- Při explicitním zřetězení obsahuje každý prvek posloupnosti adresu následníka.
- Při implicitním zřetězení se adresa následníka získá hodnotou funkce adresy předchůdce

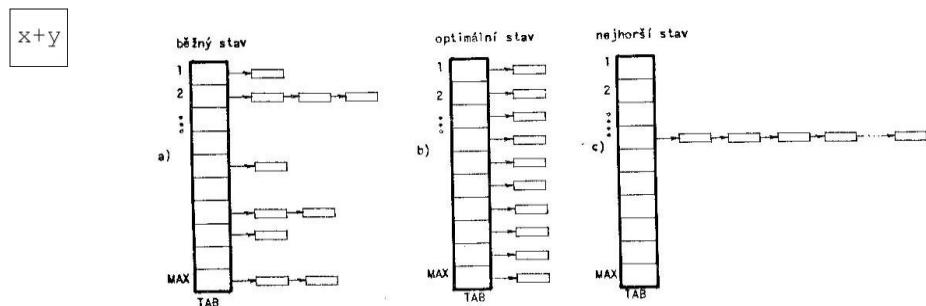
Tabulka s rozptýlenými položkami sestává z mapovacího prostoru (pole) a ze seznamů synonym. Každý seznam začíná na jednom prvku mapovacího pole.

TRP s explicitním zřetězením synonym

Pozn. I první prvek vytváří jednoprvkový seznam synonym

Maximální doba vyhledávání je dána délkou nejdelšího seznamu synonym. Rušení prvků je stejně jako ve zřetězeném seznamu. Seznamy prvků mohou být seřazeny podle klíče. To urychlí neúspěšné vyhledávání v seřazeném seznamu.

Situaci znázorňuje následující obrázek, zobrazující běžný stav, optimální stav a nejhorší stav, v něž je TRP degradována na lineární seznam.



Vytvořte operace Insert, Search a Delete pro vyhledávací tabulku implementovanou TRP s explicitním zřetězením. Klíč je typu integer. Rozptylové pole má 100 prvků.

- Rozptylové pole je pole ukazatelů na jednosměrné seznamy synonym.
- Rozptylové pole je pole ADT dvojsměrný seznam. Pro práci se seznamem použijte výhradně abstraktních operací nad ADT TDLList.

Klasifikace TRP

TRP lze klasifikovat na základě způsobu, jakým jsou vytvořeny seznamy synonym:

- TRP s explicitně zřetězenými synonymy
 - zřetězený seznam s využitím ukazatelů a DPP
 - zřetězený seznam v poli společném pro první rozptýlení i pro prvky synonymních prvků (tzv. Knuthova metoda).
- TRP s implicitně zřetězenými synonymy v poli
 - s konstantním krokem
 - krok určuje programátor
 - krok určuje program

- s proměnným krokem (kvadratická metoda)

Uvedený příklad a obrázek reprezentoval zřetězený seznam s využitím ukazatelů a DPP. Knuthova metoda je uvedena ve skriptech [1] na webové adrese:

<https://www.fit.vutbr.cz/study/courses/IAL/private/>

TRP s pevným krokem

TRP s implicitním zřetězením s pevným krokem je implementována v poli, ve kterém jsou jak první prvky seznamu synonym, tak jejich další položky. Na obrázku je popsána tabulka, kde dalším prvkem seznamu je adresa o 1 větší (pevný krok je 1), a kde

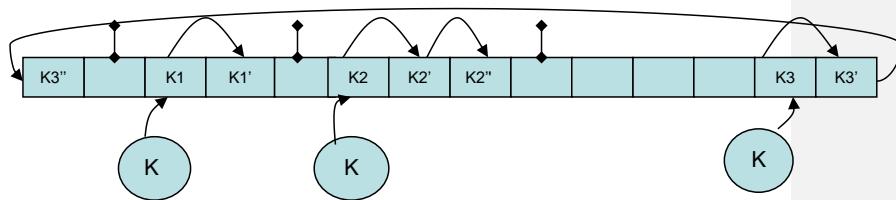
$$A_{i+1} = A_i + 1$$

"Seznam" synonym je tvořen po sobě jdoucími prvky a je ukončen první volnou položkou pole. Z toho vyplývají dvě zásady:

- S polem se pracuje jako s kruhovým seznamem!
- Položky pole musí být inicializovány tím, že se jejich indikátor obsazenosti nastaví na "volný"



x+y



Konec seznamu synonym je dán prvním volným prvkem, který se najde se zadaným krokem. Nové synonymum se uloží na první volné místo (na konec seznamu). Tabulka (pole) musí obsahovat alespoň jeden volný prvek. Efektivní kapacita je o 1 menší než je počet položek. Tabulka je implementovaná kruhovým polem.

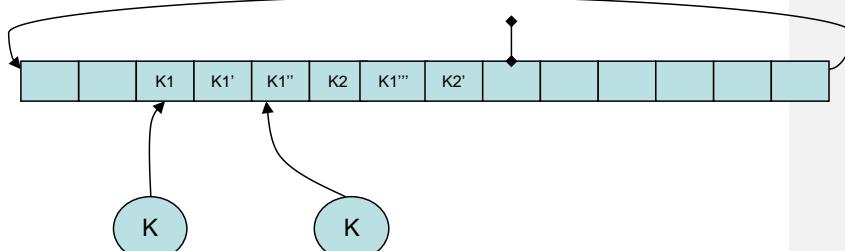
Příklad:

Nechť jsou již do tabulky vloženy klíče K1, K1' a K1''.

Následně se klíč K2 namapoval do položky, která je obsazena (je tam klíč K1''). Klíč byl uložen na první volné místo. Další klíč K1''' se namapoval do položky K1. První volné místo pro klíč K1''' bylo nalezeno za K2. Klíč K2' se namapoval do položky K2. První volné místo se nalezlo za klíčem K1'''.

V této tabulce se dva seznamy synonym překrývají. Prvek K1'' je vstupním bodem seznamu synonym K2. Nelze je zrušit ani zaslepením.

Situaci znázorňuje následující obrázek:

x+y

Velikost pole

Krok s hodnotou jedna má tendenci vytvářet „shluky“ (*cluster*). Výhodnější je krok větší než 1. Takový krok by ale měl mít možnost „navštívit“ všechny položky pole. Kdyby pole mělo sudý počet prvků, pak sudý krok (např. krok = 2) by z východiska lichého indexu mapovaného prvku mohl navštívit pouze liché indexy. Např. pole 1..10, krok 2, začátek na indexu 5 projde indexy: 5,7,9,1,3. Krok 4 a začátek na indexy 7 projde indexy: 7,1,5,9,3.

Kdyby měl krok hodnotu prvočísla, které je nesoudělné s jakoukoli velikostí pole, pak by mohl postupně projít všemi prvky pole. Výhodnější ale je, aby hodnotu prvočísla měla velikost mapovacího pole. Pak jakýkoli krok dovolí projít všemi prvky mapovacího pole.

Je vhodné dimenzovat velikost mapovacího pole TRP tak, aby bylo rovno prvočíslu.



Kvadratická metoda Mezi metody s automatickou změnou kroku patří tzv. „kvadratická metoda“. Hodnota kroku se v ní zvětšuje s každým krokem o 1. Při rozvinutí kruhového pole vytvářejí „navštívené“ adresy kvadratickou funkci. Kvadratická metoda má předpoklady nevytvářet shluky.

Ve skriptech [1] je podrobnější popis dvou variant kvadratické metody. Z hlediska použití jsou tyto metody bezvýznamné. Za pozornost stojí nápad, kterým v historické době zlepšovali programátoři nevyhovující vlastnosti TRP.

TRP s dvojí rozptylovací funkcí Metoda s dvojí rozptylovací (hashovací) funkcí patří mezi metody, v níž je krok v rozptylovacím poli určen programem – jeho druhou rozptylovací funkcí.

Nechť má rozptylovací pole rozsah 0..Max, (kde hodnota Max+1 je prvočíslo) a KInt je klíč transformovaný na celou hodnotu KInt>0, pak:

- První rozptylovací funkce vytváří hodnotu z intervalu 0..Max **ind= Kint mod (Max+1)**.
- Druhá rozptylovací funkce vytváří krok s hodnotou z intervalu 1..Max
krokind= Kint mod Max+1.

Search

Vyhledávací cyklus operace **Search** začíná na indexu získaném první rozptylovací funkcí. Končí úspěšně při nalezení prvku s hledaným klíčem, neúspěšně při dosažení konce seznamu synonym (prázdným prvkem). Index následujícího prvku je dán přičtením kroku k aktuálnímu prvku při respektování kruhovosti pole:

$$\text{ind}_{i+1} = (\text{ind}_i + \text{krok}) \bmod \text{Max}$$

Insert

Operace **Insert** vloží nový prvek na místo prvního prázdného prvku.

Delete

TRP s implicitním zřetězením se používají v aplikacích, v nichž se nepoužívá operace **Delete**.

Pozn. Pokud je přeci jen zapotřebí prvek rušit, je možné použít mechanismu "zaslepení".

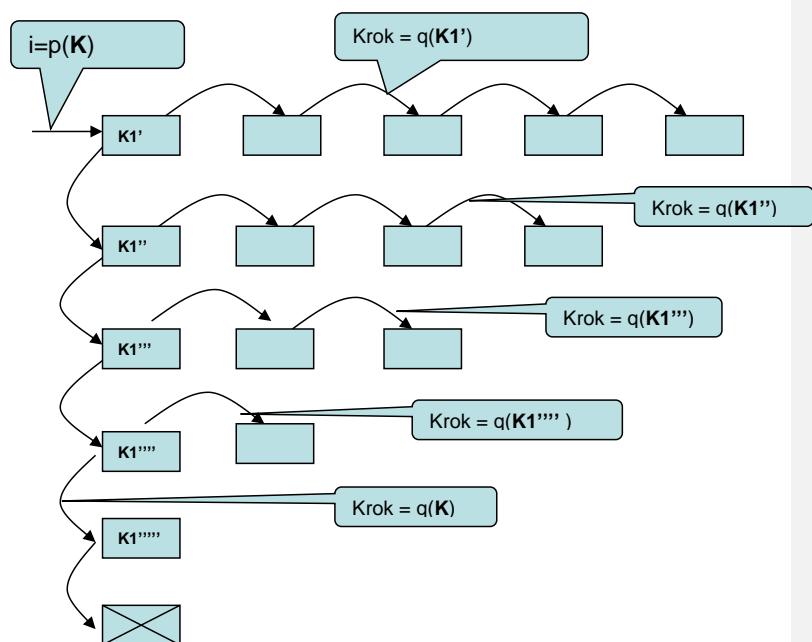
- Maximální kapacita TRP pro rozsah pole 0..Max je **Max** (o 1 menší než počet prvků). Pozn. Alespoň jeden prvek musí zůstat jako „zarážka“ vyhledávání.
- Pro efektivní použití TRP s implicitním zřetězením se pole tabulky dimenzuje tak, aby její maximální zaplnění, dané poměrem $N_{akt}/(\text{Max}+1)$, nebylo větší než cca 0.6-0.7.

Brentova varianta



Brentova varianta je varianta metody TRP se dvěma rozptylovacími funkcemi. Princip vyhledávání (Search) je shodný s TRP se dvěma rozpt. funkciemi.

Brentova varianta je vhodná za podmínky, že je počet případů úspěšného vyhledávání častější, než počet neúspěšného vyhledávání s následným vkládáním. Brentova varianta provádí při vkládání (operace Insert) dodatečnou rekonfiguraci prvků pole s cílem investovat do vkládání a získat tím lepší průměrnou dobu vyhledávání.

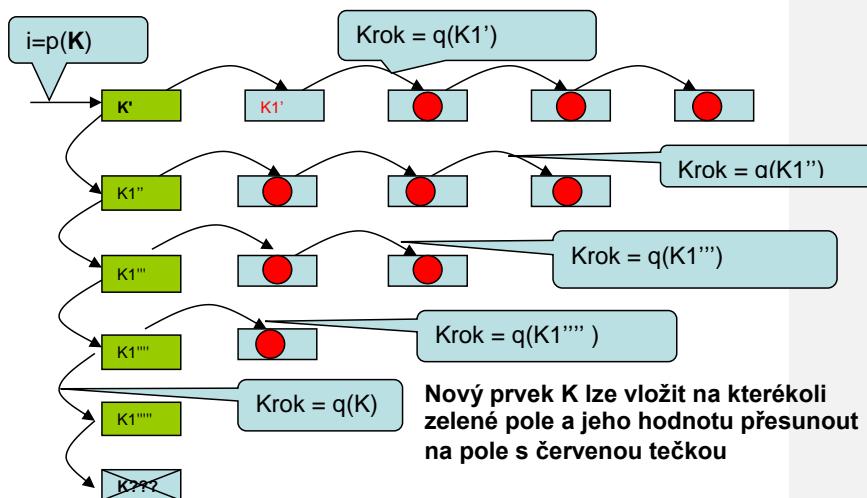


Prvek $K1'$ se přesune na první volné místo s krokem $q(K1')$ a na jeho místo se vloží K .

Normální metoda dvojího hashování by nový klíč vložila na první volné místo s krokem $q(k)$ – zde po 6 krocích.

Brentova varianta hledá první volné místo mezi červeně zakroužkovanými polí s krokem $q(K1')$, resp. $q(K1'')$ atd. Na toto místo vloží prvek $K1'$, resp. $K1''$ atd. a na uvolněné místo vloží prvek K .

Protože posun čelního prvku je menší než zde 5, je celková průměrná hodnota délky vyhledávání menší, než kdyby byl prvek K vložen na 6 pozici shora.



Hodnocení TRP

- Neexistuje obecné pravidlo jak nalézt nejvhodnější rozptylovací funkci. Dobrá rozptylovací funkce se může stanovit jen na základě znalosti vlastností množiny klíčů.
- Operaci Delete lze řešit pomocí „zaslepení“ – vložením klíče, který nebude nikdy vyhledáván.

Závěr

Čtvrtá kapitola představuje druhou stěžejní oblast předmětu. Prezentuje a vysvětuje nejvýznamnější algoritmy vyhledávání včetně jejich složitosti a stability. Znalosti této kapitoly patří k prerekvizitám řady následujících předmětů. Studenti se seznámi s rekurzivním i nerekurzivním způsobem zápisu vyhledávacích algoritmů tam, kde je to významné.

5 Řazení



Cíle kapitoly

Kapitolou "řazení" vrcholí téma předmětu Algoritmy. Řazení patří k nejzajímavějším algoritmům, jejich "dvojcyklovost" nutí ke zvýšené pozornosti při úvahách o složitosti. Algoritmy a hotové programy řazení najdeme vyřešené v řadě knihoven a publikací. Algoritmy však obsahují řadu "programovacích technik", obratů i principů, které jsou užitečné i při řešení jiných problémů než je řazení.

K přečtení kapitoly je zapotřebí asi 10 hodin. Čtenář, který nevyslechl přednášky na toto téma, si musí připočít cca 6 až 8 hodin.



5.1 Úvod

Terminologie



Pojmy jako "třídění" "řazení" nebo "setřídění" se v běžném životě zdají být blízké, ne-li identické. Definujme jejich sémantiku pro účely algoritmizace takto:

Třídění

Třídění (*sorting*) je rozdělování položek homogenní datové struktury do skupin (tříd) se (zadanými) shodnými vlastnostmi (atributy).

Řazení

Řazení (*ordering*) je uspořádání položek dané lineární homogenní datové struktury do sekvence podle relace uspořádání nad zadánou vlastností (klíčem) položek.

Setřídění

Setřídění (*merging*) je sloučení dvou nebo více seřazených lineárních homogenních datových struktur do jedné seřazené lineární homogenní datové struktury.

Pozn. Termín "třídění" v oblasti zpracování údajů vznikl v předpočítákové éře, kdy se mechanizované řazení na děrných štítcích provádělo postupným tříděním na mechanických třídicích strojích. Tyto stroje roztrídily (rozdělily) soubor štítků na 10 podsouborů podle hodnoty 0-9 vyděrované v zadáném sloupci. Operátor seřadil ručně tyto podsoubory za sebe do jednoho souboru a třídal je podle dalšího sloupu. Třídal-li se takto soubor štítků postupně podle sloupce 10,9,8,7,6 byl nakonec soubor seřazen podle hodnot vyděrovaných ve sloupcích 6-10. Tento princip zachovává metodu "radix-sort" ("řazení tříděním"). Algoritmy používané později pro řazení na počítačích nevyužívají princip třídění (*sorting*), ale pojmenování "sort" - "třídění" jim v názvu zůstalo. V anglických názvech se kmen "sort" zachovává a v anglické terminologii se s pojmem "sorting" pro řazení budeme setkávat. V české terminologii se přidržíme termínu "řazení". Ostatně, ani v tělocviku neříkáme "setříďte se podle velikosti..." a pokud něco třídíme, tak např. spíše ovoce podle druhu nebo velikosti nebo auta podle barvy. Řazení pro nás zůstane zvláštním případem třídění.

Sekvenční / nesekvenční algoritmus

Sekvenční řadicí algoritmus přistupuje k řazeným položkám struktury sekvenčním způsobem (k jednomu po druhém). **Nesekvenční** algoritmus umožňuje náhodný přístup k položkám řazené struktury.

Stabilita

Stabilita řazení je vlastnost algoritmu, který zachová relativní pořadí položek se stejnou hodnotou klíče.

Příklad. Sekvence položek 4, 2, 5', 3, 5", 8, 6, 5'', 9, 1, kde čárky označují relativní pořadí klíčů s hodnotou 5 bude mít při zachování stability řazení podobu: 1,2,3,5', 5", 5'', 6, 8, 9. Nestabilní metoda nemusí respektovat pořadí položek se shodnými klíči.



5 ŘAZENÍ

Kapitolou "řazení" vrcholí téma předmětu Algoritmy. Řazení patří k nejzajímavějším algoritmům, jejich "dvojcyklovost" nutí ke zvýšené pozornosti při úvahách o složitosti. Algoritmy a hotové programy řazení najdeme vyřešené v řadě knihoven a publikací. Algoritmy však obsahují řadu "programovacích technik", obratů i principů, které jsou užitečné i při řešení jiných problémů než je řazení.

K přečtení kapitoly je zapotřebí asi 10 hodin. Čtenář, který nevyslechl přednášky na toto téma, si musí připočít cca 6 až 8 hodin.

5.1 Úvod

Terminologie



Pojmy jako "třídění" "řazení" nebo "setřídění" se v běžném životě zdají být blízké, ne-li identické. Definujme jejich sémantiku pro účely algoritmizace takto:

Třídění (*sorting*) je rozdělování položek homogenní datové struktury do skupin (tříd) se (zadanými) shodnými vlastnostmi (atributy).

Řazení (*ordering*) je uspořádání položek dané lineární homogenní datové struktury do sekvence podle relace uspořádání nad zadánou vlastností (klíčem) položek.

Setřídění (*merging*) je sloučení dvou nebo více seřazených lineárních homogenních datových struktur do jedné seřazené lineární homogenní datové struktury.

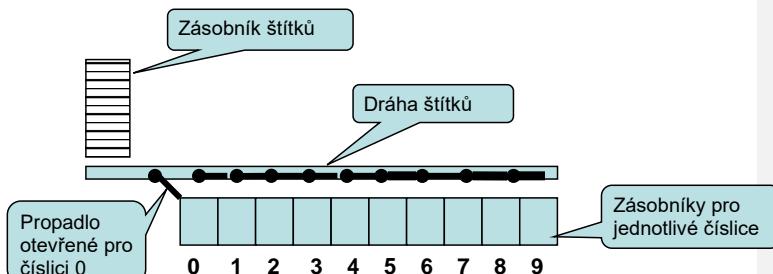
Pozn. Termín "třídění" v oblasti zpracování údajů vznikl v předpočítákové éře, kdy se mechanizované řazení na děrných štítcích provádělo postupným tříděním na mechanických třídicích strojích. Tyto stroje roztrídily (rozdělily) soubor štítků na 10 podsouborů podle hodnoty 0-9 vyděrované v zadáném sloupci. Operátor seřadil ručně tyto podsoubory za sebe do jednoho souboru a třídal je podle dalšího sloupu. Třídal-li se takto soubor štítků postupně podle sloupce 10,9,8,7,6 byl nakonec soubor seřazen podle hodnot vyděrovaných ve sloupcích 6-10. Tento princip zachovává metodu "radix-sort" ("řazení tříděním"). Algoritmy používané později pro řazení na počítačích nevyužívají princip třídění (*sorting*), ale pojmenování "sort" - "třídění" jim v názvu zůstalo. V anglických názvech se kmen "sort" zachovává a v anglické terminologii se s pojmem "sorting" pro řazení budeme setkávat. V české terminologii se přidržíme termínu "řazení". Ostatně, ani v tělocviku neříkáme "setříďte se podle velikosti..." a pokud něco třídíme, tak např. spíše ovoce podle druhu nebo velikosti nebo auta podle barvy. Řazení pro nás zůstane zvláštním případem třídění.

Sekvenční řadicí algoritmus přistupuje k řazeným položkám struktury sekvenčním způsobem (k jednomu po druhém). **Nesekvenční** algoritmus umožňuje náhodný přístup k položkám řazené struktury.

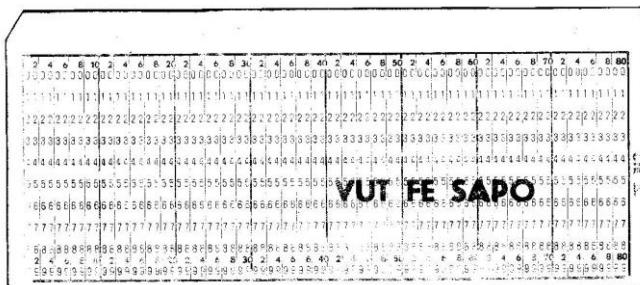
Stabilita řazení je vlastnost algoritmu, který zachová relativní pořadí položek se stejnou hodnotou klíče.

Příklad. Sekvence položek 4, 2, 5', 3, 5", 8, 6, 5'', 9, 1, kde čárky označují relativní pořadí klíčů s hodnotou 5 bude mít při zachování stability řazení podobu: 1,2,3,5', 5", 5'', 6, 8, 9. Nestabilní metoda nemusí respektovat pořadí položek se shodnými klíči.

Přirozenost	Přirozenost řazení je vlastnost algoritmu řazení, jehož <i>doba řazení sekvence</i> již uspořádané vzestupně podle hodnoty daného klíče řazena je <i>menší, než doba řazení náhodně uspořádané sekvence a ta je menší, než doba řazení sekvence již seřazené v opačném pořadí hodnoty klíčů.</i>
Smysl řazení	Jako implicitní budeme považovat seřazení podle vzestupného uspořádání klíčů (od nejmenšího k největšímu).
Experimentální hodnoty	V této kapitole jsou u některých metod uvedeny experimentální hodnoty , získané v diplomní práci studenta našeho obooru. Jejich význam slouží jen k relativnímu porovnání metod.
Třídicí stroj	Třídicí stroj firmy Hollerith (pozdější IBM) byl použit při sčítání lidu v USA v r. 1890. Vyděrovaný otvor ve sloupci štítků, reprezentující číslici, způsobil mechanické otevření propadla nad odpovídajícím zásobníkem. Na následujících obrázcích je schematické znázornění třídicího stroje a ukázka děrného štítku používaného na našem ústavu před 20 lety.



Děrný štítek



5.1.1. Řazení podle více klíčů 5.1.1. V praxi je **řazení podle více klíčů** velmi časté. Jako příklad lze uvézt:

- Řazení podle data narození, kde datum sestává ze tří číselných klíčů: rok, měsíc a den.
- Řazení studentů podle čtyř klíčů: obor, ročník, studijní průměr a jméno. Úkolem je např. vytvořit seznam po oborech, v oboru po ročnících, v ročníku podle studijního průměru a studenty se stejným průměrem seřadit abecedně podle jména.

Složená relace Problém leží řešit třemi způsoby:
a) Vytvoření složené relace uspořádání:

```

function PrvniStarsi(Prv, Druh:TDatNar):Boolean;
(* false znamená, že druhý je starší, nebo jsou oba stejně starí *)
begin
  if Prv.Rok<>Druh.Rok
  then
    PrvniStarsi:= Prv.Rok<Druh.Rok
  else
    if Prv.Mes<> Druh.Mes (* rok je shodný *)
    then PrvniStarsi := Prv.Mes<Druh.Mes
    else PrvniStarsi:= Prv.Den<Druh.Den (* měsíc je
shodný *)
  end;

```

Postupné řazení podle jednoho klíče

- b) Neuspořádanou množinu položek lze řadit postupně podle vztahující priority jednotlivých klíčů. Podmínkou je použití stabilní řadicí metody! Příklad: Skupinu osob lze seřadit podle stáří tak, že se:
 1. Napřed seřadí podle dne data narození
 2. Pak se seřadí podle měsíce data narození
 3. Nakonec se seřadí podle roku data narození
Tento způsob se podobá řazení děrných štítků v Hollerithově metodě.

- Aglomerovaný klíč**
- c) V praxi se často používá metoda „**aglomerovaného klíče**“. Uspořádaná N-tice klíčů se konvertuje na vhodný typ, nad nímž je definována relace uspořádání. Vhodným typem je typ string. Ukázkou aglomerovaného klíče je např. rodné číslo. Lze ho pro řazení použít bez úpravy jako řetězec jen pro stejné pohlaví. Má tvar: RRMMDDXXXX, ale ženy mají MM zvýšené o 50 (žena narozená v dubnu má r.č. např: 8454015471).

Příklady k procvičení



- a) Napište proceduru, která ze zadaného pole osob vytvoří seřazený seznam podle narozenin v roce. Při shodném datu narozenin má starší přednost.

```

type
  TDatNar = record
    Rok, Mes, Den:integer
  end;
  TOsoba = record
    Jmeno:string;
    DatNar:TDatNar
  end;

```

Pozn. Seznam osob seřazený "podle narozenin" vytvoří seznam, v němž jsou postupně osoby tak, jak slaví v běžném roce narozeniny. Má-li narozeniny více osob v téže dni, mají starší osoby "přednost".

- c) Je dán typ

```

type
  TObor= (infsys,intsys,pocsys,grasys);
  TStudent=record
    jmeno:string;
    obor:TObor;
    rocnik:integer;

```

```
    prumer:real
  end;
```

Vytvořte aglomerovaný (integrovany) klíč pro vytvoření seznamů:

- I. Podle oboru, v oboru podle ročníku, v ročníku podle průměru v průměru podle jména
- II. Podle průměru, v průměry oboru, v oboru podle ročníku, v ročníku podle jména.

Návod: Aglomerovaný klíč bude typu string. Průměr lze převést na typ integer např: 2.75 \Rightarrow 275. String omezte na 20 znaků.

- b) Napište proceduru libovolného algoritmu řazení pole, který znáte z prvního ročníku tak, aby se při volání procedury jedním vhodným parametrem ovládala složka, která bude klíčem řazení. Nechť pole je pole prvků typu TOsoba. Pak procedura:

```
procedure Razeni (var Pole:TPole, XX:TXX);
bude řadit jednou podle složky Rok, jindy podle složky Mes a jindy podle
složky Den, v závislosti na parametru XX. Nalezněte pro tento účel vhodný typ
a deklarujte ho. Trojí volání této procedury pokaždé podle jiné složky může
vytvořit seznam podle stáří nebo seznam podle narozenin.
```

d) Napište funkci

```
function PrvniStarci (RC1,RC2:string):Boolean;
```

e) Napište funkci

```
function MaDridNarozeniny (RC1,RC2:string):Boolean;
```

kde RC1 a RC2 jsou rodňá čísla. V případě stejně starých osob nebo stejných narozenin má přednost žena před mužem. V případě rovnosti u stejněho pohlaví rozhoduje pořadové číslo rodňá čísla XXXX.

5.1.1. Řazení bez přesunu položek

5.1.1. Řazení bez přesunu položek. Nejčastěji prováděnými operacemi v algoritmech řazení jsou přesuny položek v poli a porovnávací operace. V případě „dlouhých“ položek jsou přesuny časově velmi náročné. Především tuto situaci, ale i některé jiné situace řeší řazení polí bez přesunu položek.

K řazenému poli se vytvoří pomocné pole pořadí (tzv. pořadník) PolPor. Inicializuje se hodnotami shodnými s indexem. Výsledkem řazení je pořadník, v němž jsou uspořádány (seřazeny) indexy prvků řazeného pole.

Nečtějte jsou dány typy:

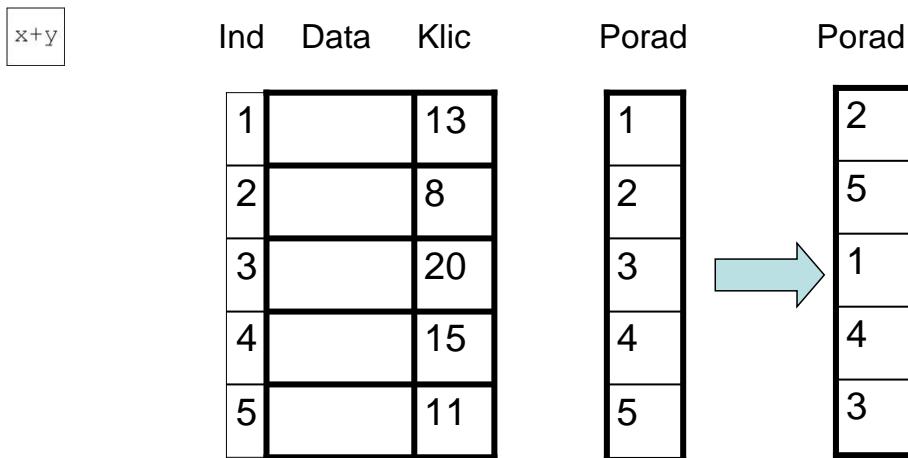
```
type
  TPoložka=record
    Data:TData;
    Klic:TKlic
  end;
  TPole=array[1..Max] of TPoložka;
  TPorad=array[1..Max] of integer;
```

```
var
  Pole:TPole;
  Porad:TPorad;
```

Pak inicializace má tvar:

```
for i:=1 to Max do Porad[i]:= i;
```

Následující obrázek znázorňuje stav pořadníku po inicializaci a po seřazení.



Každá **relace** mezi dvěma prvky pole v algoritmu řazení s přesunem se v odpovídajícím algoritmu pro řazení bez přesunu transformuje tímto způsobem:

```
Pole[i].Klic > Pole[j].Klic
```

se zapíše formou:

```
Pole[Porad[i]].Klic > Pole[Porad[j]].Klic
```

Každá **výměna** dvou prvků i a j pole v algoritmu řazení s přesunem se v zápisu algoritmu řazení bez přesunu transformuje takto:

```
Pole[i] :=: Pole[j]
```

se zapíše formou:

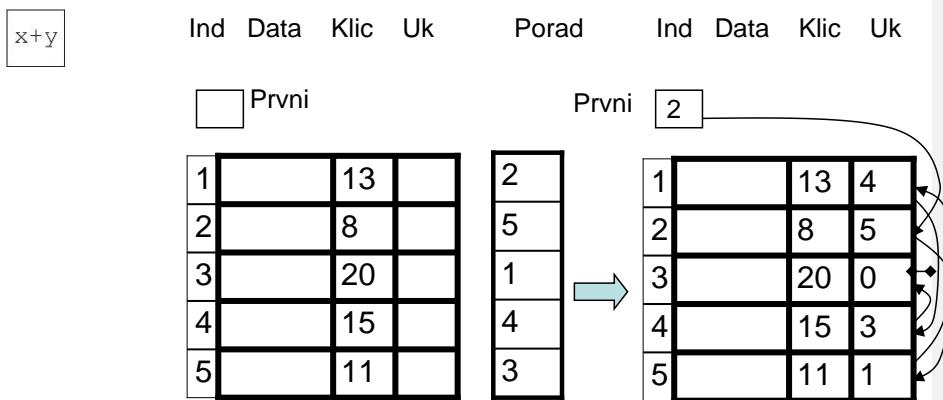
```
Porad[i] :=: Porad[j]
```

Pole seřazené bez výměny položek lze průchodem vložit do výstupního seřazeneho pole VystPole cyklem:

```
for i:=1 to Max do VystPole[i]:= Pole[Porad[i]]
```

Pole seřazené pomocí „pořadníku“ lze také zřetězit a vytvořit seřazeny seznam.

Následující obrázek znázorňuje výsledek zřetězení pole seřazeného pomocí pořadníku z předcházejícího příkladu.



Zřetězení provede úsek programu:

```
Prvni:=Porad[1];
for i:=1 to Max-1 do
  Pole[Porad[i]].Uk:=Porad[i+1];
Pole[Porad[MAX]].Uk:=0; (* 0 ve funkci nilu*)
```

Zřetězenou seřazenou posloupnost lze převést ze zdrojového pole do cílového pomocí jednoduchého cyklu, který je vhodným příkladem pro domácí prověření.

MacLarenův algoritmus



MacLarenův algoritmus uspořádá pole seřazené bez přesunu na místo samém (lat. *in situ*) - tedy proces bez pomocného pole.

```
i:=1; Pom:=Prvni;
while i<Max do begin
  (* Hledání následníka přesunutého na pozici větší než i *)
  while Pom<i do Pom:=Pole[Pom].Uk;
  (* výměna akt. prvního s akt. minimálním *)
  Pole[i] :=: Pole[Pom];
  Pole[i].Uk :=: Pom; (* stejná výměna ukazatelů*)
  i:=i+1 (* prvních i-1 prvků je již na svém místě *)
end;
```



Pozn. Komentář k MacLarenovu algoritmu.

První prvek seznamu (na který ukazuje proměnná Prvni) se vymění s prvkem pole na indexu 1. Tím se nejmenší položka dostane na své místo. Na prvek, který byl z prvního indexu pole odsunut jinam však některý prvek ukazoval. Je třeba ho najít a změnit jeho ukazatel tak, aby místo na první index ukazoval na místo, kam byl první odsunut.

Tím je první prvek ošetřen. Dalším „prvním“ se stane index o jednu větším a cyklus pokračuje tak dlouho, až se vymění předposlední (Max-1) prvek, kdy cyklus končí.

S ohledem na velkou délku položky je součet času řadícího algoritmu bez přesunu položek (minimálně linearitický) s časem McLarenova algoritmu (lineární) kratší, než čas samotného řadícího algoritmu s přesunem položek.

Klasifikace metod řazení

Klasifikace metod řazení lze provést podle přístupu k paměti a podle typu použitého procesoru:

- Podle přístupu k paměti:
 - Metody vnitřního řazení (metody řazení polí). Přímý (náhodný) přístup.
 - Metody vnějšího řazení - sekvenční přístup. Řazení souborů a řazení seznamů.
- Podle typu procesoru:
 - sériové (jeden procesor) – jedna operace v daném okamžiku
 - paralelní (více procesorů) – více souběžných operací.

Podle principu řazení lze metody dále členit:

- Princíp výběru (*selection*) - přesouvají maximum/minimum do výstupní posloupnosti
- Princíp vkládání (*insertion*) – vkládají postupně prvky do seřazené výst. posloupnosti
- Princíp rozdělování (*partition*) – rozdělují postupně množinu prvků na dvě podmnožiny tak, že prvky jedné jsou menší než prvky druhé
- Princíp slučování (*merging*) setřídí už postupně seřazené dvě podmnožiny do jedné
- Jiné principy ...

Zavedené konvence

V následujících partiích budou metody řazení v polích vykládány na silně zjednodušené struktury množiny dat, implementované polem s jednosložkovými položkami, představovanými klíčem typu integer:

```
type
  TA=array[1..N] of integer; (* Typ řazeného pole *)
var
  A:TA; (* Řazené pole *)
```

Toto pole bude vstup/výstupním parametrem (typu var) procedury řazení nebo jejím globálním objektem.

5.2 Řazení na principu

5.2 Řazení na principu výběru (*Select sort*)

výběru

Řazení na principu výběru je nejjednodušší a asi nepřirozenější způsob řazení. Jejím jádrem je cyklus pro vyhledání pozice extrémního (minimálního resp maximálního) prvku v zadaném segmentu pole. Princip lze popsat takto:

```
i:=1;
for i:=i to N do begin
    (* najdi minimální prvek pole mezi indexy i a n. Polohu (index) minima
    ulož do pomocné proměnné PInd *)
    A[i]:=A[PInd];
end
```

x+y Algoritmus má tvar:

```
procedure SelectSort(var A:TA);
var i,j,PInd, PMin:integer;
begin
    for i:=1 to N-1 do begin
        PInd:=i;      (* Poloha pomocného minima*)
        PMin:=A[i];  (* Pomocné minimum*)
        for j:=i+1 to N do
            if PMin>A[j]
            then begin
                PMin:=A[j];
                PInd:=j
            end;
            A[i] :=: A[PInd]
        end (* for *)
    end;
```

Hodnocení metody**Hodnocení metody:**

- Metoda je nestabilní. (Vyměněný první prvek se může dostat „za“ prvek se shodnou hodnotou).
- Má kvadratickou časovou složitost
- Experimentálně byly naměřeny výsledky:

(kde OSP je opačně seřazené pole a NUP je náhodně uspořádané pole, N je počet prvků.)

Experimentálně zjištěné hodnoty.

N	128	256	512
OSP	64	254	968
NUP	50	212	774

5.2.1. Bublinové řazení

5.2.1. Bublinové řazení na principu výběru (*Bubble-sort*). Princip bublinového výběru je shodný se základní metodou výběru, ale metoda nalezení extrému je jiná. Porovnává se každá dvojice sousedních prvků a v případě obráceného uspořádání se mezi sebou vymění. Při pohybu zleva

doprava se tak maximum dostane na poslední pozici, zatímco minimum se posune o jedno místo směrem ke své konečné pozici.

```
x+y
procedure BubbleSort(var A:TA);
var
    i,j: integer;
    Konec:Boolean;
begin
    i:=2;
    repeat
        Konec:=true;
        for j:=N downto i do (*porovnávání sousedních dvojic *)
            if A[j-1] > A[j]
            then begin
                A[j-1]:= A[j]; (*výměna*)
                Konec:=false
            end;
            i:=i+1
        until Konec or (i=(N+1));
end;
```

Jiná varianta zápisu algoritmu Bubble sort.

```
x+y
procedure BubbleSelect (var A:TA); (* Jiná varianta *)
var
    i, PomN:integer;
    Pokracuj:Boolean;
begin
    PomN:=N; Pokracuj:=true;
    while Pokracuj and (PomN>1) do begin
        Pokracuj:=false;
        for i:=1 to PomN - 1 do (* cyklus porovnávání dvojic *)
            if A[i+1] < A[i]
            then begin
                A[i+1] := A[i];
                Pokracuj:=true (* došlo k výměně – nelze skončit *)
            end; (*if*)
        PomN:=PomN-1;
    end;
end;
```

Hodnocení metody:

- Bublínový výběr je metoda **stabilní a chová se přirozeně**. Je to jedna z mála metod použitelná pro vícenásobné řazení podle více klíčů!
- Metoda má **kvadratickou** časovou složitost.
- Je to nejpopulárnější a **nejméně efektivní** metoda. **Je to nejrychlejší metoda v případě, že pole je již seřazeno!**
- Experimentálně naměřené hodnoty:

n	256	512
NUP	338	1562

OSP	558	2224
-----	-----	------

Od Bubble-sortu byla odvozena řada variant se snahou zlepšit efektivnost metody. Některé přináší zajímavé programovací náměty, ale z pohledu zlepšení nemají výrazný účinek.

- **Ripple-sort** si pamatuje polohu první vyměny a je-li větší než 1 neprochází dvojicemi u nichž je jasné, že se nebudou vyměňovat.
- **Shaker-sort** střídá směr probublávání zleva a zprava (používá „houpačkovou metodu“) a skončí uprostřed.
- **Shuttle-sort** „zavede“ při výměně dvojice menší prvek zpět „na své“ místo a pak pokračuje dál. Končí tím, že nevymění nejpravější dvojici.

5.2.2. Heap sort

5.2.2. Heap sort - "řazení hromadou".

Hromada (*heap*) je struktura stromového typu, u níž pro všechny uzly platí, že mezi otcovským uzlem a všemi jeho synovskými uzly je stejná relace uspořádání (např. otec je větší než všichni synové). Nejčastější případ hromady – heape je binární hromada, založená na binárním stromu.

Významnou operaci nad hromadou je její **rekonstrukce** poté, co se poruší pravidlo hromady v jednom uzlu.

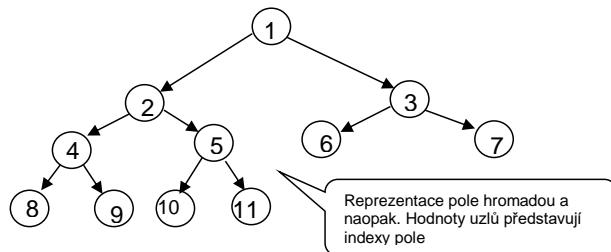
Nejvýznamnějším případem je **porušení hromady v kořeni**. Operaci, která znovuustaví hromadu porušenou v kořeni říkáme „**sift**“ (prosetí), nebo také „zatřesení hromadou“. Spocívá v tom, že prvek z kořene postupnými výměnami propadne na „své“ místo a do kořene se dostane prvek, splňující pravidla hromady. Tato operace má v nejhorším případě složitost $\log_2 n$. Jinými slovy, když hromada má v kořeni nejmenší ze všech prvků, pak když se postupně odebrá z kořene prvek, vkládá se do výstupního pole, po každém odebrání se do kořene vloží hodnota nejnižšího a nejpravějšího uzlu a poté se hromadou „zatřese“, získá se s linearitickou složitostí $n \cdot \log_2 n$ **seřazená posloupnost**.

Podstatou řadicí metody je implementace hromady polem. Je to implementace s **implicitním zřetězením** prvků binární stromové struktury hromady.

Binární strom o n úrovních (a také hromadu), který má všechny uzly na všech $(n-1)$ úrovních a na nejvzdálenější n -té úrovni má všechny uzly zleva, lze snadno implementovat polem. Pak platí pro otcovský a synovské uzly vztah: když je otcovský uzel na indexu i , pak je levý syn na indexu $2i$ a pravý syn na indexu $2i+1$.

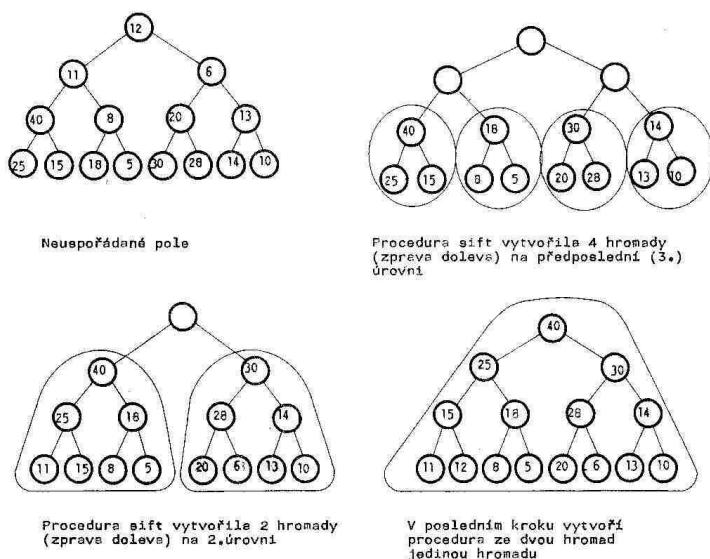
Reprezentace pole hromadou

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



Mějme proceduru SiftDown, která znova ustaví hromadu porušenou v kořeni - proseje prvek v koření, který jako jediný poruší pravidlo hromady. Následující obrázky znázorňují vytvoření hromady postupnou aplikací procedury SiftDown na uzly počínaje nejnižším a nejpravějším otcovským uzlem hromady. V kořeni hromady je vždy **maximální hodnota**. Na počátku jde o strom o (dvou) třech uzlech, kde prosetí vytvoří prvotní hromady, dále prosetím spojované po dvou ve větší celky.

Znázornění tvorby hromady



Pak hromadu lze ustavit tak, že se ustaví „porušená“ hromada, jejíž kořen je nejpravější a nejnižší „otecovský uzel“ a postupuje se ustavováním hromad po všech uzlech dole a nahoru až k hlavnímu kořeni, jak to ilustroval předcházející obrázek. Má-li pole N prvků, pak nejnižší a nejpravější otcovský uzel odpovídající hromadě má index $[N \text{ div } 2]$.

```
(* Ustavení hromady *)
Left:= N div 2; (* index nejpravějšího uzlu na nejnižší úrovni *)
Right:=N;
for i:= Left downto 1 do SiftDown(A,i,Right);
(* cyklus opakovaného prosetí "porušeného" kořene do dvou podstromů, které
již mají vlastnosti hromady *)
```

Celý algoritmus Heapsortu má pak tvar:

```
procedure HeapSort(var A:TA);
var
  i,Left,Right:integer;
begin (* Ustavení hromady *)
  Left:= N div 2; (* index nejpravějšího uzlu na nejnižší úrovni *)
  Right:=N;
  for i:= Left downto 1 do SiftDown(A,i,Right);
  (* Vlastní cyklus Heap-sortu *)
  for Right:=N downto 2 do begin
    A[1]:=A[Right]; (* Výměna kořene s akt. posledním prvkem *)
    SiftDown(A,1,Right-1) (* Znovuustavení hromady *)
  end; (* for *)
end; (* procedure *)
```

Sift-down, rekonfigurace heapu Při implementaci binárního stromu polem je důležité poznat konec větve (terminální uzel, nebo uzel který nemá pravého syna):

- Je-li dvojnásobek indexu uzlu větší než počet prvků pole N, pak odpovídající uzel je terminální.
 - Je-li dvojnásobek indexu uzlu roven počtu prvků N, má odpovídající uzel pouze levého syna.
 - Je-li dvojnásobek indexu uzlu menší než počet prvků N, má odpovídající uzel oba syny.
 -
- ```
procedure SiftDown(var A:TArr;Left,Right:integer);
(* Left je kořenový uzel porušující pravidla heapu, Right je velikost pole *)
var
 i,j:integer;
 Cont:Boolean; (* Řídicí proměnná cyklu *)
 Temp:integer; (* Pomocná proměnná téhož typu jako položka pole *)
begin
 i:=Left;
 j:=2*i; (* Index levého syna *)
 Temp:=A[i];
 Cont:=j<=Right;
 while Cont do begin
 if j<Right
 then (* Uzel má oba synovské uzly *)
 if A[j]< A[j+1]
 then (* Pravý syn je větší *)
 j:=j+1; (* nastav jako většího z dvojice synů *)
```

```

if Temp >= A[j]
then (* Prvek Temp již byl posunut na své místo; cyklus končí *)
 Cont:=false
else begin (* Temp propadá níž, A[j] vyplouvá o úroveň výš *)
 A[i]:=A[j]; (* *)
 i:=j; (* syn se stane otcem pro příští cyklus"*)
 j:=2*i; (* příští levý syn *)
 Cont:=j<=Right; (* podmínka : "cyklus pokračuje" *)
end (* if *)
end; (* while *)
A[i]:=Temp; (* konečné umístění prosetého uzlu *)
end; (* procedure *)

```

**Závěr**

- Heapsort je řadící metoda s **linearitickou složitostí**, protože „sift“ umí rekonstruovat hromadu (najít extrém mezi N prvků) s logaritmickou složitostí. Tato vlastnost může být významná i pro jiné případy.
- Heapsort je nestabilní (přesouvá prvky s velkými skoky) a nechová se přirozeně.
- Hromada (heap) je užitečná struktura. Je vhodná tam, kde je opakován zapotřebí hledat extrém (minimum, maximum) na téže množině prvků. Umožňuje to s logaritmickou rychlostí tam, kde by se to jinak dělalo s lineární složitostí.
- Naměřené hodnoty:

|     |     |      |
|-----|-----|------|
| N   | 256 | 1024 |
| SP  | 42  | 210  |
| NUP | 38  | 186  |
| OSP | 40  | 196  |

**5.3 Řazení vkládáním**

**5.3 Řazení na principu vkládání (insert-sort)** se podobá mechanismu, kterým hráč karet bere po rozdání postupně karty ze stolu a vkládá je do uspořádaného vějíře v ruce. Stejně se pracovalo s tradiční "kartotékou".

Nechť je pole rozděleno na dvě části: levou – seřazenou a pravou neseřazenou. Na začátku procesu tvoří levou první prvek. Pak má řazení strukturu:

```

for i:=2 to N do begin
 (* najdi v levé části index K, na který se má zařadit prvek A[i]*);
 (* posuň část pole od K do i-1 o jednu pozici doprava *);
 (* vlož na A[k] hodnotu zařazovaného prvku *)
end

```

Metoda je z principu potenciálně stabilní.

**Bublinové vkládání**

**Metoda bublinového vkládání (Bubble-insert sort).** Tato metoda slučuje vyhledání místa pro vložení a posun segmentu pole do jednoho cyklu postupným porovnáváním a výměnou dvojic prvků. Tím se podobá

stejnojmenné metodě výběru.

```
procedure BubbleInsertSort(var A:TArray);
var
 i,j:integer;
 Tmp:integer;
begin
 for i:=2 to N do begin
 Tmp:=A[i];
 A[0]:=Tmp; (* Ustavení zarážky *)
 j:=i-1;
 while Tmp<A[j] do begin (* hledej místo a posouvej prvek *)
 A[j+1]:=A[j];
 j:=j-1
 end; (* while *)
 A[j+1]:=Tmp; (* konečné vložení na místo *)
 end; (* for *)
end; (* procedure *)
```

#### Závěr

Metoda je stabilní, chová se přirozeně a pracuje in situ.(Je vhodná pro vícenásobné řazení podle více klíčů).

Metoda má kvadratickou časovou složitost.

Experimentálně byly naměřeny tyto hodnoty:

| n   | 256 | 512  | 1024 |
|-----|-----|------|------|
| SP  | 4   | 6    | 14   |
| NUP | 156 | 614  | 2330 |
| OSP | 312 | 1262 | 5008 |

#### Vkládání s binárním vyhledáváním

**Vkládání s binárním vyhledáváním** nahrazuje lineární vyhledávání rychlejším - binárním vyhledáváním. V případě více stejných klíčů si metoda zachová stabilitu tehdy, když vyhledá pozici za nejpravějším ze shodných klíčů.

```
procedure BinaryInsertSort(var A:TArr;N:integer);
(* N je počet prvků tabulky *)
var
 i,j,m,Left,Right,Insert:integer;
 Tmp:integer;
begin
 for i:=2 to N do begin
 Tmp:=A[i];
 Left:=1; Right:=i-1; (* nastavení levého a pravého indexu *)
 while Left <=Right do begin (* stand. bin. vyhledávání *)
 m:=(Left+Right)div 2;
```

```

if Tmp<A[m]
then Right:=m-1
else Left:= m+1;
end;
for j:=i-1 downto Left do
 A[j+1]:=A[j]; (* posun segmentu pole doprava *)
 A[Left]:=Temp;
end; (* for *)
end; (* procedure *)

```

**Závěr** Metoda je stabilní, chová se přirozeně a pracuje in situ.

Binární vyhledávání snížilo počet porovnání z lineární hodnoty na logaritmickou. Počet přesunů, které jsou časově obvykle mnohem náročnější, než porovnání se však nezměnil. Metoda proto nepřinesla velké zlepšení.

Tabulka naměřených hodnot:

| N   | 256 | 512 | 1024 |
|-----|-----|-----|------|
| NUP | 134 | 502 | 1024 |
| OSP | 248 | 956 | 3736 |

### Kontrolní otázky a úlohy

Je dána hromada o N prvcích typu integer v poli H. Napište proceduru, která rekonstruuje (znovuustaví) hromadu porušenou zápisem libovolné hodnoty na index  $1 \leq K \leq N$

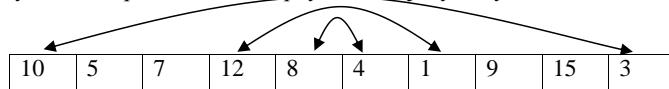
### 5.4 Quick sort 5.4 Quick sort - řazení rozdělováním - známé všude na světě pod jménem **Quick sort**

patří mezi nejrychlejší a nejzajímavější metody řazení polí. Je založeno na mechanismu rozdělení (*partition*), který přeskopí prvky pole do dvou částí tak, že v levé části jsou všechny prvky menší (nebo rovny) jisté hodnotě a v pravé části jsou všechny prvky větší než tato hodnota.

 Když se mechanismus rozdělení (*partition*) postupně aplikuje na všechny vznikající části, které mají větší počet prvků než 2, vznikne seřazené pole.

```
procedure Partition(left,right:integer;
var i,j:integer);
```

Tato procedura rozdělí následující pole na dvě části, **left..j** a **i..right**, které obsahují hodnoty menší resp. větší než 8. Šipky znázorňují výměny.



Pole po rozdělení

|      |   |   |   |   |   |    |   |    |       |
|------|---|---|---|---|---|----|---|----|-------|
| 3    | 5 | 7 | 1 | 4 | 8 | 12 | 9 | 15 | 10    |
| left |   | j |   | 1 |   |    |   |    | right |

### procedura QuickSort

Máme-li má k dispozici proceduru Partition, pak má rekurzivní zápis mechanismu řazení QuickSort tvar:

```
procedure QuickSort(var A:TArr; left, right:integer);
(* Při volání má left hodnotu 1 a right hodnotu N *)
var i,j:integer;
begin
 partition(A,left,right,i,j);
 if left < j then QuickSort(A,left,j); (* Rekurze doleva *)
 if i < right then QuickSort(A,i,right);(* Rek. doprava *)
end;
```

 Tajemství vysoké rychlosti Quicksort je skryto v mechanismu rozdělení - "partition". Jeho autorem je **C.A.R. Hoare**, významná osobnost v oboru teorie a tvorby programů.

**Medián** daného souboru čísel je hodnota, pro níž platí, že polovina čísel v souboru je větší a polovina je menší. Kdybychom znali hodnotu mediánu, mohli bychom použít tento mechanismus:

Procházíme pole zleva a najdeme první číslo, které je větší než medián. Pak procházíme zprava a najdeme první číslo, které je menší než medián. Tato dvě čísla mezi sebou vyměníme a pokračujeme v hledání dalšího většího čísla zleva a dalšího menšího zprava. Procese ukončíme, až se dva indexy (i jdoucí zleva a j jdoucí zprava) překříží. Tím algoritmus "partition" končí a indexy i a j jsou výstupními parametry vymezujícími intervaly **left..j** a **i..right**.



Hoare vtisknul algoritmu "partition" dvě významné vlastnosti:

- 1) Protože stanovení mediánu je náročné, nahradil hodnotu mediánu „libovolnou“ hodnotou z daného souboru čísel. Pracujeme s ní jako s mediánem a říkáme jí „pseudomedián“. Nejvhodnější pro tuto roli je číslo ze středu intervalu - (left+right) div 2. Experimentálně je prokázáno, že toto číslo splní svou roli velmi podobně jako medián.
- 2) Hoare použil pseudomedián jako „zarážku“ a tím ušetřil kontrolu konce pole, (co by se stalo, kdyby pole bylo naplněno stejnými čísly – pseudomedián by byl také toto číslo a při hledání prvního většího zleva by algoritmus musel kontrolovat pravý okraj pole...). Hoare hledal první hodnotu zleva, která je větší **nebo rovna**. A následně zprava, která je menší **nebo rovna**. Rovnost způsobí, že pseudomedián funguje jako zarážka:

#### Algoritmus rozdělení "partition"

```
procedure partition(var A:TArr; left,right:integer; var i,j:integer);
var
 PM:integer; (* pseudomedián *)
begin
 i:=left; (* inicializace i *)
 j:=right; (* inicializace j *)
 PM:=A[(i+j) div 2]; (* ustavení pseudomediánu *)
repeat
 while A[i] < PM do i:=i+1;
 (* hledání prvního i zleva, pro A[i]>=PM *)
 while A[j] > PM do j:=j-1;
 (* hledání prvního j zprava pro A[j]<=PM *)
 if i<=j
 then begin
 A[i]:=A[j]; (* výměna nalezených prvků *)
 i:=i+1;
 j:=j-1
 end
 until i>j; (* cyklus končí, když se indexy i a j překříží *)
end; (* procedure *)
```

#### Nerekurzívní zápis algoritmu

**Nerekurzívní zápis** QuickSortu využívá zásobník. Mechanismus rozdělení Partition rozdělí dané pole na dva segmenty. Jeden z nich se podrobí dalšímu dělení a hraniční indexy druhého se uchovají v zásobníku.

Algoritmus sestává ze dvou cyklů. Vnitřní cyklus provádí opakované dělení segmentu pole a uchovávání hraničních bodů druhého segmentu v zásobníku. Jakmile dělení pokročí tak, že „není co dělit“, vnitřní cyklus se ukončí a opakuje se vnější cyklus. Vnější cyklus vyzvedne ze zásobníku hraniční body dalšího segmentu a vstoupí opět do vnitřního cyklu. Vnější cyklus se ukončí, když je zásobník prázdný a není žádný další segment k dělení.

```

procedure NonRecQuicksort (left,right:integer);
var
 i,j:integer;
 S:TStack; (* deklarace ADT zásobník *)
begin
 SInit(S); (* inicializace zásobníku *)
 Push(S,left); (* vložení levého indexu prvního segmentu *)
 Push(S,right); (* vložení pravého indexu prvního segmentu *)
 while not Sempty do begin (* vnější cyklus *)
 Top(S,right); Pop(S);(* čtení zásobníku v obráceném pořadí *)
 Top(S,left); Pop(S);
 while left<right do begin (* vnitřní cyklus – je co dělit *)
 Partition(A,left,right,i,j);
 Push(S,i); (* uložení intervalu pravé části do zásobníku *)
 Push(S,right);
 right:=j; (* příprava pravého indexu pro další cyklus *)
 end; (* while *)
 end (* while *)
end; (* procedure *)

```

### Analýza



Analýza Quicksortu.

- Quicksort patří mezi nejrychlejší algoritmy pro řazení polí.
- Quicksort je nestabilní a nepracuje přirozeně.
- Asymptotická časová složitost je linearitmická:
  - průměrná časová složitost je  $T_a(n) \sim 17n \cdot \lg_2(n)$
  - časová složitost již seřazeného pole je:

$$T_{usp}(n) \approx 9n \cdot \lg_2(n)$$

- časová složitost pro inverzně seřazené pole je:  
 $T_{inv}(n) \sim 9n \cdot \lg_2(n)$

**Dimenzování zásobníku** pro nerekurzívní zápis Quicksortu:

Při uvedeném algoritmu, kdy se dělí vždy levý segment a pravý se uchovává, je třeba dimenzovat zásobník pro nejhorší případ t.j. , že se vždy segment rozdělí na jeden prvek a zbytek. Pak se musí uchovat **n-1** dvojic indexů.

Algoritmus, který bude dělit vždy menší segment a hranice většího uchovává v zásobníku má nejhorší případ, když se interval vždy rozdělí na dva stejně velké segmenty. V tom případě je třeba zásobník dimenzovat na kapacitu  $\lg_2 n$  dvojic indexů.



**Příklad:** Pokud by se řadilo pole o 1000 prvků, pak v případě prvního algoritmu je zapotřebí zásobník o kapacitě 999 dvojic. Ve druhém případě, kdy se menší segment dělí a větší uchovává, stačí zásobník o kapacitě  $\lg_2 1000$ , t.j. cca 10 dvojic.

Tabulka experimentálně naměřených hodnot pro Quicksort

| n   | 256 | 512 | 1024 |
|-----|-----|-----|------|
| SP  | 10  | 24  | 50   |
| OSP | 22  | 48  | 50   |
| ISP | 12  | 26  | 56   |

K domácímu procvičení: Upravte uvedený algoritmus nerekurzívního zápisu Quicksortu na variantu menšího zásobníku.

**5.5 Shell sort** **5.5 Shell sort - řazení se snižujícím se přírůstkem** je metoda, která ve své době vzbudila pozornost zvýšenou rychlosí a na první pohled nepříliš srozumitelným principem. Z hlediska klasifikace není její zařazení na první pohled snadné. Lze ale říci, že pracuje na principu bublinového vkládání a pracuje tedy na principu vkládání.

Rychlé metody se vyznačují tím, že jednotlivé prvky se přesunují k místu, na které patří, větším krokem (na rozdíl od typických metod s kvadratickou složitostí, jako je Bubble-insert, kde se prvky posunují vždy o jedno místo).

Shell sort pracoval na opakovaných průchodech podobných bublinovému vkládání, kdy vyměňoval prvky vzdálené o stejný krok. Např. následující sekvence (reprezentovaná indexy) může být rozčleněna na čtyři podsekvence čísel vzdálených o krok 4:

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

1 5 9 13 17  
2 6 10 14 18  
3 7 11 15 19  
4 8 12 16 20

V první etapě je s použitím kroku 4 každá ze čtyř sekvencí zpracována jedním bublinovým průchodem. Ve druhé etapě se krok sníží na dvě, vytvoří se dvě podsekvence a každá se zpracuje jedním bublinovým průchodem. V poslední etapě se na celou sekvenci aplikuje bublinový průchod s krokem jedna. Tím je řazení ukončeno.

Teoretické analýzy nenašly nejvhodnější řadu snižujících se kroků (viz skripta Vybrané kapitoly...). Autoři Kernighan a Ritchie (tvůrci jazyka C a Unixu) publikovali verzi algoritmu, v níž první krok byl ( $n \text{ div } 2$ ) a v první etapě docházelo k výměně ( $n \text{ div } 2$ ) dvojic, tak aby všechny byly uspořádány v žádoucím směru. V další etapě se krok vždy půlil a n-tice zpracovávané bublinovým průchodem se zdvojnásobovaly. Poslední etapou byl průchod celým polem s krokem jedna.

```
procedure ShellSort(var A:TArr, N: integer);
var
 step, I, J:integer;
begin
 step:=N div 2; (* první krok je polovina délky pole *)
```

```

while step > 0 do begin (* cykluj, pokud je krok větší než 0 *)
 for I:=step to N-1 do begin (*cykly pro paralelní n-tice *)
 J:=I-step+1;
 while (J>=1) and (A[J]>A[J+step])do begin
 (* bublinový průchod *)
 A[J] :=:A[J+step];
 J:=J-step; (* snížení indexu o krok *)
 end; (* while *)
 end; (* for *)
 step:=step div 2; (* půlení kroku *)
end; (* while *)
end; (* procedure *)

```

**Hodnocení**

Shell sort je nestabilní metoda. Pracuje „in situ“. V uvedené modifikaci pracuje rychleji než Heapsort ale pomaleji než Quicksort. Zatím co v doporučené literatuře (Vybrané kapitoly ...) jsou uvedeny již opuštěné varianty Shell sortu i s tabulkami experimentálně získaných hodnot, shora uvedená metoda nebyla podrobena experimentům se srovnatelnou metodikou, která by dovolila výsledky vzájemně porovnat s předcházejícími metodami. Z literatury a z novějších experimentů však lze říci, že chování uvedené metody je v čase rychlejší než Heapsort a pomalejší než Quick-sort. Nepotřebuje ani předehru pro vytvoření hromady jako Heapsort, ani rekursi nebo zásobník, jako QuickSort. Je proto možné ji nazvat "nekorunovaným králem" řadicích metod a zaslouží si mnohem větší pozornost, než nejznámější a mezi amatéry nejčastěji používané a přitom nejpomalejší bublinové řazení.

**5.6 Řazení setříd'ováním**

**5.6 Řazení setříd'ováním** pracuje na principu slučování - tedy na principu komplementárnímu k principu rozdělování (jako je Quick-sort). Slučování má podobu "setřídění", což je proces sloučení dvou nebo více seřazených posloupností do jedné posloupnosti.

**Setřídění**

**Setřídění** dvou seřazených posloupností do jedné výsledné posloupnosti si můžeme představit na následujícím příkladu:

Mějme dva balíčky karet s celými čísly uspořádané tak, že karta s nejnižším číslem je nahore. Pak vytvoření jednoho výsledného seřazeného balíčku karet získáme takto:

1) Vezmeme dvě horní karty. Porovnáme je a menší z nich vložíme na vrchol výsledného balíčku.

2) Pokud je balíček, jehož kartu jsme právě odložili neprázdný, vezmeme z vrcholu novou kartu a opakujeme krok 1, v jiném případě přeneseme karty zbývajícího balíčku postupně na vrchol výsledného balíčku.

Mechanismus algoritmu setříd'ujícího dva seznamy do jednoho výsledného seznamu ukazuje následující příklad. Výsledný seznam využívá hlavičky, která je v závěru zrušena.



```

procedure MergeLists(L1,L2:TList; var L3:TList);
var PomUk :TList; (* pomocný posuvný ukazatel *)

begin
 new(L3); (* vytvoření hlavičky *)

```

```

PomUk:=L3;
while (L1<>nil) and (L2<>nil) do (* cyklus slučování *)
 if L1^.Data < L2^.Data
 then begin (* "Odložení" prvku z L1 *)
 PomUk^.Puk:=L1;
 PomUk:=L1;
 L1:=L1^Puk
 end
 else begin (* "Odložení" prvku z L2 *)
 PomUk^.Puk:=L2;
 PomUk:=L2;
 L2:=L2^Puk
 end;

 if L1=nil (* připojení neprázdného seznamu L1 nebo L2*)
 then PomUk^.Puk:=L2
 else PomUk^.Puk:=L1
(* Zrušení hlavičky *)
PomUk:=L3;
L3:=L3^.Puk;
dispose (PomUk)
end;

```

 1) Napište úsek programu (procedury), která setřídí prvky typu integer dvou seřazených polí do výsledného pole, jehož velikost je dostatečná (větší nebo rovna součtu velikostí zdrojových polí).

2) Navrhněte algoritmus pro vícenásobné setřídování.

2A) Je dáno více jednorozměrných polí seřazených čísel typu integer (např. řádky v matici) a je dán cílový vektor o dostatečné velikosti. Naplňte cílový vektor hodnotami matice tak, aby jeho prvky byly seřazeny podle velikosti

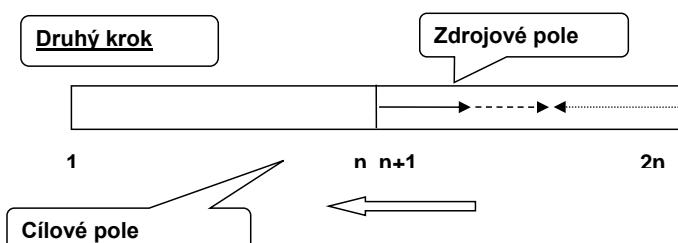
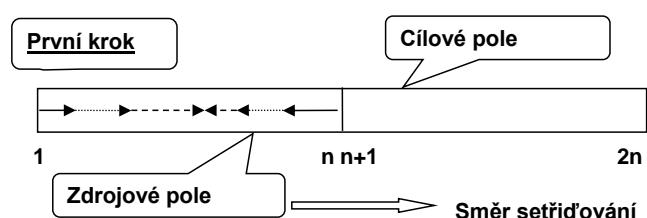
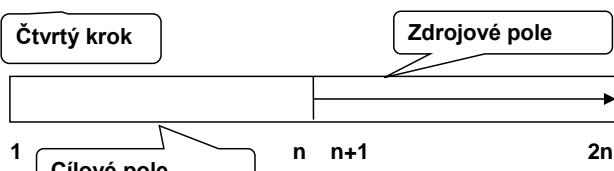
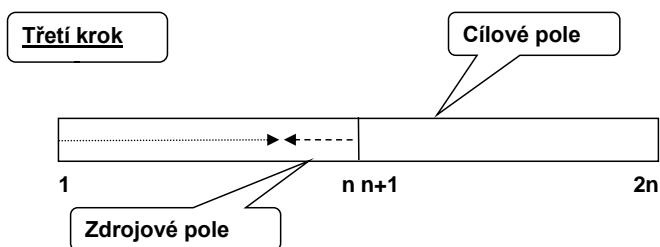
2B) Je dáno pole n ( $n > 2$ ) seřazených zřetězených seznamů čísel typu integer. Vytvořte z nich jeden cílový seznam seřazených čísel.

3) Algoritmy příkladu 2 vyžadují nalezení extrému (např. minimálního prvku) z n prvních prvků nevyčerpaných sekvencí. Navrhněte variantu algoritmu pro příklad 2B, která pro opakování hledání minima použije hromadu (heap).

### 5.7 Merge-Sort

**5.7. Merge-Sort** je sekvenční metoda využívající přímý přístup k prvkům pole. Postupuje polem zleva a současně zprava a setřídíuje dvě „proti sobě“ postupující neklesající posloupnosti. Výsledek se ukládá do cílového pole a počet vzniklých posloupností se počítá v počítaadle. Algoritmus končí, vznikne-li jen jedna cílová posloupnost.

Schéma postupu řazení Merge-Sort je uvedeno na následujících obrázcích. Šipky zleva a zprava ve zdrojovém poli představují dvojice neklesajících posloupností, které jsou setříděny do jedné posloupnosti, která se uloží do cílového pole. V jednotlivých krocích se střídá levá a pravá polovina pole v roli zdrojového a cílového prostoru. Tato skutečnost je implementována pomocí tzv. "houpačkového" mechanismu použitého v algoritmu.

$x+y$  $x+y$ 

Algoritmus je podrobně popsán na str. 177 textu skript "Vybrané kapitoly..." (str.18/44 stran souboru pdf), které najdete na webových stránkách předmětu

( <https://www.fit.vutbr.cz/study/courses/IAL/private/Texty/Skripta/ch7.pdf> ).

Významným rysem algoritmu je jeho houpačkový mechanismus, který automaticky střídá pozici zdrojového a cílového pole i krok postupující proti sobě orientovanými slučovanými neklesajícími posloupnostmi. Tento mechanismus patří k základním programovacím technikám (viděli jsme ho již na verzi Bublinového řazení - „Shakersort“). V uvedeném textu je demonstrován způsob postupného vytváření algoritmu postupným zjemňováním (*stepwise refinement*) - snižováním abstrakce jednotlivých jeho částí.

 Metoda Merge-sort je nestabilní, nechová se přirozeně a nepracuje „in situ“.

Asymptotická časová složitost je  $TM(n) \sim (28 * n + 22) \lg_2(n)$

Algoritmus je velmi rychlý, ale z hlediska konstrukce programu nepatří k jednoduchým a často používaným algoritmům. Z hlediska programovacích technik patří k velmi zajímavým algoritmům.

Experimentálně naměřené hodnoty jsou uvedeny v následující tabulce:

|     |     |     |
|-----|-----|-----|
| n   | 256 | 512 |
| SP  | 8   | 13  |
| NUP | 6   | 12  |
| ISP | 32  | 72  |

**5.8. List Merge Sort - řazení polí setřídováním seznamů** - je metoda řazení založená na **ListMergeSort** principu slučování s využitím principu řazení bez přesunu položek.

V prvním kroku se musí zřetězit neklesající posloupnosti s pomocí indexů pole Pom v roli ukazatelů. Začátky neklesajících posloupností se vloží do pomocné datové struktury typu **seznam**. Koncový index má hodnotu nula (nil). V následujícím obrázku je pět neklesajících posloupností. Čtvrtá z nich má délku rovnou jedné.



| Ind | 1 | 2 | 3 | 4 | 5 | 6  | 7  | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|-----|---|---|---|---|---|----|----|---|---|----|----|----|----|----|
| A   | 2 | 5 | 9 | 3 | 7 | 10 | 16 | 5 | 8 | 6  | 3  | 11 | 18 | 20 |
| Pom | 2 | 3 | 0 | 5 | 6 | 7  | 0  | 9 | 0 | 0  | 12 | 13 | 14 | 0  |

V následujícím cyklu se vyzvednou ze začátku seznamu začátky dvou zřetězených neklesajících posloupností. Jejich setříděním vznikne jedna zřetězená neklesající posloupnost. Její začátek se vloží na konec seznamu. Algoritmus končí, je-li v seznamu již jen začátek jedné zřetězené neklesající posloupnosti. Výsledek se může do podoby seřazeného pole zpracovat např. MacLarenovým (M.Donald MacLaren) algoritmem.

Jádrem algoritmu je setřídění dvou seznamů zřetězených v pomocném poli indexovými ukazateli. Níže uvedený algoritmus používá, na rozdíl od algoritmu uvedeném ve skriptech „Vybrané kapitoly...“

([https://www.fit.vutbr.cz/study/courses/IAI/private/Texty/Skripta/ch7.pdf\\_uložených\\_v\\_IS](https://www.fit.vutbr.cz/study/courses/IAI/private/Texty/Skripta/ch7.pdf_uložených_v_IS) na str.187 textu a str. 28/44 souboru pdf) jen jednu frontu a je tudíž jednodušší.

Algoritmus ListMergeSortu lze v algoritmickém pseudojazykem popsat takto:

**QInit(Q); (\* Inicializace fronty začátků seznamů \*)**

---

(\* 1 \*) Při průchodu polem řazených prvků zřetězíme prvky neklesajících posloupnosti pomocí pomocného pole indexů (ukazatelů). Přitom index (ukazatel) začátku každé neklesající posloupnosti vložíme do seznamu L. Index posledního prvku každé posloupnosti má hodnotu nula (nil). Výsledkem tohoto bloku jsou zřetězené seznamy a seznam L naplněný jejich začátky.

---

```
repeat (* Předpokládá se pole o nenulové délce - alespoň jeden seznam *)
 CopyFirst(L, Zac1); (* Čtení začátku prvního seznamu *)
 DeleteFirst(L); First(L);
 if Active(L)
 then begin (* ve frontě jsou alespoň dva seznamy, lze číst druhý *)
 CopyFirst(L, Zac2); (* Čtení začátku druhého seznamu *)
 DeleteFirst(L); First(L);
 if A[Zac1] < A[Zac2]
 then InsertLast(L, Zac1)(* Uložení výsledného začátku do fronty*)
 else InsertLast(L, Zac2) (* Uložení výsledného začátku do fronty*)
```

---

(\* 2 \*) Setříď seznamy Zac1 a Zac2; koncový index nastav na nulu.

---

```
end (* if *)
until not Active(L);
```

Výsledek má podobu seřazeného zřetězeného seznamu, který lze např. MacLarenovým algoritmem umístit do téhož pole tak, aby se vytvořilo pole seřazených položek.

Příklad:

Vytvořte úsek programu (procedury), která setřídí dva seznamy implementované zřetězením v poli. Řešením je modifikace algoritmu uvedeného v odstavci o setřídění.

ListMergeSort je algoritmus pracující bez přesunu položek. Je potenciální stabilní (ve skriptech „Vybrané kapitoly...“ je použita fronta a proto je tato verze nestabilní). Stabilita se musí zajistit tím, že se použije obostranně ukončená fronta (DEQUE) do níž se uloží začáty seznamů. Pro setřídování se vzemou dva seznamy ze začátku a při rovnosti položek se napřed vezme položka z prvního seznamu a výsledek se opět uloží na začátek fronty DEQUE. Jakmile je v DEQUE jen jeden seznam, je to výsledek řazení. Experimentálně

byly naměřeny hodnoty uvedené v následující tabulce.

| n   | 256 | 512 |
|-----|-----|-----|
| SP  | 2   | 6   |
| NUP | 32  | 74  |
| OSP | 22  | 48  |

### 5.9. Radix sort

**5.9. Radix-sort řazení tříděním podle základu** - je počítačovou obdobou řazení tříděním na děrnoštítkových třídicích strojích.

Na třídicích strojích je základem desítková číslice na daném rádu v daném sloupci štítku. Ve většině počítačových aplikací je to desítková číslice čísla v kódu BCD( Binary-Coded Decimal).

Řazení tříděním je principiálně metodou pracující bez přesunu položek. Proto je třeba vytvořit k řazenému poli pomocné pole ukazatelových indexů.

Následující obrázek demonstruje princip řazení tříjmístných čísel. Na druhém rádku jsou trojmístná čísla seřazena podle hodnoty poslední číslice předcházejícího rádku. Metoda řazení musí být stabilní a proto je uspořádání předchozího rádku významné. Na třetím rádku jsou čísla seřazena podle pořadí předposlední číslice. Na posledním rádku jsou čísla seřazena podle hodnoty první číslice. Poslední rádek již vytváří posloupnost seřazenou podle hodnoty čísel. Metoda je tedy obdobná metodě "řazení podle více klíčů", kde jsou jednotlivé klíče reprezentovány číslicemi a postupuje se od nejnižšího rádu k nejvyššímu.

| x+y | 008 | 381 | 966 | 377 | 504 | 625 | 199 | 552 | 230 | 416 | 833 | Pole před řazením    |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|----------------------|
|     | 230 | 381 | 552 | 833 | 504 | 625 | 966 | 416 | 377 | 008 | 199 | Třídění podle řádu Ø |
|     | Ø   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   |     |                      |
|     | 504 | 008 | 416 | 625 | 230 | 833 | 552 | 966 | 377 | 381 | 199 | Třídění podle řádu 1 |
|     | Ø   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   |     |                      |
|     | 008 | 199 | 230 | 377 | 381 | 416 | 504 | 552 | 625 | 833 | 966 | Třídění podle řádu 2 |
|     | Ø   | 1   | 2   | 3   | 4   | 5   | 6   | 7   | 8   | 9   |     |                      |

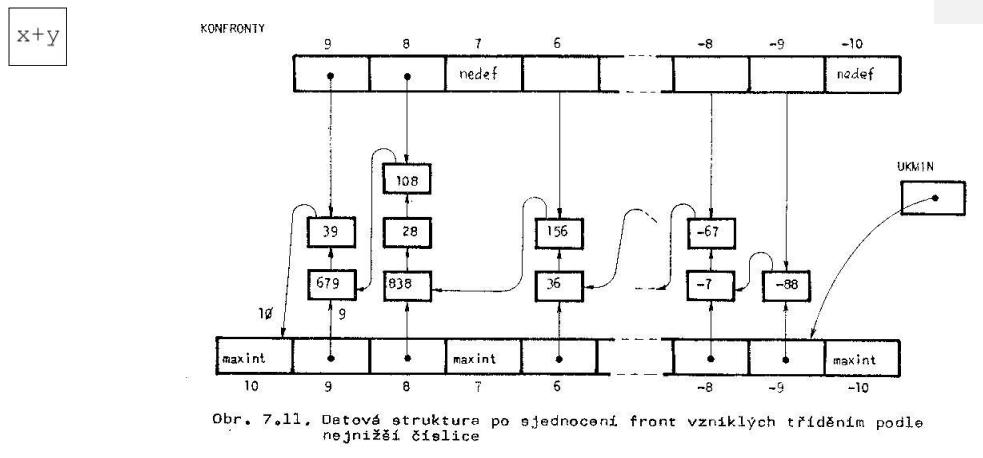
Obr. 7.10. Řazení trojciferných čísel podle základu 10

Implementaci datových struktur pro řazení čísel v BCD kódu znázorňuje následující obrázek. Jednotlivé číslice mohou mít hodnotu 0 až 10 a číslo může být kladné nebo záporné. Je nutno rozlišit stejnou číslici kladného od stejné číslice záporného čísla a proto budou vytvářeny "příhrádky" (podobně těm u děrnoštítkového stroje) pro 20 různých

číslíc.

Pozn. I nula zde bude mít svou kladnou a zápornou "přihrádku", což na obrázku není patrné.

"Přihrádky" jsou reprezentované seznamy. Ukazatelé na začátky a konce seznamů jsou v polích "KONFRONTY" a "ZACFRONTY". "Přihrádka" je jako prázdná inicializována hodnotou maxint v poli začátků.



V první fázi algoritmu se průchodem polem zjistí, kolik číslic má číslo s největším počtem číslic. Tato hodnota - POCCIF - určuje počet průchodů řadicího cyklu.

Tělo počítaného cyklu sestává z inicializace datových struktur pro seznamy (přihrádky). V prvním průchodu cyklem se prochází polem se **zvyšujícím se indexem** a jednotlivá čísla se zařazují do seznamů (přihrádek) podle hodnoty nejnižší číselice. Na konci průchodu se všechny seznamy spojí do jednoho seznamu (od nejmenšího k největšímu), jak to znázorňují šipky na obrázku. Ve druhém průchodu cyklem se znova inicializuje datové struktury pro nové zařazování do "přihrádek", ale pole se již neprochází od začátku do konce se zvyšujícím se indexem, ale prochází se jím **jako jedním seznamem s pomocí ukazatelů, které jej zřetězují!**

Radix-sort je stabilní metoda. Stav uspořádání nemá podstatný vliv na čas a proto se jeví jako by se nechoval přirozeně. Metoda nepracuje „in situ“, protože potřebuje pomocné pole indexových ukazatelů.

Časová složitost má tvar:

$$TMAX(n) = (42 * POCCIF + 15) * n + (16 + 34 * ZAKLAD) * POCCIF + 15$$

což vyjadřuje **lineární složitost!** Teoreticky je to tedy nejrychlejší algoritmus.

Experimentálně byly naměřeny tyto hodnoty pro náhodně uspořádaném pole:

|            |            |            |             |
|------------|------------|------------|-------------|
| <b>n</b>   | <b>256</b> | <b>512</b> | <b>1024</b> |
| <b>NUP</b> | <b>24</b>  | <b>60</b>  | <b>102</b>  |

Schéma algoritmu řazení RadixSort

```

begin
 Inicializace proměnných;
 Stanovení hodnoty POCCIF – maximálního počtu číslic (míst)
 for j:=1 to POCCIF do begin
 Inicializace pole ZACFRONTY;
 Třídění do front podle j-té číslice
 Nalezení nejnižší neprázdné fronty (v poli ZACFRONTY) a
 uložení jejího ukazatele do UKMIN
 Spojení front do jediného seznamu počínaje prvkem A[UKMIN]
 end; (* for *)
 Sekvenční seřazení prvků seznamu do výstupního pole
end;

```

15.10.2005

59

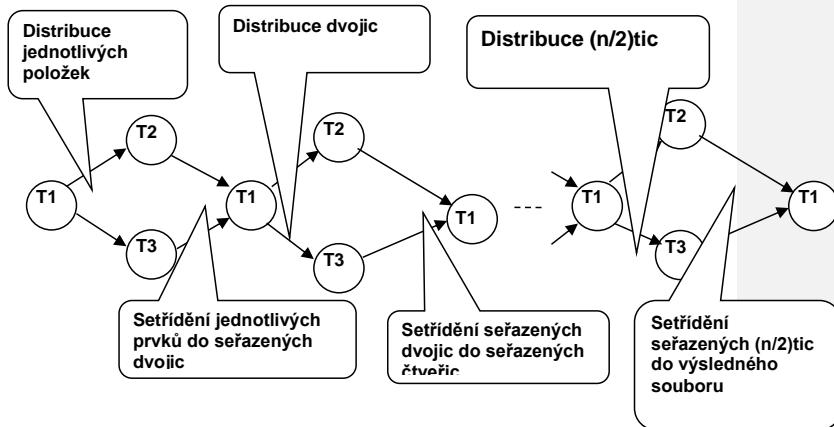
### 5.10 Sekvenční řazení

**5.10 Sekvenční řazení** je nejčastěji známo pod pojmem **řazení souborů**, protože soubory měly původně typicky sekvenční organizaci. Historicky byly reprezentovány zařízením s magnetickou páskou a proto někdy metody nazývají také řazení magnetických pásek. řazení souborů je setřídění - *merging* a práce s neklesajícím posloupnostmi.

### Přímé setřídování souborů

**Přímé setřídování souborů** znázorňuje následující obrázek, na němž kroužky s písmenem T znázorňují soubor (kotouč nebo kazetu s magnetickou páskou). Této metodě se také říká "třípásková přímá metoda".

### Třípásková přímá metoda

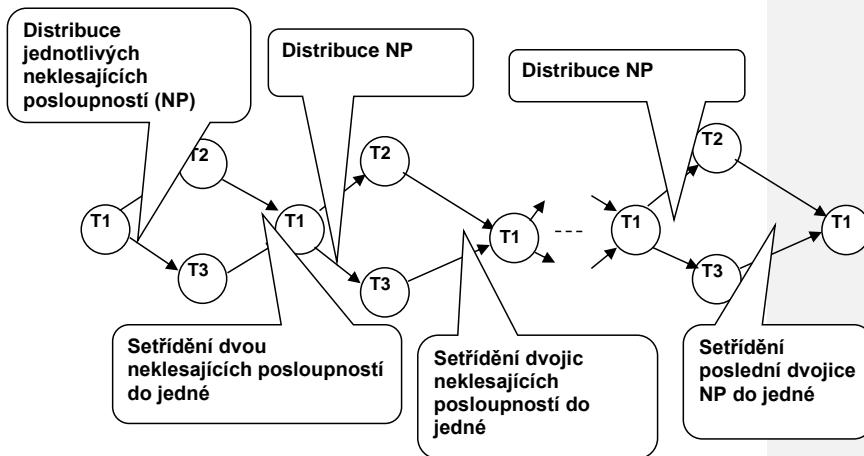


V této metodě jsou zapotřebí tři soubory (pásky). Jeden zdrojový soubor (T1) a dva pracovní soubory (T2 a T3). Každá fáze sestává ze dvou částí: distribuční a setřídovací. V první fázi se prvky zdrojového souboru rozdělí - distribuují rovnoměrně do dvou souborů T2 a T3 ve druhé části této fáze se čtením prvních prvků těchto vytvoří seřazené dvojice a uloží se do T1.

Ve druhé fázi se podobným způsobem distribuuje seřazené dvojice a následně se setřídí a seřazené čtveřice a uloží se opět na T1. Tak se postupuje v distribuci  $2^n$ -tic a jejich setřídování do  $2^{n+1}$ -tic tak dlouho, až s vznikne jedna seřazená posloupnost. Je zřejmé, že počet fází lze vyjádřit vztahem  $\lg n$  (kde závorky  $\lfloor \rfloor$  vyjadřují úpravu hodnoty na nejbližší vyšší (nebo rovné) celé číslo), kde n je počet prvků seřazované posloupnosti.

### Třípásková přirozená metoda

### Třípásková přirozená metoda



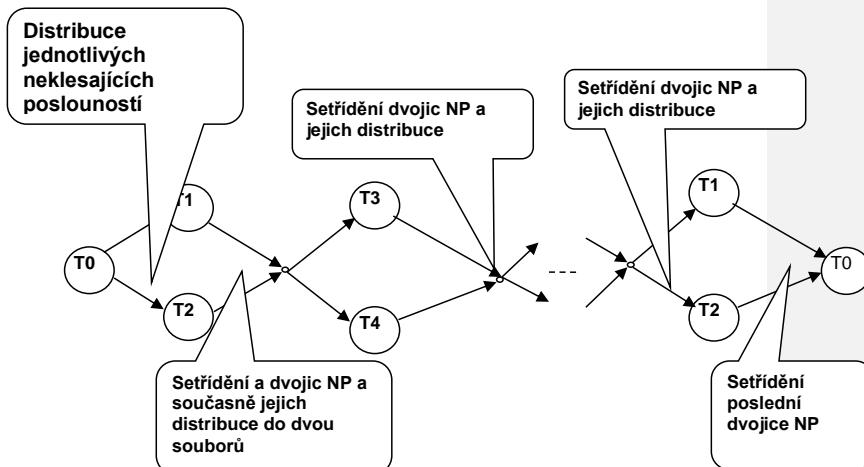
Zásadní rozdíl mezi třípáskovou přímou a přirozenou metodou spočívá v tom, že přirozená metoda distribuuje a setřídíuje neklesající posloupnosti zdrojových souborů. Z toho vyplývá, že hodnota počtu fází má podobný tvar, ale N je počet neklesajících posloupností ve zdrojovém souboru. Znamená to, že již seřazený soubor je zpracován v jedné fázi, s lineární složitostí. Počet potřebných fází je tedy  $\lceil \lg_2 N \rceil$ , kde N je počet neklesajících posloupností zdrojového souboru.

#### Čtyřpásková přirozená metoda

**Čtyřpásková přirozená metoda** je předchůdcem mnohačestného vyváženého setřídování, které bude následovat. Bylo by možné hovořit i o čtyřpáskové přímé metodě, ale protože tato prapůvodní metoda je již zcela neaktuální, nemá smysl ji dále vylepšovat.

Čtyřpásková metoda vyžaduje o jednu páskovou mechaniku (nebo o jeden pracovní soubor) více. Ve světě technického vybavení nepředstavovala magnetopásková mechanika bezvýznamné náklady a proto se vždy musel vážit její přínos.

Čtyřpásková metoda výrazně zvyšuje rychlosť zpracování pro třídění, protože každou fázi redukuje o distribuční krok, který je u mechanických zařízení provázen potřebou převinutí (*rewind*).



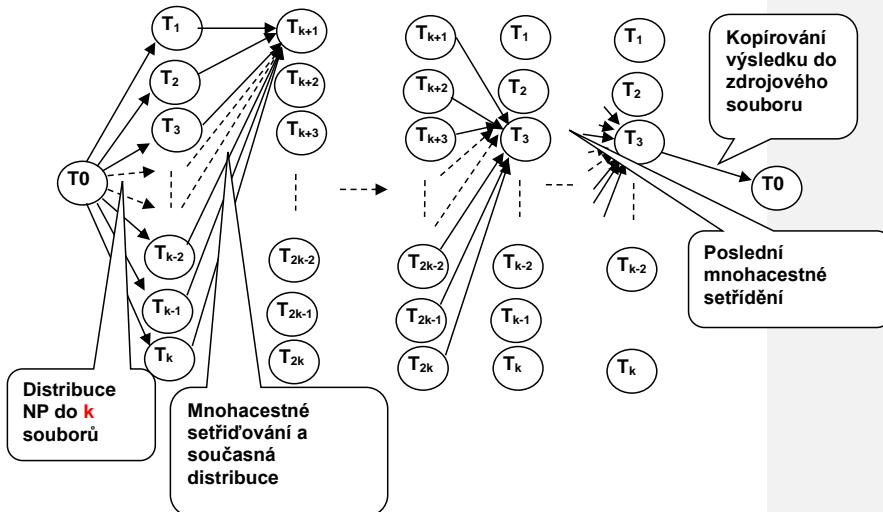
Po prvotní distribuci neklesajících posloupností do dvou pracovních souborů se provádí setřídování každé dvojice posloupností a jejich střídavé ukládání na další dvojici pracovních souborů.

#### Mnohacestné vyvážené setřídování

**Mnohacestné vyvážené setřídování (balanced multiway merging)** je jen rozšířením čtyřpáskové metody na  $2k$  souborů ( $k > 2$ ) s použitím mnohacestného setřídování neklesajících posloupností (kde  $k$  je počet souborů na zdrojové a cílové straně systému).

Zatímco 4 pásková metoda slučovala dvojice neklesajících posloupností, mnohacestné seřazování, které používalo n zdrojových a n cílových souborů, setřídovalo mnohacestně n zdrojových posloupností do jedné cílové posloupnosti. Až doposavad v logaritmicky vyjadřované časové složitosti dominoval základ s hodnotou dvě, protože šlo o "půlení" nebo o jev spojující dvě části. V mnohacestném vyváženém setřídování přebírá hodnotu základu číslo k, vyjadřující počet souborů na jedné straně. Pak je tedy počet fází dán vztahem  $\lceil \lg k \rceil$ .

Toto řazení bylo velmi populární v 60.a 70.letech, kdy sálovým počítačům vévodily řady skříní magnetopáskových jednotek, jejich počet měl nejčastěji hodnotu celé mocniny dvou, a tak se opravdu velké počítače mohly chlubit 16 nebo i 32 skřínemi magnetopáskových jednotek.



$\Sigma$  Mnohacestné vyvážené setřídování patří k nejvýkonnějším metodám sekvenčního řazení. Má uplatnění i v době, kdy mechanické sekvenční magnetopáskové paměti již nepatří k běžné výbavě výkonných počítačů. Jeho principu lze využít i při řazení rozsáhlých souborů organizovaných v pamětech s index-sekvenčním přístupem a zejména u souborů implementovaných rozsáhlými seznamy.

## Polyfázové řazení

**Polyfázové seřazování** zmíníme hlavně pro zajímavost jeho myšlenky a pro další ukázku využití Fibonacciho posloupnosti. Polyfázové řazení je založeno na snaze „ušetřit“ počet potřebných souborů (kdysi každý soubor představoval jednu magnetopáskovou mechaniku!!) při zachování rychlosti dané 2K soubory. Polyfázové setřídování má s použitím K+1 souborů rádově shodnou složitost jako mnohafázové setřídování s 2K soubory. Vykazuje tedy „úsporu“ K-1 souborů (mechanik).

Metoda je založena na principu postupného setřídování K neklesajících posloupností ze zdrojových souborů do jednoho cílového souboru tak dlouho, dokud se jeden ze zdrojových souborů nevyprázdní (neobsahuje již žádnou neklesající posloupnost). Pak se tento prázdný soubor zamění s cílovým souborem a proces setřídování pokračuje tak dlouho, až se vytvoří jediný výsledný cílový seřazený soubor.

Potíž metody je situace, kdy se ve stejném okamžiku vyprázdní více, než jeden soubor. Lze najít počáteční rozdělení posloupností tak, aby se v každém následujícím kroku vyprázdnila vždy jen jeden soubor?

Tento problém lze vyřešit pomocí Fibonacciho posloupnosti. Fibonacciho posloupnost k-tého řádu je definována k+1 počátečními hodnotami. Následující prvek má hodnotu součtu aktuálního prvku a k předchozích prvků.

Fibonacciho posloupnost 1. řádu s inicializačními konstantami 0 a 1 má další členy: 1,2,3,5,8,13,21,34,55,...

Fibonacciho posloupnost 4. řádu s inicializačními hodnotami 0,0,0,0,1 má další členy:  
1,2,4,8,16,31,61,120,236...

Neklesající posloupnosti pro třípáskovou polyfázovou metodu, která pracuje ze dvou zdrojových souborů do jednoho cílového souboru, lze na počátku rozdělit s využitím Fibonacciho např. posloupnosti takto:

| f1 | f2 | f3 | $\Sigma$ |
|----|----|----|----------|
| 13 | 8  | 0  | 21       |
| 5  | 0  | 8  | 13       |
| 0  | 5  | 3  | 8        |
| 3  | 2  | 0  | 5        |
| 1  | 0  | 2  | 3        |
| 0  | 1  | 1  | 2        |
| 1  | 0  | 0  | 1        |

Počáteční rozložení pro 21 posloupnosti je: 5 a 5+8 .

Pro soustavu 6 pásek (souborů), se budou neklesající posloupnosti z 5 zdrojových souborů setřídovat do jednoho cílového souboru. Pro inicializační rozdělení se použije Fibonacciho posloupnost 4. řádu, která má 5 počátečních hodnot a má tvar: 0,0,0,1,1,2,4,8,16,31,61,120,236,...

| f1 | f2 | f3 | f4 | f5 | f6 | $\Sigma$ |
|----|----|----|----|----|----|----------|
| 16 | 15 | 14 | 12 | 8  | 0  | 65       |
| 8  | 7  | 6  | 4  | 0  | 8  | 33       |
| 4  | 3  | 2  | 0  | 4  | 4  | 17       |
| 2  | 1  | 0  | 2  | 2  | 2  | 9        |
| 1  | 0  | 1  | 1  | 1  | 1  | 5        |
| 0  | 1  | 0  | 0  | 0  | 0  | 1        |

Tabulka Fibonacciho posloupnosti 4. řádu:

|       |   |   |   |   |   |   |   |   |   |    |    |    |     |     |
|-------|---|---|---|---|---|---|---|---|---|----|----|----|-----|-----|
| i     | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  | 10 | 11 | 12  | 13  |
| $f_i$ | 0 | 0 | 0 | 0 | 1 | 1 | 2 | 4 | 8 | 16 | 31 | 61 | 120 | 236 |

65 posloupností je dán počátečním součtem inicializačního rozdělení  $16+15+14+12+8$ .

kde  $16=1+1+2+4+8$

$$15=1+2+4+8$$

$$14=2+4+8$$

$$12=4+8$$

$$8=8$$

Podobně pro 129 posloupností bychom dostali počáteční rozdělení: 31,30,28,24,16

kde  $31=1+2+4+8+16$

$$30=2+4+8+16$$

... atd.

Inicializační hodnoty jsou tedy dány součtem postupně zleva se snížujícího počtu předchozích hodnot. Z toho vyplývá, že vyhovující počáteční rozdělení je možné jen pro určité hodnoty celkového počtu posloupností. Otázka, jak řešit případy libovolného počtu je otázkou vhodné distribuce „prázdných“ posloupností, které „zaslepí“ použití některých souborů. Uzávorka tabulky „vhodných“ čísel pro metody s různými počty souborů je uvedena ve skriptech "Vybrané kapitoly.Pozn. Ve skriptech je formálně nesprávný popis získání inicializačních hodnot.

## Závěr

### Závěr

Řazení je třetím nejvýznamnějším tématem předmětu. Kapitola seznamuje studenty s nejvýznamějšími algoritmy vnitřního řazení (řazení polí) i sekvenčního (vnějšího) řazení (řazení souborů) a to v s rekurzivním i nerekurzivním zápisem tam, kde je to účelné.

## 6. Vyhledávání v textu

### Vyhledávání

#### v textu

**Didaktické cíle:** Vyhledávání podřetězců v řetězích spadá do oblasti vyhledávání a je tedy dodatečným rozšířením kapitoly 4. Tato problematika spadá do kategorie zpracování textu (*text processing*) a je předmětem výzkumu tzv. "stringologie". S principem vyhledávání podřetězců (vzorků, *patterns*) se setkáváme nejen u řady "vyhledavačů" a jazykových editorů a korektorů ale také např. u antivirových programů, které v programu vyhledávají identifikovaný nejvýznamnější vzorek virového programu. Dále uvedené metody představují ukázky principů nyní již rozsáhlého souboru vyhledávacích metod. Student se naučí základům algoritmů vyhledávání podřetězců v řetězích. Kapitola má spíše doplňkový charakter.

**DEF**

Pro další popis metod budeme používat tyto konvence:

**P**..... vyhledávaný vzorek (*pattern*):

**P[i]** .... i-tý znak vzorku

**m**=length(**P**) nebo **PL** .... délka vzorku

**T**..... Prohledávaný text

**T[i]**..... i-tý znak prohledávaného textu

**n**=length(**T**) nebo **TL** ..... délka prohledávaného textu

Klasický algoritmus, ke kterému intuitivně dojde každý pokročilý programátor má tvar:

```
function Match(P,T:String; PL,TL:index) :index;
(* V případě neúspěchu vrátí hodnotu TL+1 *)
var PomZacT, BezT, BezP:index;
begin
 PomZacT:=1; BezT:=1; BezP:=1; (* inicializace *)
 while (BezT<=TL) and (BezP <=PL) do
 if T[BezT]=P[BezP]
 then begin (* Posun po vzorku v řetězci *)
 inc(BezT); inc(BezP)
 end else begin (* posun začátku řetězce a nové porovnání *)
 inc(PomZacT); BezT:=PomZacT; BezP:=1;
 end; (* while, if *)
 if BezP>PL
 then Match:=PomZacT (* Našel *)
 else Match:=BezT (* t.j. Nenašel a vrátil hodnotu TL+1 *)
 end. (* function *)
```

Analýza algoritmu

- Je-li vzorek na počátku řetězce, provede se PL porovnání (nejlepší případ)
- Není-li P[i] v řetězci T obsažen, provede se TL srovnání.

V nejhorším případě dojde na každé startovací pozici k (PL-1) shodám. Pak se provede mn srovnání a algoritmus má složitost O(mn). Pro toto musíme ukázat, že takový případ může nastat. (V některých algoritmech může být jeden krok časově

náročný, zatím co jiný nikoliv). Takový případ je např.  $P='AAA...AB'$  a  $T='AAA...AAA'$ .

V přirozených jazycích jsou takové případy řídké. Uvedený algoritmus v nich pracuje průměrně dobře. (Statistiky ukazují, cca. 1.1 porovnání na jeden znak řetězce  $T$ ). Pro některé aplikace je však nepřijatelný návrat v řetězci (t.j.  $\text{Bez}T:=\text{PomZac}T$  v cyklu). Následující algoritmus byl vytvořen pro odstranění návratu a zlepšuje jeho nejhorší případ.

### 6.1. Knuth - Morris-Prattův algoritmus

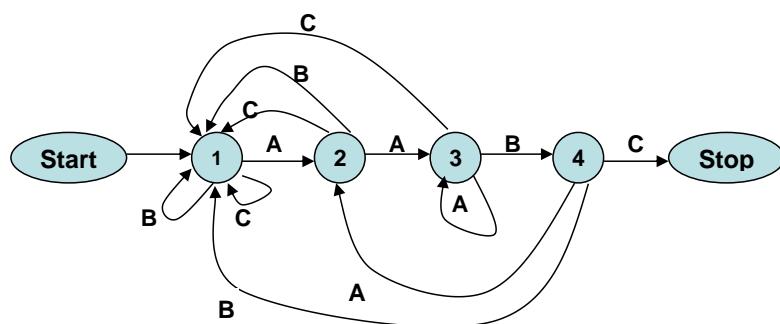
**DEF** **6.1. Knuth-Morris-Prattův algoritmus** (KMP) využívá ke své práci konečný automat. Lze ho interpretovat jako zvláštní druh stroje - procesoru, nebo vývojového (postupového) diagramu či grafu a můžeme mu porozumět bez hlubší znalosti teorie automatů.

Použitý konečný automat v podobě grafu má tři typy uzelů:

- Startovací uzel START
- Uzel STOP s významem "úspěšný konec"
- Uzel READ s významem "čti další znak". Dojdeme-li v prohledávaném řetězec na konec a není k dispozici další znak, jde o neúspěšný konec

Nechť  $A$  je množina znaků (abeceda), které mohou být v řetězci, a  $o=|A|$  je kardinalita abecedy. Pak z každého uzlu vychází o orientovaných hran, oceněných jednotlivými znaky abecedy. Pro vzorek 'AABC' a abecedu  $\{A,B,C\}$  dostaneme automat vyjádřený následujícím grafem:

x+y



Nevýhodou tohoto řešení je skutečnost, že z každého uzlu vychází tolik hran, kolik je znaků abecedy.

Graf automatu KMP používá pouze dvě hrany:

ANO – souhlas – Y

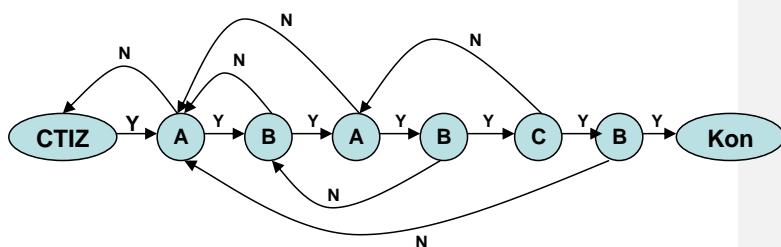
NE – nesouhlas – N

Pro vzorek 'ABABCB' má automat KMP tvar znázorněný v následujícím obrázku v němž:

**DEF**

1. na hraně N se porovná starý (přečtený) znak s uzlem
2. hrana Y z uzlu CTIZN čte nový znak
3. Na hraně Y se čte nový znak a porovná se s uzlem
4. Dosažení uzlu Kon znamená nalezení vzorku
5. Dosažení konce prohledávaného textu před dosažením STOP znamená neúspěšné nalezení

**x+y**



#### Konstrukce automatu KMP:

Automat je reprezentován vzorkem P a vektorem FAIL, který má prvky typu integer, reprezentující cílový index zpětné šipky.

Příklad tvorby vektoru pro P='ABABABC'B'

Vždy platí, že FAIL[1]=0;

| index | 1        | 2        | 3        | 4        | 5        | 6        | 7 | 8     | 9 |
|-------|----------|----------|----------|----------|----------|----------|---|-------|---|
| P:    | A        | B        | <b>A</b> | <b>B</b> | <b>A</b> | <b>B</b> | C | B     |   |
| T:    | <u>A</u> | <u>B</u> | <u>A</u> | <u>B</u> | A        | B        | X | ..... |   |

Předpokládejme, že na 7. pozici se **x=>C**; pak další možné místo, na kterém může vzorek v textu začínat je třetí pozice Protože došlo k nesouhlasu na 7 pozici a protože platí:

(P1 ... P4)=(P3...P6)

To znamená, že nové porovnání může začít na indexu 5 a tedy FAIL[7]=5

Vyjádřeno jinak:

$$\text{FAIL}[k] = \max r \{ (r < k) \text{ and } (P_1..P_{r-1}) = (P_{k-r+1}..P_{k-1}) \}$$

Vektor Fail má pro vzorek 'ABABABC'B' hodnoty: 0,1,1,2,3,4,5,1.

Vytvoření vektoru FAIL popisuje následující procedura:



Aktivní znalost následujícího algoritmu nepatří k látce požadované ke zkoušce.

```

procedure KMPGraf(P:String;PL:index;var Fail:TFail);
var
 k,r: index;
begin
 Fail[1]:=0;
 for k:=2 to PL do begin
 r:=Fail[k-1];
 while (r>0) and (P[r]<>P[k-1]) do r:=Fail[r];
 (* v zápisu cyklu je použit zkratový booleovský výraz ! *)
 Fail[k]:=r+1
 end; (* for *)
end; (* procedure *)
Celkový počet porovnání je (2m-3). To představuje lineární časovou složitost.

(Odvození složitosti je v následujícím textu).

```



#### Analýza algoritmu:

Na základě tvaru algoritmu lze říci, že algoritmus má složitost  $O(m \cdot 2)$ , protože vnější cyklus se provede m krát a ve vnitřním cyklu se r posunuje někam k nule, v nejhorším případě m krát. To je však povrchní rozbor.

Posuďme počet úspěšných porovnání  $P[r]=P[k-1]$ , protože úspěšné porovnání ukončí cyklus. Celkem se provede maximálně  $(m-1)$  úspěšných porovnání  $\rightarrow$  (for  $k:=2$  to  $PL$  do ...). Po každém neúspěšném porovnání se sníží r (protože  $Fail[r] < r$ ), takže lze počet neúspěšných porovnání omezit shora určením, kolikrát se sníží r. Všimněme si, že:

- r se na počátku přírádí  $Fail[1]$  t.j. =0
- r se zvýší právě o 1 s každým cyklem for, protože:  
 $Fail[k]:=r+1$   
 k se zvyšuje cyklem for  
 $r:=Fail[k-1]$   
 čili, první příkaz cyklu provádí vlastně  $r:=r+1$
- r se zvýší celkem  $(m-2)$  krát
- r je vždy nezáporné

Protože r začíná s nulovou hodnotou, zvyšuje se  $(m-2)$  krát a není nikdy záporné, nemůže se snižovat vícekrát, než  $(m-2)$  krát. Počet neúspěšných porovnání je tedy  $(m-2)$  a celkový počet porovnání je tedy  $(2m-3)$ . To představuje lineární časovou složitost.

**Algoritmus KMP** Algoritmus pro KMP vyhledání vzorku v řetězci

```

function KMPMatch(T,P:string; TL,PL:index;
Fail:TFail):index;
var
 TInd,PInd:index;
begin
 TInd:=1; PInd:=1;
 while (TInd<=TL) and (PInd<=PL) do
 if (PInd=0) or (T[TInd]=P[PInd]) (* Zkratové Bool. výr.! *)
 then begin
 inc(TInd);

```

```

 inc(PInd)
end else (* jdi po hraně "N" *)
 PInd:=Fail[Pind];
if PInd>PL
then KMPMatchL=TInd-PL (* Našel vzorek na indexu TInd-PL *)
else KMPMatch:=TInd (* Nenašel, vrací hodnotu TL+1 *)
end; (* function *)

```

Tento algoritmus provede maximálně  $2n$  porovnání ( $n=TL$ ), a celkový algoritmus vyhledávání, který je složen z konstrukce automatu a z vyhledání má tedy složitost  $O(n+m)$ , což je lepší než  $O(nm)$ .

Některé empirické studie ukazují, že přímý i KMP algoritmus udělají v řetězci vytvořeném přirozeným jazykem přibližně stejný počet porovnání, ale KMP nejde v textu zpět, což lze považovat za jistou výhodu.

## 6.2. Boyer Mooreův algoritmus

Boyer-Mooreův algoritmus (BMA) vychází z úvahy, že při porovnávání vzorku s řetězcem, lze některé znaky, o nichž lze prohlásit, že se nemohou rovnat, přímo přeskočit. Čím delší je vyhledávaný vzorek (a čím více se poskytuje informace) tím větší počet znaků v prohledávaném řetězci může dobrý algoritmus přeskočit. BMA používá dva heuristické principy pro postup v prohledávaném řetězci.

### První heuristika BMA

(Pozn. Heuristika je podle encyklopédie zvláštní způsob řešení problémů; postup vedoucí k nalezení optimálního řešení problému bez znalosti daného algoritmu; procedura objevení, invence, která může vést k řešení. Heuristicke metody se liší od formálních, založených na matematických modelech a často zkracují čas potřebný k řešení (ve srovnání s metodou úplného výběru možných alternativ).

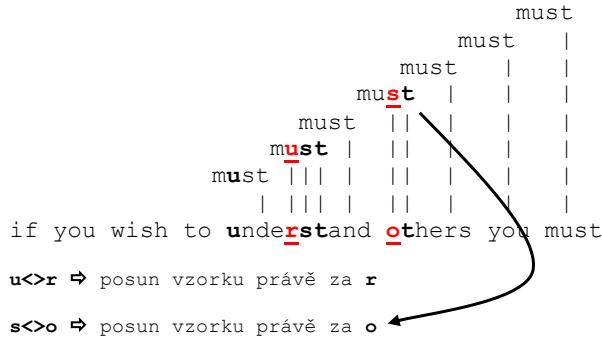
Princip první heuristiky ilustruje následující příklad postupu při vyhledávání vzorku "must" v prohledávaném textu.

Příklad:

|     |                                                                                                                                          |
|-----|------------------------------------------------------------------------------------------------------------------------------------------|
| x+y | <pre>         must       must           must               must                         if you wish to understand others you must </pre> |
|-----|------------------------------------------------------------------------------------------------------------------------------------------|

Když shledáme, že  $t <> y$ , můžeme také konstatovat, že vzorek neobsahuje žádné  $y$ , a proto můžeme posunout začátek vzorku o 4 místa dále. Podobně je tomu u dvou dalších porovnávání. Při dalším porovnání dochází na konci opět k nerovnosti, ale řetězec na testovaném místě obsahoval 'u', což je znak vzorku. Proto se posune

vzorek tak, aby souhlasila poloha znaku u. Dále bude porovnávání pokračovat takto:

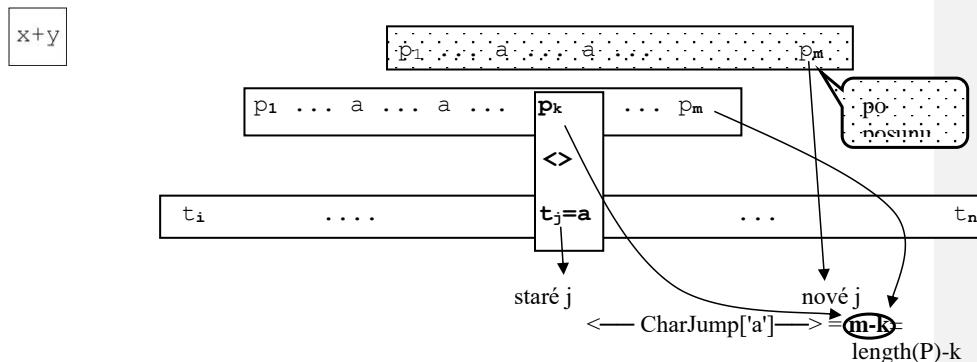


(jak je u BMA obvyklé, vzorek se začíná porovnávat **zprava**). Po neúspěchu porovnání při 'u' (must) a 'r' (understand) se posune vzorek tak, aby právě minul 'r'. Podobně tomu bude s 's' (must) a 'o' (others). Poslední ukázané porovnání je neúspěšné, ale znak 'u' je součástí vzorku. Následující porovnání bude úspěšné a výsledkem je nalezení zadaného vzorku.

V tomto příkladu se provede pouze 18 porovnání znaků. Protože k nalezení vzorku dojde až na 38 pozici, provedou ostatní algoritmy nejméně 41 porovnání.

Analýza:

Počet pozic, o které lze "skočit" dopředu, dojde-li k nesouhlasu porovnávaného vzorku závisí na znaku, který označme  $t_j$ . Hodnoty těchto počtů můžeme uložit do pole (označme ho identifikátorem CharJump), které bude indexováno typem "znak" a bude mít tedy počet prvků shodný s počtem prvků použité abecedy. Pro řízení prohlížecího algoritmu je pohodlnější uchovávat hodnotu, o kterou se má zvýšit index  $j$ , od něhož se zahají testování ve směru zprava-doleva, než počet pozic, o který se vzorek posouvá podél prohledávaného textu. Z předchozího příkladu je vidět, že když se  $t_j$  nevyskytne ve vzorku  $P$ , lze poskočit o " $m$ " pozic. Na následujícím obrázku je vidět, jak lze vypočítat skok v případě, že se  $t_j$  ve vzorku nachází. Ve skutečnosti se  $t_j$  (v příkladě znak 'a') může ve vzorku vyskytnout vícekrát. V tom případě provedeme nejménší možný skok, odvozený od nejpravějšího výskytu znaku  $t_j$  ('a') ve vzorku. (Pro algoritmus je typické, že vzorek se nikdy neposunuje podél textu ve zpětném směru; je-li aktuální pozice v  $P$  vlevo od nejpravějšího výskytu  $t_j$  v  $P$ , nepřináší již CharJump[ $t_j$ ] žádný užitek).





Algoritmus pro výpočet skoku (posuvu) vzorku pro BMA.

Aktivní znalost následujícího algoritmu nepatří k látce požadované ke zkoušce.

```

type
 TCharJump=array[char] of integer;

procedure ComputeJumps (P:TString; var CharJump:TCharJump);
(* stanovení hodnot pole CharJump, které určují posuv vzorku *)
var
 ch:char;
 k:integer;
begin
 for ch:=chr(0) to chr(255) do begin
 CharJump[ch]:=P.length;
 (*for *)
 end; (*for *)
 for k:=1 to length(P) do begin
 CharJump[P[k]]:= length(P)-k
 end; (*for *)
end; (* procedure *)

```

## Druhá heuristika

Již použití pole CharJump pro postup vzorku podél textu činí BMA podstatně rychlejší než algoritmus KMP. Přesto lze použít další myšlenky:

Předpokládejme, že nejpravější úsek vzorku P souhlasí s odpovídajícím úsekem v textu a teprve vlevo od tohoto úseku dochází k nesouhlasu:

Kombinace 'ats' se ve vzorku ale vyskytuje dvakrát! Vzorek P můžeme posunout tak, aby kombinace 'ats' stály proti sobě.

|    |                          |
|----|--------------------------|
| P: | b   ats   andcats        |
|    | ^                        |
| T: | ... d   ats   ... nové j |

Je-li  $r$  nejlevější index, pro který platí:  
 $(p_r \dots p_{r+m-k-1}) = (p_{k+1} \dots p_m)$  a současně  $p_{r-1} \neq p_k$

pak pro prvek pomocného pole platí: **MatchJump[k]=m-r+1**.

Pole MatchJump je definované pro daný vzorek P a určí se dále popsáným algoritmem.



Znalost následujícího algoritmu nepatří k látce požadované ke zkoušce.

```

procedure ComputeMatchJump(P:string; var MatchJump:TMatchJump);
(* procedura pro výpočet pole MatchJump, m *)
var
 k,q,qq:integer;
 Backup:TMatchJump;
begin
 for k:=1 to m do MatchJump[k]:=2*m-k; (* největší možný skok posunu *)
 k:=m; q:=m+1;
 while k>0 do begin
 Backup[k]:=q;
 while (q<=m) and (P[k]<>P[q]) do begin
 MatchJump[q] := min(MatchJump[q],m-k);
 q:=Backup[q];
 end; (* while (q<m).. *)
 k:=k-1;
 q:=q-1
 end; (* while k>0 *)

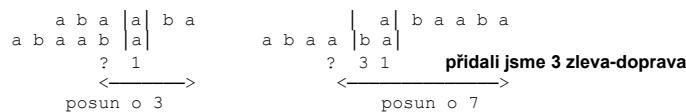
 for k:=1 to q do MatchJump[k]:= min(MatchJump[k], m+q-k);

 qq:=BackUp[q];
 while q<m do begin
 while q<=qq do begin
 MatchJump[q]:=min(MatchJump[q], qq-q+m);(* funkce vrací menší ze dvou *)
 q:=q+1
 end; (* while q<qq *)
 qq:=BackUp[qq]
 end; (* q<m *)
end; (* procedure ComputeMatchJump *)

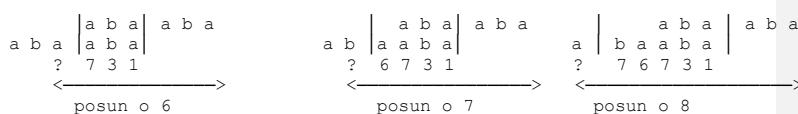
```



#### Příklad pro výpočet MatchJump ilustrujme ukázkou pro vzorek 'abaaba'



Všimněme si, že první 'ba' a druhé 'ba' není použito, protože obě předchází 'a' a nedochází tedy k nesouhlasu na pozici před příponou. Dojde-li k nesouhlasu na 4. pozici vzorku, neexistuje žádná poloha pro zarovnání s jiným 'a' vzorku, než s prvním.



Výsledkem pro řetězec P: a b a a b a  
je pole MatchJump: 8 7 6 7 3 1

Boyer-Mooreův algoritmus prochází textem (porovnává vzorek zprava doleva) a pro

posuv bere ten z výsledků dvou heuristik, který je výhodnější.

```
Boyer-Mooreův algoritmus
function BMA(P,T:string; CharJump:TCharJump;
MatchJump:TMatchJump) :index;
(* funkce pro stanovení indexu polohy vzorku nalezeného v daném textu *)
var
 j, (* j je index do textu *) k (* k je index do vzorku *) :index;
begin
 j:=length(TP); k:=length(P);
 while (j<=length(T)) and (k>0) do begin
 if T[j]=P[k]
 then begin
 j:=j-1; k:=k-1
 end else begin
 j:=j+max(CharJump[T[j]], MatchJump[k]);
 (* funkce max vybere větší - výhodnější posun *)
 k:=length(P)
 end; (* if *)
 end; (* while *)
 if k=0
 then BMA:=j+1 (* našla se shoda *)
 else BMA:=length(T)+1; (* shoda se nenašla *)
end; (* function *)
```



Závěr

Chování BMA závisí na kardinalitě abecedy a na opakování podřetězců ve vzorku. Empirické studie s hovorovým jazykem ukázaly, že pro délku vzorku  $m > 5$  provádí algoritmus přibližně 0.24 až 0.3 porovnání z počtu znaků v prohledávaném textu. Jinými slovy, porovnává asi jednu čtvrtinu až jednu třetinu znaků prohledávaného textu. BMA patří k velmi efektivním algoritmům pro vyhledávání podřetězců v řetězcích

## 7. Rekurze

### Cíl kapitoly

Cílem kapitoly je vysvětlit některé základy tvorby a zápisu algoritmů s využitím rekurze a na vhodných příkladech demonstrovat jejich použití.

Rekurze je významný nástroj zápisu algoritmu. Je komplementem k iteraci (cyklu). Podobně jako cyklus zkracuje zápis opakované části kódu. Jsou programovací jazyky i styly, které jsou na rekurzi založeny (LISP, Prolog) nebo rekurzivní zápis podporují a je řada programátorů i aplikací, které se jí vyhýbají. Rekurze umožňuje elegantní zápis algoritmu zejména tam, kde je algoritmus odvozen z definicí, které jsou samy rekurzívny. Správnost algoritmu je pak dosažena správností popisu definice do kódu. Kurz IAL si klade za cíl seznámit studenty s rekurzivním i nerekurzivním tvarem algoritmu všude tam, kde je to vhodné.

### Princip implementace rekurze

Na úrovni strojově orientovaného jazyka je rekurze implementována podprogramem, který dovoluje, aby byl znova vyvolán dříve, než je ukončen. To znamená, že se v okamžiku rekurzivního volání musí zaznamenat plný stav (stavový vektor) procesu podprogramu, který byl dosažen, aby mohl být při návratu znova aktivován a přerušený podprogram mohl pokračovat. Stav podprogramu uchovávají především lokální proměnné. Stav se uchovává v paměti zásobníkového typu, na jehož vrchol se při každém rekurzivním volání uloží stavový vektor. Takovému volání se také říká "zanoření" rekurze. Každé ukončení podprogramu je provázeno odstraněním stavového vektoru z vrcholu zásobníku a aktivací nového stavového vektoru, který obnoví stav opuštěný zanořením. Ukončení podprogramu se také říká "vynoření" rekurze.

### Rekurzivní struktura

O struktuře se říká, že je rekurzivní jestliže je definována sama sebou, nebo jestli částečně obsahuje sama sebe. Rekurze je zvlášť významná v matematických definicích. např.:

- Přirozená čísla:  
 a) 1 je přirozené číslo  
 b) každý následník přirozeného čísla je přirozené číslo

Faktoriál:

- a)  $0!=1$   
 b) pro  $n > 0$  je  $N!=n^*(n-1)!$

Mocnost rekurze spočívá ve schopnosti definovat nekonečnou množinu objektů konečným popisem a to i v případě, že ve struktuře není explicitně uvedena iterace.

Hovoříme-li o rekurzivní struktuře, pak to může být jak řídicí struktura tak datová struktura. Řídicí rekurzivní strukturu představuje rekurzivní procedura. Datovou rekurzivní strukturu představuje dynamická struktura, jejíž prvek může ukazovat sám na sebe nebo na jiný prvek téhož typu.

### Efektivní

Funkce  $f(x_1, x_2, \dots, x_n)$  je **efektivně vyčíslitelná**, existuje-li neintuitivní postup

**vyčíslitelnost****DEF**

(formální postup na základě určitých pravidel), pomocí něhož můžeme stanovit hodnotu funkce  $f(k_1, k_2, \dots, k_n)$  pokaždé, jsou-li zadány relevantní parametry  $k_1, k_2, \dots, k_n$ . Takový postup se pak nazývá algoritmem.

Pozn.

Algoritmus splňuje tři základní vlastnosti: determinovanost, masovost a resultativnost. Normální algoritmy Markova vycházejí z principu přepisovacího systému, jehož zvláštním případem jsou gramatiky. Turingovy algoritmy se definují pomocí zvláštního automatu - Turingova stroje, který slouží jako model univerzálního počítače. Jinou možností vyjádření algoritmu jsou nástroje rekurzivních funkcí.

Rekurzivní vyčíslitelnost vyjadřuje Churchova teze (A. Church). Ta říká, že jakoukoli intuitivně vyčíslitelnou funkci lze popsat s pomocí rekurzivních funkcí. (Jako "teze" se to označuje proto, že nelze formálně dokázat, že vyčíslitelnost pojata intuitivně je ekvivalentní vyčíslitelnosti definované formálně). Ukazuje se, že Markovovy algoritmy (A.A.Markov), Turingovy algoritmy (Alan Turing) i rekurzivní funkce mají stejnou vyjadřovací sílu pro popis algoritmu a že Markovovy i Turingovy algoritmy lze převést na definici jisté rekurzivní funkce. Výhodou rekurzivních algoritmů je jednoduchá struktura a homogenní forma. Programování ve stylu rekurzivních funkcí se označuje také jako **funkcionální programování**.

**Vztah rekurze  
a iterace**

Použití rekurze vede obvykle k rozkladu (redukci) úlohy na podúlohy, které se řeší podobným způsobem. Po konečném počtu redukcí vzniknou úlohy, které lze řešit přímo. Řešení nadřazených úloh se získá skládáním řešení podúloh. To se děje tak dlouho, dokud není získáno řešení původní úlohy. Jde tedy o využití principu "rozděl a panuj" (lat. "*divide et impera*", angl. "*divide and conquer*").

Rekurzivní technika je vhodná tam, kde je možné úlohu rozložit v rozumném čase na dílčí podúlohy, jejichž složitost není příliš veliká. Je-li  $n$  rozměr dané úlohy a součet rozměrů částečných úloh je  $an$ , pro nějaké konstantní  $a > 1$ , pak má rekurzivní algoritmus polynomiální složitost. Jestliže však rozklad úlohy rozměru  $n$  vede na  $n$  částečných úloh rozměru  $n-1$ , má rekurzivní algoritmus exponenciální složitost.

Jedním z hlavních problémů rekurze je tedy otázka efektivnosti výpočetního postupu. Pojem rekurze se vyskytuje také v souvislosti s datovými strukturami. Kromě velké složitosti časové je s podstatou rekurzivního algoritmu spjata také jeho složitost prostorová (paměťová). Ta plyne z toho, že rekurzivní výpočetní postup po návratu z hlubší úrovni může potřebovat hodnoty dat méně hluboké úrovně. V takovém případě je tedy zapotřebí pro každou úroveň rekurze, kterou výpočet prochází, založit v paměti nový exemplář zpracovávaných datových struktur.

Z výše uvedeného vyplývá, že zkoumání vztahu mezi rekurzí a iterací má základní důležitost z hlediska optimalizace výpočetních postupů definovaných pomocí rekurze. Je nutno se zabývat jak samotným problémem zamezení opakování vyhodnocování, tak převodem rekurzivních algoritmů na iterativní.

Dá se ukázat, že využití iterativní řídicí struktury při výpočtu rekurzivně definované funkce spočívá ve využití tzv. **akumulačního parametru** (akumulátoru), v němž se shromažďuje informace postupně získaná na jednotlivých úrovních rekurze. Nově získaná informace může přitom překrýt předcházející informaci a tak velikost požadované paměti je určena pouze rozsahem posledně vytvořené informace, protože informace z různých úrovní rekurze není třeba ulákat "vedle sebe" do zásobníku.

Viz příklad výpočtu faktoriálu zapsaného iterací.

### Konečnost rekurze

Podobně jako cyklus, který konečným zápisem umožňuje popis nekonečného procesu či struktury, i rekurze musí řešit v praxi problém konečnosti. Obecně může být rekurze definována jako kompozice sekvence příkazů S neobsahující rekurzivní operaci P a operace P samotné, tedy

$$P = n[S, P]$$

kde n je úroveň zanoření rekurze.

Schéma rekurzivního algoritmu, zabezpečujícího konečnost je pak

$$P = \text{if } B \text{ then } n[S, P]$$

nebo  $P = n[S, \text{if } B \text{ then } P]$

Má-li se zajistit konečnost, musí se zabezpečit, aby došlo k situaci, v níž **B=false**. Je-li možno zabudovat do rekurzivního algoritmu celočíselnou proměnnou **n>0**, pak lze rekurzivní funkci zapsat:

$$P(n) = \text{if } n > 0 \text{ then } n[S, P(n-1)]$$

nebo  $P(n) = n[S, \text{if } n > 0 \text{ then } P(n-1)]$

**Použití rekurze** Rekurzivní algoritmus se nejlépe využije tam, kde řešený problém sám je definován rekurzivně. To ale neznamená, že odpovídající rekurzivní algoritmus je nejlepším řešením daného problému.

Program, v němž je vhodné se vyhnout rekurzi má tvar:

$$P = \text{if } B \text{ then } (S; P) \quad (\text{A})$$

nebo  $P = (S; \text{if } B \text{ then } P)$

Pozn. "P =" .... rozumíme - "P je definováno jako" ....

Pro faktoriál lze tento problém zapsat takto:

$$\begin{aligned} P = & \text{ if } i < n \text{ then } (i := i + 1; F := i * F; P) \\ & i := 0; F := 1; P \end{aligned}$$

První řádek lze přepsat do procedury:

```
procedure P;
begin
 if i < n
 then begin
 i := i + 1; F := i * F; P
 end
end;
```

Častější, ale v podstatě ekvivalentní, je použití funkce, v níž proměnná **i** hráje roli parametru a F je přímo jménem funkce, takže P není potřebné. Následující ukázka počítá hodnotu faktoriálu.

```
x+y
function F(i:integer):integer;
begin
 if i>0
 then F:=i*F(i-1)
 else F:=1
end
```

V tomto případě je vhodné rekurzi nahradit iterací:

```
i:=0; F:=1;
while i<n do
begin
 i:=i+1; F:=i*F
end
```

Obecně lze programy zapsané schématem **(A)** zapsat  
**P = (x:=x0; while B do S)**

Ještě výraznější a složitější příklad lze uvést v souvislosti s výpočtem členu Fibonacciho posloupnosti:

```
x+y
function Fib(n:integer):integer;
begin
 if n=0
 then Fib:=0
 else if n=1
 then Fib:=1
 else Fib:=Fib(n-1) + Fib(n-2)
end
```

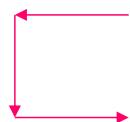
## 7.1. Hilbertovy křivky

### Hilbertovy křivky.

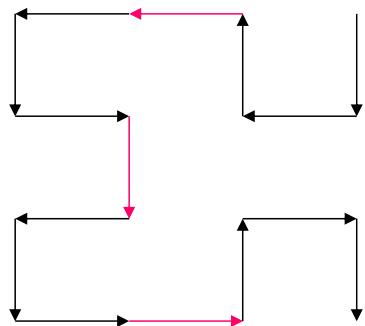
Zajímavé rekurzivní algoritmy se váží ke grafickým problémům. Jedním z nich jsou tzv **Hilbertovy křivky**.

Hilbertova křivka řádu **i>1** je spojením čtyř Hilbertových křivek řádu **(i-1)** o poloviční velikosti spojkou a s odpovídající rotací křivek řádu **(i-1)**.

Hilbertova křivka prvého řádu H1 – čtyři Hilbertovy křivky 0-řádu (príslušně natočené do jedné ze čtyř vzájemně pravoúhlých pozic) jsou spojeny (červenými) spojkami



Hilbertova křivka 2. řádu (H2) je složena ze čtyř Hilbertových křivek 1. řádu (H1). Ty jsou natočené podle definice a jsou spojeny (červenými) spojkami



Rekurzivní schéma je dáno popisem:

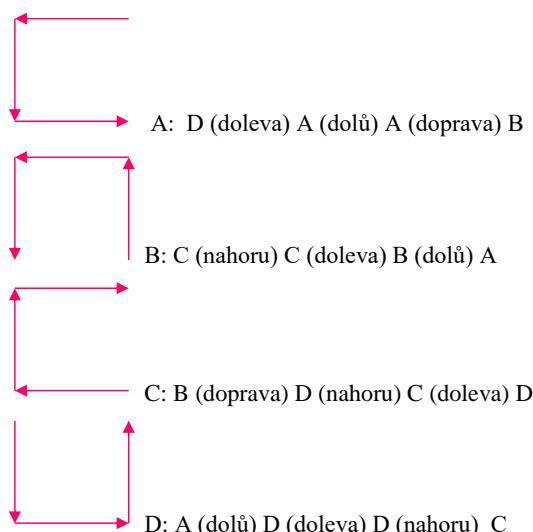


Schéma Hilbertovy křivky A je popsáno procedurou:

```
procedure A(i:integer);
begin
 if i > 0
 then begin
 D(i-1); Line(x,y,x-h,y); x:=x-h;
 A(i-1); Line(x,y,x,y-h); y:=y-h;
 A(i-1); Line(x,y,x+h,y); x:=x+h;
 B(i-1)
 end
end;
```

Podobným způsobem lze definovat i procedury B,C a D. Program pro vytvoření Hilbertových křivek rádu až **n** má tvar:

```
program Hilbert;
const
 n=4; (* řád křivky *)
 h0=512; (* počet pixelů základní spojky*)
var
 i,h,x,y,x0,y0:integer;
procedure A(i:integer);
begin end;
procedure B(i:integer);
begin end;
procedure C(i:integer);
begin end;
procedure D(i:integer);
begin end;
begin
 i:=0; h:=h0; x0:=h div 2; y0:=x0;
 repeat (* Kresli křivku o řádu i *)
 i:=i+1; h:=h div 2;
 x0:=x0 + (h div 2);
 y0:=y0 + (h div 2);
 x:=x0; y:=y0;
 A(i)
 until i=n;
end.
```

## 7.2. Algoritmy s návratem: "Cesta koně" a "Osm dam"

7.2. Algoritmy s návratem: "Cesta koně" a "Osm dam".  
 Pro algoritmy s návratem je typické, že řešení se nehledá předpisem pro jeho dosažení, ale metodou pokusů a omylů. Na této cestě, na které je mnoho "křízovatek", se vydáme vždy prvním z několika možných směrů a zaznamenáme si cestu. Jsme-li donuceni vrátit se do tohoto místa zpět v důsledku neúspěchu, volíme příště další z možností směrů. Algoritmus skončí neúspěšně, vyčerpají-li se všechny možnosti a nedojde-li ke splnění podmínek, které řešení vyžaduje.

Dva typické příklady algoritmů této třídy jsou spojeny se šachovnicí. Jsou to "**Cesta koně**" a "**Osm dam**". Algoritmus "Cesta koně" je nalezení posloupnosti polí na šachovnici, kterými šachová figurka "kůň" projde celou šachovnici počínaje startovacím polem tak, aby prošel všemi polí a žádným z nich dvakrát.

"Osm dam" je nalezení takového umístění osmi dam na šachovnici o rozměrech **8\*8**, aby se podle pravidel šachu vzájemně neohrožovaly.

Pro algoritmus s návratem nazvaným "Cesta koně" lze definovat pokus o další

tah koně s následujícím schématem:

```
procedure PokusODalsiTah;
begin "inicializace výběru tahu";
repeat
 "vyber následující z možných tahů"
 if "tah je přijatelný"
 then begin
 "zaznamenej tah";
 if "šachovnice není vyčerpaná"
 then begin
 PokusODalsiTah; (* rekurzivní volání*)
 if "pokus se nezdařil"
 then "vymaž zaznamenaný tah"
 end (* if „šachovnice ...“ *)
 end (* if „tah je přijatelný“ *)
 until("tah byl úspěšný") or ("není další možný tah")
 end;
```

Schéma řešení problému problém Osmi dam lze rekurzivně zapsat takto:

```
procedure zkus (i:integer);
begin
 "inicializuj výběr pozice pro i-tou dámou";
repeat
 "vyber další pozici";
 if "dáma nikoho neohrožuje"
 then begin
 "ustav dámu";
 if i<8 then begin
 zkus(i+1);
 if "nepodařilo se"
 then "odstraň dámu"
 end (* if i<8 *)
 end (* if „dáma nikoho ...“ *)
 until "podařilo se" or "není další pozice pro dámu"
 end
```

### Případová studie

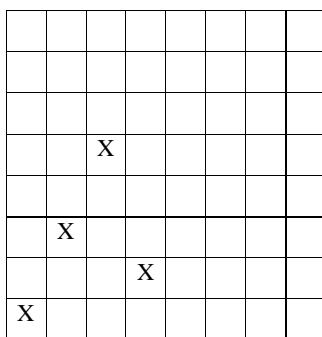
Nerekurzivní schéma řešení problému 8 dam lze popsat takto:

- Šachovnice 8x8 je reprezentovaná celočíselným vektorem o osmi prvcích. Hodnota i-té položky vektoru z intervalu 1..8 představuje rádek, v němž je na v i-tém sloupci umístěna dáma.
- Před zahájením je první položka nastavena na hodnotu 1. (Dáma je umístěna v prvním sloupci do prvního rádku).
- Algoritmus se snaží umístit dámu v dalším sloupci počínaje prvním rádkem směrem nahoru tak, aby se neohrožovala s dámami umístěnými v předcházejících sloupcích (kontroluje shodu rádku a umístění dam na obou diagonálách v předchozích sloupcích). Když najde rádek, do něhož lze umístit dámu, přechází na další sloupec a opakuje pokus o umístění dámy. Když se v

daném sloupci nenajde žádný řádek, v němž by se dáma neohrožovala s žádnou z dam v předcházejících sloupcích, vrací se algoritmus o jeden sloupec níž (vlevo) a pokouší umístit dámu na některém z dalších řádků.

d) Krok (c) se opakuje; když se podaří umístit úspěšně dámou v osmém sloupci, nalezlo se jedno řešení. Algoritmus končí (po nalezení všech řešení) když dojde k pokusu o umisťování dámy v nultém (neexistujícím) sloupci.

Stav šachovnice po umístění dámy ve 4. sloupci.



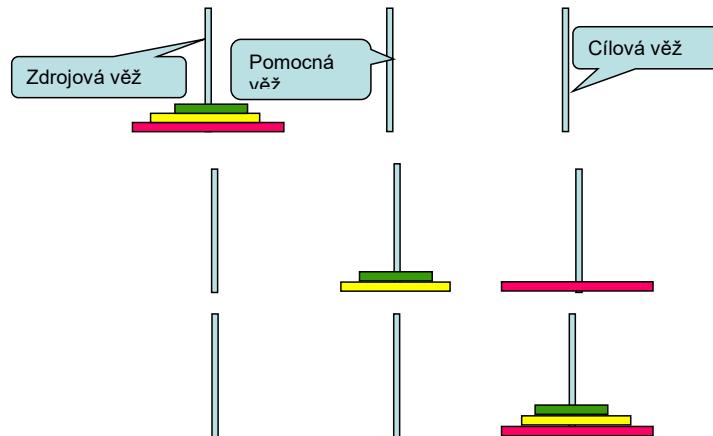
Hodnoty vektoru reprezentujícího šachovnici po umístění dámy ve 4. sloupci.

|   |   |   |   |  |  |  |  |
|---|---|---|---|--|--|--|--|
| 1 | 3 | 5 | 2 |  |  |  |  |
|---|---|---|---|--|--|--|--|

Úloha: Napište program pro nerekurzívní řešení problému 8 dam.

### 7.3. Hanojské věže

Jedním z nejznámějších a nejpopulárnějších algoritmů, demonstrujících princip rekurze, je algoritmus staré hry zvané "Hanojské věže". Hanojské věže jsou reprezentovány třemi tyčemi a kotouči různého průměru, které jsou na tyči nasunuty vždy tak, že kotouč s menším průměrem je umístěn nad kotoučem s vyšším průměrem. Úkolem je přesunout věž z kotoučů ze zdrojové věže na cílovou věž tak, že žádny kotouč nesmí být odložen jinam, než na pomocnou tyč.



Algoritmus rekurzívní verze pro věž o výšce  $N$  lze slovně popsat takto: Pokud je výška věže větší než 1, provedou se tři kroky: V prvním kroku se na pomocnou tyč přesune jako celek věž o výši  $N-1$ . Na zdrojové tyče zůstane nejnižší kotouč s největším průměrem. (Tento krok reprezentuje rekurzi). V dalším kroku se kotouč přesune ze zdrojové tyče na cílovou tyč. Tento krok dokumentuje pohyb kotoučů a jeho výpisem nebo grafickým zobrazením lze sledovat řešení na výstupu. Třetím krokem je přesun věže z pomocné tyče na tyč cílovou.

Algoritmus je popsán následující procedurou:

```
procedure PRESUNVEZ (Vyska:integer; var Odkud,Kam,
 Pom: integer);
begin
 if Vyska>0
 then begin
 PRESUNVEZ (Vyska-1,Odkud,Pom,Kam);
 Presundisk (Odkud,Kam);
 PRESUNVEZ (Vyska-1,Pom,Kam,Odkud)
 end
end;
```

Pozn. Funkce tyče je dána pořadím v seznamu parametrů, daným v hlavičce deklarace procedury.

Ukázková varianta nerekurzívní verze algoritmu "Hanojských věží" je velmi podobná nerekurzívní verzi průchodu stromem.

```
procedure Strkej (V,O,K,P:integer);
var Pom:integer ;
begin
 while V<>0 do begin
 Push (V,O,K,P); K:=:P; V:=V-1
```

```

 end
end;

procedure PRESUNVEZ (V,O,K,P:integer);
begin
 StackInit;
 Strkej (V,O,K,P);
 while not SEmpty do begin
 Pop (V,O,K,P);
 PrenesDisk (O,K);
 Strkej (V-1,P,K,O)
 end (* while *)
end;

```

**Σ Shrnutí a závěr** Rekurze je jedním z velmi významných nástrojů zápisu programu. Z teoretického hlediska má nezastupitelnou roli při definici a zápisu algoritmů, při dokazování správnosti programů a v řadě jiných oblastí, které vyžadují formální matematickou notaci a postupy. V praxi má řadu příznivců především mezi uživateli, kteří prošli formálním vzděláním v oblasti počítačově orientované algebry, teorie informatiky, teorie formálních jazyků aj. a má i řadu odpůrců. zejména v řadách ,byť úspěšných, samouků a praktiků. Složitější a rozsáhlější rekurzívni zápis mohou být náročnější na pochopení a jejich programy mohou být obtížnější laditelné a sledovatelné (trasovatelné). Nástroje na ladění programů nejsou vždy vybaveny pro práci s rekurzivním zápisem algoritmu. V řadě základních prototypových algoritmů však rekurze umožňuje vytvářet krystalicky srozumitelné algoritmy a správné programy a programátorky, který neovládne základy rekurze nepřekročí ochotnický rámec programátorské činnosti.

## 8. Dokazování správnosti programu

Cílem kapitoly je seznámit čtenáře se základy formálních nástrojů důkazu správnosti algoritmu a programu. Neočekává se, že by student prováděl samostatné náročnější dokazování programu, ale seznámení s postupy důkazu a s metodami dokazování umožní studentu hlubší poznání tvorby správných programů.

Je významnou teoretickou oblastí algoritmizace. Nelze očekávat, že by praktický programátor používal nástroje a metody dokazování při každodenní tvorbě programů. Znalost zásad dokazování správnosti a vlastností správných programů však významně přispívají k tvorbě správných programů. Správnost je jednou z několika významných vlastností, které činí program úspěšným z hlediska jeho prodeje jako zboží a z hlediska jeho používání po dobu jeho života.

Neformálně lze říci, že program je správný, když plně vyhovuje svému účelu. Je-li účel popsatelný formálními prostředky, je možné formálně prokázat, že sémantika programu naplňuje formálně popsany výsledek (účinek) programu.

Vzájemné vazby a souvislosti rozsáhlého programu mohou vytvářet natolik

složitý systém, že formální postup řešící správnost takového programu by překročil dostupné možnosti lidského řešitele. Proto se dokazování správnosti programu soustředí na stavebnicové prvky (celky), z nichž se skládá složitý systém a které lze z hlediska algoritmizace považovat za základní.



Pozn. "Správnost" programu může mít i ekonomický aspekt. Jsou-li náklady na odstranění nesprávnosti programu způsobené za dobu života programu větší, než "škody", které nesprávnost způsobí, může se uživatel rozhodnout, že bude takový program považovat za "správný" pro dané použití. Úsměvným příkladem je vzpomínka na jeden z prvních velkých programů řešících v polovině sedmdesátých let mzdy velkého podniku. Tento program selhal v případě několika zaměstnanců, jejichž alimentační povinnosti překročily 10 případů. Protože šlo tehdy o importovaný systém bez dokumentace, bylo jednodušší vyjmout tyto pracovníky ze souboru tisíců zaměstnanců a zpracovat jejich mzdu ručně, než se pokoušet zasahovat do systému, k němuž nebyly ani zdrojové informace, ani podrobnější dokumentace.

Následující odstavce představují elementární principy dokazování správnosti programu tak, jak je jako jeden z prvních uvedl Nielaus Wirth ve své knize "Systematic programming" (Prentice Hall, Englewood Cliffs, 1973).

Základním principem dokazování je stanovení oborů hodnot proměnných a vztahů mezi proměnnými pro každý příkaz programu. Vztahy se vyjadrují tvrzeními (logickými výroky), vztahujícími se k jednotlivým místům programu a nezávislými na cestě, po níž se k danému místu postup v algoritmu dostane. Pro tato stanovení platí několik základních pravidel:

**Pravidlo 1:** Pro každý příkaz algoritmu jsou nalezeny podmínky - tvrzení, které platí před a po provedení příkazu **S** (*statement*). Tvrzení platné před příkazem se nazývá antecedence P (*precondition*) a po provedení příkazu se nazývá konsekvence Q (*postcondition*).

...  
(\* Platí antecedence **P** \*)  
**S**;  
(\* Platí konsekvence **Q** \*)

Je-li antecedence příkazu **S** tvrzení **P** a jeho konsekvence **Q**, pak můžeme toto tvrzení zapsat notací **(S,P)=>Q**.

**Pravidlo 2:** Spojuje-li se před příkazem **T** několik větví algoritmu, pak konsekvence všech předcházejících příkazů **S<sub>i</sub>** |(1 < i < n) musí logicky implikovat antecedenci následujícího příkazu **T**.

...  
case i of  
 1: **S<sub>1</sub>**; (\* platí konsekvence **Q<sub>1</sub>** \*)  
 2: **S<sub>2</sub>**; (\* platí konsekvence **Q<sub>2</sub>** \*)  
 ...  
 n: **S<sub>n</sub>**; (\* platí konsekvence **Q<sub>n</sub>** \*)  
end (\* case \*);  
 (\* Platí **Qi=>P** |(1 < i < n), kde P je antecedence příkazu **T** \*)  
**T**;  
...

**Pravidlo 3:** Platí-li před podmíněným příkazem s podmínkou **B** tvrzení **P**, pak konsekvence příkazu za "then" je "**B and P**" a konsekvence příkazu za

"else" je "**not B and P**".

Příklad:

```
x+y
 ...
(* P = (-10 < x < 10) *)
if x < 0
then (* Qthen = (-10 < x < 0) *) ... Sthen
else (* Qelse = (0 < x < 10) *) ... Selse
...
...
```

**Pravidlo 4:** Pro přiřazovací příkaz **v:=e** platí: Nechť **P<sub>ev</sub>** je antecedence příkazu **v:=e** (**e** - expression, **v**- variable). Konsekvenci tohoto příkazu dostaneme tak, že každý výskyt výrazu **e** v antecedenci nahradíme proměnnou **v**.

Inverzní pravidlo říká, že je-li **Q** konsekvence přiřazovacího příkazu **v:=e**, pak jeho antecedenci získáme náhradou každého výskytu proměnné "v" v konsekvenci výrazem "e". Ilustrace pravidla:

Pozn. Všimněme si, že v antecedenci děláme úpravu některé z rovnic tak, abychom si připravili pro konsekvenci substituci výrazu **e** za proměnnou **v**. Postupné substituce jsou v následujícím příkladě zdůrazněny podtržením, zvýrazněním i barvou.

```
(* P1: y=x ; d= 2x-1; => d+2=2x+1 *)
d := d+2;
(* Q1=P2: y=x ; d=2x+1; => y+d=x +2x+1=(3x+1) *)
y := y+d;
(* Q2=P3: d=2x+1; y=(3x+1) ; => d+2=2x+3 *)
d := d+2;
(* Q3=P4: y=(3x+1) ; d=2x+3; => y+d=(3x+1) +2x+3=(5x+4)
y := y+d;
(* Q4: y=(5x+4) ; d=2x+3; *)
```

Aplikujme tato pravidla na známé algoritmy pro násobení a dělení přirozených čísel pomocí sčítání a odčítání:

Násobení **z=x\*y**

Jaký má vliv zvýšení činitele **y** o hodnotu **z** ?

```
(* P1: (z+u*y=x*y; u>0;) => z+y+(u-1)*y=x*y *)
z := z+y;
(* Q1=P2: (z+(u-1)*y=x*y; (u-1)>0 *)
u := u-1;
(* Q2: (z+u*y=x*y; u>0)
```

Pozn. Konsekvence **Q<sub>1</sub>** se pro další příkaz stává antecedencí **P<sub>2</sub>**.

Přepsáno do celkového tvaru algoritmu pro násobení:

```

(* x>0; y>0 *)
z:=0;
u:=x;
(* z=0; u=x *)
repeat
 (* z+u*y=x*y; u>0 *)
 z:=z+y;
 u:=u-1;
until u=0;
(* z=x*y; u=0 *)
Q.E.D. (lat. Quod erat demonstrandum) - "což bylo dokázati".

```

Poslední konsekvence je výrok přesně popisující výsledný součin.

Podobně lze dokázat správnost algoritmu dělení  $q = x \text{ div } y$ , kde pomocná proměnná  $r$  nabude v závěru algoritmu významu "zbytku po dělení".

```

(* P1: (r+q*y=x; (r-y)=>0) => (r-y)+(q+1)*y=x *)
r:=r-y;
(* Q1=P2: (* r+(q+1)*y=x; r=>0 *)
q:=q+1;
(* Q2 : (* r + q*y = x; r=>0 *)

```

Přepsáno do celkového tvaru algoritmu

```

(* x>0, y>0 *)
q:=0;
r:=x;
(* q=0; r=x *)
while r>y do begin
 r:=r-y;
 q:=q+1;
 (* q*y+r=x; r=>0 *)
end; (* while *)
(* q*y+r=x; 0<=r<y *) Q.E.D.

```

Pozn. Všimněme si, že zbytek po dělení - jinak výsledek operace ( $x \text{ mod } y$ ) - je souběžným produktem operace dělení.

**Pravidlo 5a:** Nechť je dáno tvrzení **P**, které je invariantní (neměnné) vzhledem k příkazu **S** (to znamená, že provedení příkazu **S** nemá na tvrzení **P** žádny vliv; tvrzení je současně antecedencí i konsekvencí příkazu). Pak pro příkaz cyklu **S'** typu "while **B** do", jehož vnitřním příkazem je příkaz **S** platí konsekvence ve tvaru **P and not B**.

$$(S', P) \Rightarrow P \text{ and not } B$$

```
(* P *)
while B do begin (* příkaz S *)
 S
end (* while *)
(* P and not B *)
```

**Pravidlo 5b:** Nechť pro příkaz S platí dva předpoklady:

$$\begin{aligned} (S, P) &\Rightarrow Q \\ (S, Q \text{ and not } B) &\Rightarrow Q \end{aligned}$$

Pak příkaz cyklu S" typu "repeat ... until B", který obsahuje vnitřní příkaz S, má antecedenci "P" a konsekvenci "Q and B"

$$(S'', P) \Rightarrow Q \text{ and } B$$

```
(* P *)
repeat
 S
until B;
(* Q and B *)
```

S ohledem na skutečnost, že pro cyklus typu repeat-until se vyžaduje platnost obou předpokladů (P) a také (Q and not B), bývá tento cyklus častěji zdrojem chyb než cyklus typu while, i když jeho použití někdy vede ke kratšímu zápisu.

**Invariant cyklu** je podmínka, která se nemění v průběhu cyklu. Invariant cyklu důležitý nejen pro důkaz správnosti algoritmu, ale při slabším využití i pro stanovení podmínek konečnosti cyklu. Minimální podmínkou konečnosti cyklu je požadavek, aby příkaz S měnil hodnotu alespoň jedné proměnné, na niž závisí hodnota B.

Obecně se konečnost cyklu zajistí tak, že se do těla cyklu zavede celočíselná funkce N taková, že  $B = N > 0$ . Je-li na počátku cyklu hodnota funkce  $N > 0$  a každé provedení příkazu uvnitř cyklu sníží hodnotu funkce N o jedničku, je konečnost cyklu zaručena, pokud cyklus končí splněním podmínky  $N=0$ .

## 8.1. Správnost algoritmu indukce

**8.1. Správnost algoritmu indukcí (Correctness by induction).** Následující odstavec je převzat z publikace "Baase,S.: Computer Algorithms, Introduction to Design and analysis. Second edition, Addison-Wesley, 1988, str. 15 - 17".

Pro ověření správnosti programu je třeba udělat 3 kroky:

- 1) definovat, co rozumíme správností,
- 2) definovat vstupní data a požadovaný výsledek na výstupu,
- 3) dokázat, že algoritmus produkuje z definovaných dat požadované výsledky.

Algoritmus sám má dvě významné stránky:

- 1) metoda zvolená k řešení
- 2) posloupnost akcí (příkazů), které metodu realizují

Důkaz správnosti metody může být velmi složitý a nespadá vždy do problematiky algoritmizace (ale např. do oblasti numerické matematiky apod.)

Zvolenou metodu implementujeme programem. Je-li program krátký a jasný, stačí ke stanovení správnosti neformální prostředky. Některé detaily prověříme pečlivěji (např. počáteční a koncové hodnoty počítadel cyklu). Tyto metody nemají charakter důkazu, ale pro malé programy mohou být postačující. Pro důkaz správnosti cyklů se často používají formálnější metody, jako stanovení invariantu cyklů matematickou indukcí.

Invariante cyklů jsou podmínky a relace, které jsou splněny proměnnými na konci každého průchodu cyklem. Invarianty cyklů se konstruují velmi pečlivě tak, aby se snadno mohlo dokázat, že se na konci cyklu dosáhlo požadovaného stavu. Invarianty cyklu se ustavují indukcí pro počet průchodů cyklem. Hlavním přínosem invariantu je skutečnost, že celková délka důkazu příkazu cyklu, který ve skutečnosti provádí opakování sekvenčního opakování, není úměrný celkovému počtu příkazů, ale je podstatně kratší!

Tuto techniku budeme ilustrovat na jednoduchém příkladu algoritmu pro sekvenční vyhledávání hodnoty nebo indexu prvku v poli. Algoritmus porovnává klíč s každou položkou, která je na řadě tak dlouho, pokud nenajde shodu nebo pokud není pole vyčerpáno. Není-li hledaná položka v seznamu, vrátí algoritmus hodnotu 0.

Algoritmus:

Vstup: **Pole, n, klíč** kde pole "Pole" má **n** prvků téhož typu jako je **klíč**.

Výstup: **Index** - umístění položky rovné klíči v poli, nebo 0 v případě nenalezení.

```

1. index:=1;
2. while (index<=n) and (Pole[index]<>klíč) do begin
3. index:=index+1;
4. end (* while *);
5. if index > n then index:=0;
```

Pozn. Předpokládejme zkrácené vyhodnocení booleovského výrazu, které zabrání referenci Pole[n+1].

Než začneme dokazovat správnost, specifikujme přesněji, co má algoritmus dělat. Požadavek může být prostý: Algoritmus určí, kolikátý prvek pole **Pole** o **n** položkách má hodnotu **klíč**. Nenalezne-li se žádný shodný prvek, je výsledná pozice rovna 0.

Tato definice má dvě vady. Nezmiňuje se jaký má být výsledek, je-li v poli více položek se shodnou hodnotou rovnou klíči a neříká, pro jaké hodnoty **n** algoritmus pracuje. Můžeme předpokládat nezáporné **n**, ale co se stane pro

$n=0$ ? Přesněji lze činnost algoritmu vyjádřit takto:

Je dáno pole **Pole** o **n** prvcích (**n>0**) a je dána hodnota **klič** stejného typu jako prvky pole. Sekvenční vyhledávání určí pořadí prvního výskytu prvku pole se shodnou hodnotou. V případě nenalezení vrátí algoritmus hodnotu 0.

**Důkaz:**

Indukcí ustavme následující výrok:

Pro  $1 \leq k \leq (n+1)$  na konci řádku 2 platí tyto podmínky (invarianty cyklu):

$$\begin{aligned} \text{index} &= k \\ (\text{ForAll } \text{index}:1 \leq \text{index} < k) [\text{Pole}[\text{index}] \neq \text{klič}] \end{aligned}$$

- a) Nechť  $k=1$ . Pak **index=1** a druhá podmínka je splněna sama o sobě.
- b) Předpokládejme, že podmínky jsou splněny pro nějaké  $k < (n+1)$  a dokažme, že výrok platí i pro  $k+1$ . Na základě předpokladu indukce platí po **k-tém** průchodu (pro  $1 \leq \text{index} < k$ ), že  $\text{Pole}[\text{index}] \neq \text{klič}$ . Provede-li se úspěšný test na řádku 2 (to znamená po **k+1** průchodech), můžeme předpokládat, že test byl **k krát** úspěšný (true) a to dále znamená, že  $\text{Pole}[\text{index}] \neq \text{klič}$  a tudíž i  $\text{Pole}[k] \neq \text{klič}$ . **Index** se ale v cyklu zvýší o 1, takže test na řádku 2 se provede **k+1** krát. To ukončuje důkaz indukcí.
- c) Nyní předpokládejme, že testy na řádku 2 se provedou právě **k** krát, kde  $1 \leq k \leq (n+1)$ . Předpokládejme dále, že na řádku 5 mohou nastat dva případy: Výsledkem je 0 pro případ, že  $k=n+1$ . Podle výroku, který jsme právě dokázali platí, že  
 $(\text{ForAll } \text{index}:1 \leq \text{index} < (n+1)) [\text{Pole}[\text{index}] \neq \text{klič}]$

a výsledek 0 je tedy správný. Stojí za povšimnutí, že výsledek je správný i pro prázdné pole, tedy pro **n=0**.

Je-li naopak **index=k<n**, pak se cyklus skončí pouze při splnění podmínky  $\text{Pole}[k] = \text{klič}$ . Tím lze považovat důkaz za splněný.

Na první pohled se zdá, že uvedený důkaz je dosti pracný a lze si představit, jak by vypadal důkaz rozsáhlejšího programu se složitějšími datovými strukturami.

#### Shrnutí

Dokazování správnosti algoritmu patří k teoretickému základu algoritmizace. Jeho praktické využití pro rozsáhlejší programu lze jen stěží předpokládat.

Jádrem řady programů je algoritmus, jehož správnost rozhoduje o funkčnosti celku. Je účelné použít mechanismů důkazu nebo postupů při dokazování správnosti takového jádra pro zajištění správné funkce celého programu.

Použití antecedence a konsekvence pro specifikaci sémantiky příkazů se používá v popisu některých programovacích jazyků.

Ovládnutí postupů dokazování správnosti algoritmů vede k dobrým návykům při intuitivním vytváření správných programů i bez formálního využití mechanismu důkazu.

#### 8.2. Dikstrova - 8.2. Dijkstrova tvorba dokázaných programů. Vytváření správných

**tvorba  
dokázaných  
programů**



programů podle Dijkstry. E.W. Dijkstra ve své knize "A Discipline of Programming" (Prentice Hall, 1976), popisuje formalizovaný postup tvorby dokázaných programů. Na rozdíl od předchozích přístupů, v nichž se dokazuje správnost intuitivně zapsaných algoritmů, je zde algoritmus, který vzniká na základě matematických transformací, ve své závěrečné podobě úplný a správný. Pro tvorbu dokázaných programů zavedl Dijkstra vlastní algoritmický jazyk a vlastní matematický aparát. Po stručném úvodu této formálních nástrojů bude uvedeno několik ilustrativních příkladů dokázaných algoritmů, které se vztahují k předmětu IAL. Podrobněji je látká popsána v kapitole 13 skript Honzík a kol.: Vybrané kapitoly programovacích technik.



Celá tato řada odpovídá svou teoretickou náročností úrovni magisterského studia. Nebude předmětem zkoušení a v opoře IAL je uvedena pro základní orientaci a jako ukázka náročnosti této disciplíny na vyšším stupni studia.

#### 8.2.1. Základní matematický aparát.

Předpokládejme, že v průběhu výpočtu největšího společného dělitele dvou pírozených čísel X,Y projdeme stavy x,y, pro něž platí:

$$\text{NSD}(x,y) = \text{NSD}(X,Y) \text{ and } (0 < x \leq X) \text{ and } (0 < y \leq Y)$$

kde NSD je označení funkce největšího společného dělitele, X,Y jsou konstanty pro určitý výpočet a určují počáteční hodnoty proměnných x,y. Podobným vztahům budeme říkat "podmínky" nebo "predikáty".

Jestliže se systém po skončení své aktivity určitě dostane do stavu splňujícího podmínuku **P** pak říkáme, že systém určitě ustaví pravdivost **P**. Každý predikát je definován v každém bodu stavového prostoru za předpokladu, že v každém bodu tohoto prostoru má hodnotu "**true**" nebo "**false**". Nadále budeme predikáty používat pro označení množiny takových bodů stavového prostoru, v nichž je predikát pravdivý.

O predikátech **P** a **Q** říkáme, že jsou si rovny ("P=Q"), jestliže označují stejnou podmínuku nebo jestliže označují stejnou množinu stavů. Dále budeme používat dva speciální predikáty s vyhrazeným označením "**T**" a "**F**". T je predikát pravdivý ve všech bodech uvažovaného prostoru. Odpovídající množinou je universum. F je predikát nepravdivý ve všech bodech uvažovaného prostoru a odpovídá mu množina prázdná.

Předpokládejme výpočetní mechanismus (dále jen mechanismus) označený **S** a podmínuku **R**, kterou musí splňovat stav mechanismu po skončení své aktivity. Podmínuku **R** nazveme "konečná podmínka" (*postcondition*).

Pak zápis **wp(S,R)** bude označovat nejslabší počáteční podmínuku (*weakest precondition*), která zaručuje, že mechanismus se dostane v konečné době do stavu splňujícího konečnou podmínuku **R**. Není-li nejslabší počáteční podmínka splněna, nelze zaručit, že se mechanismus **S** dostane do stavu splňujícího **R**, i když to nesplnění podmínky nevylučuje. Při nesplnění podmínky **(S,R)** se může mechanismus dostat do stavu nesplňujícího **R**

**DEF**

**DEF**

**DEF**

nebo do stavu nekonečné aktivity.

Množina všech možných konečných podmínek pro daný mechanismus je tak rozsáhlá, že její znalost, např. v tabelární podobě, která by určila rychlé určení  $(S, R)$  je prakticky nezvládnutelná. Proto je definice sémantiky mechanismu daná ve formě pravidel, popisujících, jak odvodíme k dané konečné podmínce  $R$  odpovídající nejslabší počáteční podmínce  $wp(S, R)$ . Pro daný mechanismus  $S$  a daný predikát  $R$  je takové pravidlo, jež dá za výsledek  $(S, R)$  označováno jako "transformace predikátu" a definuje se jí sémantika mechanismu  $S$ .

Nejčastěji nás však nezajímá úplná sémantika mechanismu. Mechanismu  $S$  používáme pouze pro zvláštní účel - pro ustavení pravdivosti určité konečné podmínky  $R$ , pro niž byl mechanismus navržen. Ani pro tuto určitou konečnou podmínu  $R$  nás nezajímá přesná a úplná forma  $wp(S, R)$ , ale obvykle o něco silnější "postačující" podmínka  $P$ , pro niž platí:

$$P \rightarrow wp(S, R) \text{ pro všechny stavy}$$

Pak  $P$  je postačující počáteční podmínka. V terminologii množin to znamená, že množina stavů označená symbolem  $P$  je podmnožinou stavů, kterou označuje zápis  $wp(S, R)$ .

Chápeme-li transformaci predikátu  $wp(S, R)$  jako funkci konečné podmínky  $R$ , pak má tato funkce několik základních vlastností:

1) Pro každý mechanismus  $S$  platí  $wp(S, F) = F$  (1.1)  
Této vlastnosti se říká "zákon vyloučeného zázraku".

2) Pro každý mechanismus  $S$  a konečné podmínky  $Q$  a  $R$  takové,  
že platí  $Q \rightarrow R$  pro všechny stavy, platí také

$wp(S, Q) \rightarrow wp(S, R)$  (1.2)  
pro všechny stavy. Této vlastnosti se říká "zákon monotónnosti".

3) Pro každý mechanismus  $S$  a konečné podmínky  $Q$  a  $R$  platí:

$wp(S, Q) \text{ and } wp(S, R) = wp(S, Q \text{ and } R)$  pro všechny stavy (1.3)  
a také

$wp(S, Q) \text{ or } wp(S, R) = wp(S, Q \text{ or } R)$  (1.4)

### 8.2.2. Definice základních mechanizmů

Definujme pro tvorbu základních mechanizmů tyto elementární mechanizmy:

1) Prázdný příkaz "skip", jehož sémantika je dána transformací:

$$wp("skip", R) = R \quad (2.1)$$

2) Příkaz zastavení v důsledku chybového stavu "abort", se sémantikou

$$\text{wp}("abort", \mathbf{R}) = \mathbf{F} \quad (2.2)$$

3) Přiřazovací příkaz  $\text{wp}(x:=E, \mathbf{R}) = R_x^E$  (2.3)

kde zápisem  $R_x^E$  se rozumí textová kopie  $\mathbf{R}$ , v níž je každý výskyt proměnné  $x$  nahrazen výrazem  $E$

např. :  $\text{wp}(x:=7, x=7) = (7=7) = \mathbf{T}$   
 nebo  $\text{wp}(x:=7, x=6) = (7=6) = \mathbf{F}$   
 nebo  $\text{wp}(x:=x-1, x^2>1) = ((x-1)^2>1) = (x>2 \text{ or } x<0) = (x<>1)$   
 (\* pro celá čísla \*)

Pomocí BNF lze příkaz definovat zatím takto:

```
<příkaz> ::= "skip" | "abort" | <přiřazovací příkaz>
<přiřazovací příkaz> ::= <proměnná> := <výraz>
```

Pro některé účely rozšíříme přiřazovací příkaz o možnost paralelního přiřazení takto:

```
<přiřazovací příkaz> ::= <proměnná> := <výraz> |
 <proměnná>, <přiřazovací příkaz>, <výraz>
```

Tento příkaz umožní např. zápisem  $x_1, x_2 := E_1, E_2$  přiřadit dvěma proměnným současně hodnoty dvou výrazů nebo zápisem  $X, Y := Y, X$  provést vzájemnou výměnu hodnot dvou proměnných.

Pozn. Podobné příkazy nejsou ve standardním Pascalu povoleny!

### 8.2.3 Definice složených příkazů

Nejjednodušším způsobem, jak odvodit ze dvou daných funkcí jednu funkci novou je způsob, v němž hodnota první funkce slouží jako argument druhé. Již tradičně má notace takového složení tvar " $S_1; S_2$ " a jeho sémantika je dána vztahem:

$$\text{wp}(S_1; S_2, \mathbf{R}) = \text{wp}(S_1, \text{wp}(S_2, \mathbf{R})) \quad (3.1)$$

Tato definice se často označuje jako "sémantická definice středníku". Jinými slovy říká: jestliže v posloupnosti " $S_1, S_2$ " má mechanismus  $S_2$  dosáhnou určitého konečného stavu splňujícího podmínu  $R$ , pak jeho nejslabší počáteční podmínu musí zaručit konečný stav mechanismu  $S_1$ . Nejslabší počáteční podmínka mechanismu " $S_1, S_2$ " k dosažení konečného stavu  $\mathbf{R}$  je tedy dána nejslabší počáteční podmínkou mechanismu  $S_1$ .

Příklad: Sekvence příkazů: " $x := x + y$ ;  $y := x - y$ ;  $x := x - y$ " realizují vzájemnou

výměnu hodnot x a y bez další pomocné proměnné, tedy mechanismus, který v Dijsktrrově jazyku může být popsán příkazem „x,y:=y,x“.

Důkaz: Dosadíme do vztahu 3.1 a s pomocí 2.3 dostaneme:

```

wp ("x:=x+y; y:=x-y; x:=x-y", (x=X) and (y=Y)) =
= wp ("x:=x+y; y:=x-y", ("x:=x-y", (x=X) and (y=Y))) =
= wp ("x:=x+y; y:=x-y", (x-y=X) and (y=Y)) =
= wp ("x:=x+y", ("y:=x-y", ((x-y)=X) and (y=Y))) =
= wp ("x:=x+y", ((x-(x-y))=X) and (((x-y)=Y)) =
= wp ("x:=x+y", (y=X) and (x-y)=Y) =
= (y=X) and ((x+y)-y)=Y =
= (y=X) and (x=Y) Q.E.D. (Lat. "Quod Erat Demonstrandum" =
"Což bylo dokázati")

```

Složitější kompozicí jednoduchých příkazů jsou řízené příkazy. Umožňují tvorbu alternativních a repetičních řídicích struktur. Definici příkazu pak můžeme pomocí BNF rozšířit o alternativní příkaz "**IF**" a repetiční příkaz "**DO**" takto:

```

<příkaz> ::= ... if <soubor řízených příkazů> fi |
do <soubor řízených příkazů> od
<soubor řízených příkazů> ::= <řízený příkaz> {!<řízený příkaz>}
<řízený příkaz> ::= <řídicí hlavička příkazu> {;<příkaz>}
<řídicí hlavička příkazu> ::= <Booleovský výraz> →→ <příkaz>

```

kde symbol "!" má funkci oddělovače jednotlivých alternativ, jejichž pořadí v souboru řízených příkazů nemá žádný význam.

### 8.2.3.1 Popis příkazu "**IF**".

Alternativní příkaz "**IF**" má několik důležitých vlastností:

- Předpokládá se, že všechny řídicí Booleovské výrazy jsou definované. V jiném případě může vyhodnocení nedefinovaného výrazu vést k nesprávně provedené aktivitě a tedy i celý příkaz "**IF**" nemusí pracovat správně.
- Obecně vede řídicí struktura "**IF**" k nedeterminovanosti, protože pro každý počáteční stav, který způsobí, že více než jeden Booleovský výraz je pravdivý, může být pro určení aktivity vybrán kterýkoli z nich.
- Jestliže je počáteční stav takový, že žádný z Booleovských řídicích výrazů není pravdivý, pak aktivace takového počátečního stavu povede k zastavení s chybou a v tom případě je řídicí struktura "**IF**" ekvivalentní příkazu "abort". K témuž vede i příkaz "**IF**" s prázdným souborem řízených příkazů, tedy konstrukce "**if fi**".

Nejslabší počáteční podmínka příkazu "**IF**" je stanovena takto: Nechť "**IF**" je označení příkazu, jehož tvar je:

```

if
B1 →→ S1

```

$\neg B_2 \rightarrow \neg S_2$

$\neg \dots$

$\neg B_n \rightarrow \neg S_n$

**fi**

kde  $S_i$  je seznam příkazů řízených Booleovským výrazem  $B_i$ , pak pro libovolnou konečnou podmínu  $\mathbf{R}$  platí:

$$\text{wp} ("IF", \mathbf{R}) = ((\text{Exist } j: 1 \leq j \leq n) [B_j]) \text{ and } ((\text{ForAll } j: 1 \leq j \leq n) [B_j] \rightarrow \text{wp} (S_j, \mathbf{R})) \quad (3.1.1)$$

#### 8.2.3.2 Popis příkazu "DO"

Formální definice nejslabší počáteční podmínky pro repetiční příkaz "DO" je poněkud složitější, než pro příkaz "IF".

Nechť "DO" je označení alternativního příkazu se stejným souborem řízených příkazů. Nechť podmínky  $H_k(\mathbf{R})$  jsou definovány takto:

$$H_0(\mathbf{R}) = \mathbf{R} \text{ and not}(\text{Exist } j: 1 \leq j \leq n) [B_j] \quad (3.2.1)$$

$$\text{a pro } k > 0 \quad H_k(\mathbf{R}) = \text{wp} ("IF", H_{k-1}(\mathbf{R})) \text{ or } H_0(\mathbf{R}) \quad (3.2.2)$$

$$\text{pak} \quad \text{wp} ("DO", \mathbf{R}) = (\text{Exist } k: k \geq 0) [H_k(\mathbf{R})] \quad (3.2.3)$$

Intuitivně chápeme  $H_k(\mathbf{R})$  jako nejslabší počáteční podmínu zaručující ukončení aktivity příkazu "DO" po maximálně  $k$  průchodech cyklem. Každý průchod je určen výběrem některého z řídicích výrazů a aktivuje odpovídající řízené příkazy.  $H_k(\mathbf{R})$  současně zabezpečuje, že po ukončení příkazu "DO" bude systém ve stavu splňujícím konečnou podmínu  $\mathbf{R}$ .

Pro  $k=0$  ukončí příkaz "DO" svou aktivitu, aniž provede výběr některého řídicího výrazu, protože žádný z nich, jak plyne z 3.2.1 není pravdivý. Pak počáteční pravdivost podmínky  $\mathbf{R}$  je nutnou a postačující podmínkou pro splnění konečné podmínky  $\mathbf{R}$ .

Pro  $k > 0$  rozlišíme dva případy:

a) žádný z Booleovských výrazů řídicích výrazů není pravdivý a v tom případě z 3.2.1 a z 3.2.2 plyne pravdivost  $\mathbf{R}$ .

b) Alespoň jeden řídicí výraz je pravdivý. Pak se provede jeden průchod řízeným příkazem, který je ekvivalentní příkazu "IF" se stejnou strukturou souboru řízených příkazů. (Protože druhý člen pravé strany vztahu 3.2.1 je nepravdivý a jeho negace je pravdivá, nemůže dojít k "abortu"). Po skončení aktivity příkazu tohoto průchodu musí systém přejít do stavu, který zabezpečuje, že po maximálně  $k-1$  dalších průchodech bude ustaven stav splňující podmínu  $H_0(\mathbf{R})$ . Je to

zabezpečeno tím, že konečným stavem průchodu (resp. ekvivalentního příkazu "**IF**") je podle 3.2.1 podmínka  $H_{k-1}(\mathbf{R})$ .

Repetiční příkaz "**DO**", jehož počáteční stav splňuje podmínu 3.2.1 je ekvivalentní prázdnému příkazu "skip". K témuž vede i prázdný soubor řízených příkazů, tedy konstrukce **do od**.

#### 8.2.4. Teorém alternativního příkazu "**IF**"

Nechť je dán příkaz "**IF**" a predikát BB pro něž platí:

$$BB = (\text{Exist } j : 1 \leq j \leq n)[B_j].$$

S použitím uvedených konvencí lze teorém příkazu "**IF**" vyjádřit takto:

nechť **P** a **Q** jsou predikáty pro něž platí:

$$(\text{ForAll } j : 1 \leq j \leq n)[(P \text{ and } B_j) \rightarrow \mathbf{wp}(S_j, Q)] \quad (4.1)$$

$$\text{a také } P \rightarrow BB \quad (4.2)$$

$$\text{pak tedy platí } P \rightarrow \mathbf{wp}("IF", Q) \quad (4.3)$$

Důkaz: Podle definice příkazu "**IF**" (3.1.1) platí:

$$\mathbf{wp}("IF", Q) =$$

$$((\text{Exist } j : 1 \leq j \leq n)[B_j]) \text{ and } ((\text{ForAll } j : 1 \leq j \leq n)[B_j] \rightarrow \mathbf{wp}(S_j, Q))$$

Musíme tedy dokázat:

$$P \rightarrow (\text{Exist } j : 1 \leq j \leq n)[B_j] \text{ and } (\text{ForAll } j : 1 \leq j \leq n)[B_j] \rightarrow \mathbf{wp}(S_j, Q)$$

Pravdivost implikace prvního členu vyplývá z (4.2) a stačí tedy dokázat, že pro všechny stavy platí:

$$P \rightarrow (\text{ForAll } j : 1 \leq j \leq n)[B_j] \rightarrow \mathbf{wp}(S_j, Q) \quad (4.4)$$

Pro všechny stavy, pro něž je **P** nepravdivé, je vztah (4.4) pravdivý z definice implikace. Všechny stavy, pro něž je **P** pravdivé a pro všechna j rozlišíme dva případy:

a) Bud' je **B<sub>j</sub>** nepravdivé, pak je  $B_j \rightarrow \mathbf{wp}(S_j, Q)$  pravdivé z definice implikace a pak je pravdivý i vztah (4.4)

b) Je-li naopak **B<sub>j</sub>** pravdivé, pak je na základě (4.1) pravdivé i **wp(S<sub>j</sub>, Q)**. Pravdivé je tedy i  $B_j \rightarrow \mathbf{wp}(S_j, Q)$  a tím i (4.4). Tím je dokázána platnost vztahu (4.4) a tím i (4.3), Q.E.D.

Závěr teorému říká, že **P**, které splňuje podmínky (4.1) a (4.2), implikuje nejslabší počáteční podmínu zabezpečující počáteční stav, který se mechanismem "**IF**" změní do konečného stavu, splňujícího podmínu **Q**. Odvození vyhovujícího **P** je tedy důkazem správnosti alternativní konstrukce "**IF**". Skutečnost, že premisy mají stejný tvar jako závěr je zárukou, že důkaz konstrukce bude mít (vzhledem ke konstrukci samotné) lineární charakter své délky.

### 8.2.5 Teorém invariance pro repetiční příkaz "DO"

Tento teorém, který odvodil C.A.R.Hoare, se považuje za jeden z nejzávažnějších teorémů programování.

Zavedeme pomocné formální prostředky:

Zápis  $wdec(S,t)$  (weakest decrement) nechť se interpretuje jako nejslabší počáteční podmínka pro takový počáteční stav, který zaručuje, že mechanismus  $S$  v konečné době sníží hodnotu  $t$ , kde  $t$  je celočíselná funkce proměnných programu. pak lze psát:

$$wdec(S,t) = \mathbf{wp}('tau:=t; S', t < tau) \quad (5.1)$$

Tento vztah lze podle (3.1) rozvést na:

$$\begin{aligned} wdec(S,t) &= \mathbf{wp}('tau:=t', \mathbf{wp}(S, t < tau)) \text{ a konečně na} \\ wdec(S,t) &= [\mathbf{wp}(S, t < tau)]^{tau} \end{aligned} \quad (5.2)$$

kde pravá strana vztahu (5.2) se interpretuje tak, že ve výrazu konečné podmínky bude každé  $tau$  nahrazeno hodnotou  $t$ . Použití  $wdec(S,t)$  ilustruje následující příklad:

Příklad:  $wdec("x:=x-y, x+y")$  je nejslabší podmínka, za níž příkaz " $x:=x-y$ " sníží hodnotu funkce  $x+y$ .

Podle (5.2) lze psát:

$$\begin{aligned} wdec("x:=x-y", x+y) &= [("x:=x-y", x+y < tau)]^{tau}_{x+y} = \\ &[x-y + y < tau]^{tau}_{x+y} = x < x+y = y > 0 \end{aligned}$$

Je-li  $y > 0$ , pak příkaz  $x := x - y$  sníží hodnotu funkce  $x+y$ .

Nechť je repetiční příkaz "DO" definován zápisem

```
do
 B1 --> S1
 ! B2 --> S2
 !
 ! ...
 ! Bn --> Sn
od
```

a predikát  $\mathbf{BB}$  je definován:  $\mathbf{BB} = (\mathbf{Exist} j: 1 \leq j \leq n)[B_j]$

Pak lze teorém invariance repetičního příkazu formulovat takto:

nechť  $\mathbf{P}$  je predikát takový, že platí

$$(\mathbf{ForAll} j: 1 \leq j \leq n)[(\mathbf{P} \text{ and } B_j) \rightarrow (\mathbf{wp}(S_j, \mathbf{P}) \text{ and } wdec(S_j, t))] \quad (5.3)$$

$$\text{a současně } \mathbf{P} \rightarrow t > 0 \quad (5.4)$$

$$\text{pak } \mathbf{P} \rightarrow \mathbf{wp}("DO", \mathbf{P} \text{ and } (\text{not } \mathbf{BB})) \quad (5.5)$$

Tvrzení (5.5) říká, že jestliže počáteční stav zaručuje pravdivost predikátu **P** a jestliže kterýkoliv z řídicích výrazů je vybrán a jeho řízené příkazy provedeny, pak po skončení aktivity zůstává predikát **P** pravdivý. Predikát **P** tedy zůstává pravdivý (invariantní, neměnný) bez ohledu na počet, kolikrát bude ten či onen řídicí výraz vybrán a jeho řízené příkazy provedeny. Po skončení aktivity celé repetiční konstrukce "**DO**", kdy už žádný z řídicích výrazů není pravdivý, zaručuje konečný stav pravdivost tvrzení:

**P and (not BB)**

Vztah (5.3) zaručuje neustálé snižování funkce **t** a (5.4) zaručuje, že její hodnota je kladná. Funkce **t** je celočíselná funkce a je tedy spolehlivým prostředkem zakončení aktivity repetiční konstrukce.

Závažnost teorému invariance pro cyklus spočívá v tom, že pravdivost jeho výroků nezávisí na počtu průchodů. Z toho vyplývá, že lze postavit o cyklu tvrzení i v tom případě, kdy počáteční stav neurčuje tento počet průchodů. Umožňuje to provést důkaz správnosti repetiční konstrukce, jehož délka není úměrná počtu průchodů cyklu.

### 8.2.6. Příklady

#### 8.2.6.1 Největší společný dělitel dvou celých čísel

Nechť  $X, Y$  jsou celá čísla větší než 0. Hledáme největší společný dělitel (dále jen NSD) těchto čísel. Řešení má formálně tvar:

$$\mathbf{R} : Z = \text{NSD}(X, Y) \text{ and } (X > 0) \text{ and } (Y > 0) \quad (6.1.1)$$

Z definice NSD dvou celých čísel vyplývá:

$$\text{NSD}(X, Y) = \text{NSD}(Y, X) \quad (6.1.2)$$

$$\text{a } \text{NSD}(X, X) = X \quad (6.1.3)$$

Lze dokázat, že platí také

$$\text{NSD}(X, Y) = (\text{NSD}(X, Y - X) \text{ and } (Y > X)) \quad (6.1.4)$$

a z toho lze dále odvodit, že platí

$$\text{NSD}(X, Y) = \text{NSD}(X - Y, Y) \text{ and } (X > Y) \quad (6.1.5)$$

$$\text{NSD}(X, Y) = \text{NSD}(X + Y, Y) = \text{NSD}(X, Y + X) \quad (6.1.6)$$

Pro získání řešení je důležitý vztah (6.1.3) a jestliže konečný stav mechanismu zajistí relaci  $x=y$ , pak tento stav zajišťuje také řešení  $\text{NSD}(x,y)=x$ . Mechanismus však musí také zajistit neměnnost (invarianci) relace:

$$\mathbf{P} : \text{NSD}(X, Y) = \text{NSD}(x, y) \text{ and } (0 < x \leq X) \text{ and } (0 < y \leq Y) \quad (6.1.7)$$

Z počátečního stavu  $x=X$  a  $y=Y$  bude mechanismus zpracovávat  $x$  a  $y$  tak, aby se zachovala invariance  $\mathbf{P}$  a aby se dosáhlo relace  $x=y$ . Vztah (6.1.4) resp (6.1.5) nabízí změnu  $x$  a  $y$  jejich rozdílem. Prozkoumejme podmínu, za níž příkaz " $x:=x-y$ " dosáhne žádoucí konečné podmínky  $\mathbf{P}$ .

$$\mathbf{wp} ("x:=x-y", \mathbf{P}) = (\text{NSD}(x-y, y) = \text{NSD}(X, Y)) \text{ and } (0 < (x-y) \leq X) \text{ and } (0 < y \leq Y) \quad (6.1.8)$$

Jinými slovy, pro mechanismus " $x:=x-y$ " se hledá podmínka  $\mathbf{P}$  taková, aby platila pravá strana rovnice, která říká, že pro každou úpravu argumentů operace NSD musí být zajištěna shodnost výsledků (na základě platnosti (6.1.2 - 7)).

Z teorému invariance pro cyklus vyplývá, že najdeme-li  $\mathbf{P}$  a  $\mathbf{B}_i$  takové, že platí

$$(\mathbf{P} \text{ and } \mathbf{B}_i) \rightarrow \mathbf{wp} (\mathbf{S}_i, \mathbf{P}) \text{ and } \mathbf{wdec}(\mathbf{S}_i, t)$$

a  $\mathbf{P} \rightarrow t > 0$   
pak  $\mathbf{P} \rightarrow \mathbf{wp} ("DO", \mathbf{P} \text{ and } (\text{not } \mathbf{BB}))$ .

Ze vztahu (6.1.7) je vidět, že  $\mathbf{P}$  implikuje všechny členy pravé strany vztahu (6.1.8) s výjimkou  $0 < (x-y)$ . Z toho vyplývá, že:

$$(\mathbf{P} \text{ and } (x > y)) \rightarrow \mathbf{wp} ("x:=x-y", \mathbf{P}) \quad (6.1.9)$$

A v důsledku symetrie tedy také:

$$(\mathbf{P} \text{ and } (y > x)) \rightarrow \mathbf{wp} ("y:=y-x", \mathbf{P}) \quad (6.1.10)$$

Zbývá najít funkci  $t$ , která vyhovuje podmínkám teorému invariance.  
Nechť  $t=x+y$ . Ze vztahu (6.1.7) platí, že  $\mathbf{P} \rightarrow t > 0$ , a pak lze tedy odvodit:

$$\begin{aligned} \mathbf{wdec}("x:=x-y", x+y) &= [\mathbf{wp} (x:=x-y, \mathbf{tau} > (x+y))]^{\mathbf{tau} t} = [\mathbf{tau} > (x-y)]^{\mathbf{tau} x+y} = \\ &= (x+y) > x = y > 0 \end{aligned}$$

A pak tedy ze vztahu (6.1.7) platí, že  $\mathbf{P} \rightarrow \mathbf{wdec}("x:=x-y", x+y)$ .

Výsledný program má tuto strukturu:

"Ustav počáteční podmínu  $\mathbf{P}$ ";  
**do** "snižuje hodnotu funkce  $t$  při invarianci podmíny  $\mathbf{P}$ "  
**od** (\*  $\mathbf{P}$  and (not  $\mathbf{B}$ ) →  $\mathbf{R}$ ,  
 jinými slovy pravdivost podmínky  $\mathbf{P}$  a nepravdivost  
 podmínky  $\mathbf{B}$  implikuje řešení  $\mathbf{R}$  \*)

Konečný tvar odvozeného programu s dokázánou správností je:

$x:=X; y:=Y; (* \mathbf{P} *)$   
**do**

```

x>y →→ x:=x-y (* P and B1 *)
! y>x →→ y:=y-x (* P and B2 *)
od;
z:=x; (* splnění P and (not B) zaručuje dosažení výsledku *)

```

#### 8.2.6.2 Součin dvou celých kladných čísel

Nechť X,Y jsou celá kladná čísla (X>0) and (Y>0).

Řešení součinu těchto dvou čísel má tvar:

$$\mathbf{R}: Z = X * Y \quad (6.2.1)$$

Invariantní relace cyklu se často tvoří využitím pomocných proměnných, které na počátku nabývají počátečních hodnot vstupujících argumentů. Jejich změnou v průběhu cyklu, při zachování invariance relace dosáhneme řešení. Relací vhodnou pro daný příklad je:

$$\mathbf{P} : (z+x*y = X*Y) \text{ and } (0 <= x <= X) \text{ and } (0 <= y <= Y) \quad (6.2.2)$$

Ze vztahů (6.2.1) a (6.2.2) vyplývá, že

$$(P \text{ and } (y=0)) \rightarrow R \quad (6.2.3)$$

Program pak bude mít formálně tento tvar:

```

"Ustavení počátečního stavu splňujícího P";
do
 x<>0 →→ "snižuj hodnotu y při zachování platnosti
 invariance P"
 od (* P and (y=0) implikuje R *)

```

nejjednodušším mechanismem snižujícím hodnotu y a současně zabezpečujícím konečnost cyklu je příkaz "y:=y-1".

$$\begin{aligned} \text{Pak } ("y:=y-1, P) = & ((z+x*(y-1)=X*Y) \text{ and } (0 <= x <= X) \text{ and } (0 <= (y-1) \\ & <= Y)) = ((z-x+x*y = X*Y) \text{ and } (0 <= x <= X) \text{ and } (0 <= (y-1) <= Y)) \end{aligned} \quad (6.2.4)$$

Aby se zachovala pravdivost invariance P, je třeba současně s přiřazením y:=y-1 provést přiřazení z:=z+x, které kompenzuje ve vztahu (6.2.4) úbytek vzniklý snížením hodnoty y. Program pak bude mít tvar:

```

x, y, z := X, Y, 0; (* ustavení P *)
do
 y<>0 →→ z, y := z+x, y-1 (* P and B1 *)
 od (* P and (not BB) implikuje řešení R, které ustavuje hodnotu z na
 výsledek součinu *)

```

Snižování y lze provést rychleji, připustíme-li některé další operace. Nechť existuje booleovský predikát pro lichost čísla "odd(x)" (např. odd(9) má

hodnotu true, odd(8) má hodnotu false) a nechť je k dispozici operátor pro celočíselné dělení dvěma "half(x)" (např. half(5)má hodnotu 2, half(6) má hodnotu 3) a pro celočíselné zdvojení hodnoty "double(x)" (např double(3) má hodnotu 6, double(4) má hodnotu 8).

Pak za předpokladu, že y je sudé, tzn za předpokladu platnosti podmínky  
not odd(y) (6.2.5)

lze snižování provést přiřazovacím příkazem "y:=half(y)".

Aby se zachovala platnost invariance **P**, je třeba současně kompenzovat tento úbytek hodnoty proměnné y přírůstkem hodnoty proměnné x pomocí příkazu "x:=double(x)".

Cyklus

**do** not odd(y)  $\rightarrow$  x,y:=double(x), half(y) **od**

zachová platnost invariance **P** a současně končí neplatností podmínky (6.2.5), což znamená, že y je nyní liché. Takový cyklus se bude s výhodou doplňovat s příkazem původního programu "y,z:=y-1,z+x", který při zachování platnosti podmínky **P** znova ustanoví sudost proměnné y a tím podmínu pro jeho rychlejší snižování půlením. Program pak má tento konečný tvar:

```
x, y, z := X, Y, 0; (* ustavení P *)
do
 Y <> 0 \rightarrow do
 not odd(y) \rightarrow x, y, :=double(x), half(y)
 od; (* platí odd(y) *)
 y, z := y-1, z+x (* platí not odd(y) *)
 od (* splnění P and (not BB) implikuje řešení R: z=X*Y *)
```

### 8.2.6.3 Binární vyhledávání

Nechť je dáno pole celých čísel, pro které platí:

$A[0] \leq A[1] \leq \dots \leq A[N-1] < A[N]$

a nechť je dána hodnota klíče x, pro který platí  $A[0] \leq x \leq A[N]$ .

Nalezněme algoritmus, který ustanoví pravdivost Booleovské proměnné SEARCH v případě, že x se rovná hodnotě některého prvku zadaného pole, tedy řešení ve tvaru:

**R** : SEARCH=(Exist i:(0<=i<N))[x=A[i]] (6.3.1)

Vzhledem k seřazenosti pole skončí repetiční proces dosažením podmínky

**R'** :  $A[i] \leq x < A[i+1]$  (6.3.2)

Pak tedy platí (**R'** and SEARCH=(x=A[i]))  $\rightarrow$  **R** (6.3.3)

Invariantní relaci zavedeme pomocí proměnné  $j$  a vztahu (6.3.2) s cílem, aby  $\mathbf{P}$  and  $(j=i+1) \rightarrow \mathbf{R}$ . Pak tedy bude o invariantní relaci  $\mathbf{P}$  platit:

$$\mathbf{P} : (A[i] \leq x < A[j]) \text{ and } (0 \leq i < j \leq N) \quad (6.3.4)$$

Cílem cyklu je zpracovat hodnoty  $i$  a  $j$  při platnosti invariance  $\mathbf{P}$  tak, aby se dosáhlo platnosti relace  $j:=i+1$ . Program bude mít tedy následující strukturu:

```
i, j := 0, N; (* Ustavení P *)
do
 j <> (i+1) --> "úprava i a j při zachování platnosti invariance P"
od; (* P and (j=(i+1)) *)
SEARCH := (x=A[i]); (* Ustavení řešení R *)
```

Necht' v důsledku úprav uvnitř cyklu nabude proměnná  $i$  nebo proměnná  $j$  nové hodnoty  $m$ . Nalezněme nejslabší počáteční podmínu pro mechanismus " $i:=m$ " resp. " $j:=m$ ".

$$\mathbf{wp} ("i=:m", \mathbf{P}) = (A[m] \leq x < A[j]) \text{ and } (0 \leq m < j \leq N) = \mathbf{P} \text{ and } (A[m] \leq x) \text{ and } (m < j) \quad (6.3.5)$$

$$\mathbf{wp} ("j:=m", \mathbf{P}) = (A[i] \leq x < A[m]) \text{ and } (0 \leq i < m \leq N) = \mathbf{P} \text{ and } (x < A[m]) \text{ and } (i < m) \quad (6.3.6)$$

Necht' funkce  $t=j-i$ . Nalezněme podmínky, za nichž zvolené mechanismu zaručí konečnost aktivity cyklu.

$$wdec("i=:m", tau > (j-i)) = [tau > (j-m)]^{tau}_{j-i} = (j-i) > (j-m) = m > i \quad (6.3.7)$$

$$wdec("j:=m", tau > (j-i)) = [tau > (m-i)]^{tau}_{j-i} = (j-i) > (m-i) = j > m \quad (6.3.8)$$

jak vyplývá ze vztahů (6.3.5-8), musí nová hodnota  $m$  splňovat podmínu

$$i < m < j \quad (6.3.9)$$

Z hlediska symetrie je pro  $m$  vhodnou hodnotu, která půlí interval  $(i,j)$ , tedy

$$m := half(i+j)$$

Vztah  $\mathbf{P}$  and  $(j < (i+1))$ , který platí po celou dobu cyklu, zajišťuje pro takové  $m$  platnost vztahu (6.3.9). Protože maximálně smí nabýt i hodnoty  $j-2$ , pak

$$i_{max} = half(i+j) = half(2*j-2) = j-1$$

a také

$$j_{min} = half(i+j) = half(2*i+2) = i+1$$

Celý program pak bude mít konečný tvar:

$$i, j := 0, N; (* ustavení podmínky P *)$$

```

do
 j<>(i+1) → m:=half(i+j); (* platí i<m<j *)
 if
 A[m]<x → i:=m (* P and (A[m]<=x) and (m<j) *)
 ! x<A[m] → j:=m (* P and (x<A[m]) and (i<m) *)
 fi
od; (* P and j=(i+1) ustavuje R'*)
SEARCH:=(x=A[i]) (* ustavuje R *)

```

#### 8.2.6.4 Teorém pro lineární vyhledávání

Nechť **B** je booleovská funkce celočíselného argumentu. Mějme program tvaru:

```

i:=0;
do
 B(i) → i:=i+1
od

```

Pro tento program, jehož základem je cyklus, lze napsat invariantní relaci **P** ve tvaru:

$$\begin{aligned} P(i) &= (\text{ForAll } j: (0 \leq j < i) [\text{not } B(j)] \\ (6.4.1) \end{aligned}$$

Důkaz: **wp**("i:=i+1", **P**(i)) = **P**(i+1) = (**ForAll** j: (0 <= j < i+1) [**not** **B**(j)]) =  
= **ForAll** j: (0 <= j < i) [**not** **B**(j)] and **not** **B**(i) =  
= **P**(i) and **not** **B**(i) Q.E.D.

uvedený cyklus se ukončí tehdy a jen tehdy, platí-li výrok

$$\text{Exist } j: (0 \leq j < i) [B(j)] \quad (6.4.2)$$

Pak bude platit

$$P(i) \text{ and } B(i) = \text{ForAll } j: (0 \leq j < i) [\text{not } B(j) \text{ and } B(i)] \quad (6.4.3)$$

což jinými slovy znamená, že i je nejmenší hodnota, pro niž je hodnota funkce **B** pravdivá. Mechanismu, který vyhledává v zadáné posloupnosti prvek s nejnižším pořadím, pro které platí zadána podmínka, se říká lineární vyhledávání. Teorém pro lineární vyhledávání se uplatní v řadě problémů, jejichž součástí je právě lineární vyhledávání. Teorém lze demonstrovat na jednoduchém příkladě:

Nechť je dáno pole celých čísel A[0], A[1], ..., A[N-1] a nechť je dán celočíselný klíč x. Chceme stanovit řešení:

$$R: SEARCH = (\text{Exist } i: (0 \leq i < N) [x = A[i]]) \quad (6.4.4)$$

Podle teorému lineárního vyhledávání bude aktivita cyklu konečná pouze při pravdivosti výroku **Exist** i: (0 <= i < N) [x = A[i]]. Tuto pravdivost můžeme zaručit rozšířením pole o prvek A[N]=x a zavedením pomocného řešení

$$\mathbf{R}':\text{SEARCH}=(\mathbf{Exist } i:(0 < i < N) [x = A[i]]) \quad (6.4.5)$$

$$\text{Pak } (\mathbf{R}' \text{ and } (i < N)) \rightarrow \mathbf{R} \quad (6.4.6)$$

Výsledný program má tvar:

```
i, A[N] := 0, x;
do
 A[i] <> x → i := i + 1
od;
SEARCH := i < N;
```

Tento algoritmus je známý pod názvem rychlé lineární vyhledávání, nebo lineární vyhledávání se zarážkou.

## Shrnutí



Ve skriptech Honzík a kol. :Vybrané kapitoly programovacích technik naleznete také odvozený algoritmus pro ekvivalence dvou kruhových seznamů implementovaný v poli a jeho implementaci. Algoritmus je zajímavý svou lineární časovou složitostí.

Uvedená statě tvorby dokázaných programů je ukázkou teoretického příspěvku k tvorbě správných programů. Smyslem tohoto odstavce je ukázat budoucím tvůrcům programů principy, kterých se může v náročnějších a na teoretickém základě vystavěných aplikacích.

## 9 Přílohy

### 9.1.

### 9. Přílohy

#### 9.1. Příklady pro seznamy

##### Příklady pro seznamy

Následující odstavec představuje programovou jednotku (modul) v jazyce Borland Pascal, která implementuje ukázkové operace nad jednosměrným seznamem.

##### 9.1.1. Jednosměrný seznam

```
unit Let_un;
(* Modul operací nad jednosměrnými seznamy *)
interface
(* >>>>>>> Pouzite typy <<<<<<< *)

type
 TUk=^TPrvek; (* Ukazatel na prvek jednosmer. seznamu *)
 TData=integer; (* Datova sl. prvku *)
 TDUk=^TDPrvek; (* Ukazatel na prvek dvojsmerneho seznamu *)
 TUkLL=^TListOfLists; (* Ukazatel na seznam seznamu *)
```

```

TCardinal=0..MaxInt;

TPrvek= record (* Typ prvku jednosmerneho seznamu *)
 Uk:TUK;
 Data:TData;
end; (* record *)

TDPrvrek=record (* Typ prvku dvousmerneho seznamu *)
 LUk,Puk:TDUK;
 Data:TData;
end; (* record *)

TList=record (* ATD jednosm. seznam *)
 Zac,Kon,Act,Mark:TUK;
 MarkUsable:Boolean;
 Len:TCardinal;
end; (* record *)

(* Do Mark se zaznamena stav aktivity operaci Mark.
Promenna MarkUsable je pro test pouzitelnosti operace ActSetMarked;
nastavi se na false, rysi-li se zaznamenany prvek.
Len je delka seznamu, nevyuzivana..*)

TDList=record (* ATD dvojsm. seznam *)
 Zac,Kon,Act,Mark:TDUK;
 MarkUsable:Boolean;
 Len:TCardinal;
end; (* record *)

TListOfLists=record
 Next:TUKLL;
 List:TList;
end;

(*=====
(* ===== Deklarace Operaci ===== *)
(* ===== Jednosmerny Seznam ===== *)

procedure InitList(var List:Tlist);
procedure InsertFirst(var List:TList; Elemt:TData);
procedure InsertLast(var List:TList; Elemt:TData);
procedure DeleteFirst(var L:TList);
procedure ActSucc(var List:TList);
function IsActive(List:TList):Boolean;
procedure First(var List:TList);
procedure CopyFirst(L:TList; var Elemt:TData);
procedure CopyLast(List:TList; var Elemt:TData);
procedure Copy(List:TList; var Elemt:TData);
procedure PostInsert(var List:TList; Elemt:TData);
procedure PostDelete(var List:TList);
procedure Actualize (List:TList; Elemt:TData);
procedure Clear(var List:TList);
function lengthList(List:TList):TCardinal;
function IsEmpty(List:Tlist):Boolean;
procedure MarkAct (var List:TList);
procedure SetActMarked (var List:TList);
function IsMarkUsable (List:TList):Boolean;

```

```

procedure FindAndRewrite(List:Tlist; Klic,Elem:TData);
procedure FindAndPostInsert(var List:Tlist; Klic,Elem:TData);
procedure FindAndDelete(var List:TList; Klic:TData);
procedure FindAndPostDelete(var List:TList; Klic:TData);
function EquList(L1,L2:TList) :Boolean;
function EquListRec(L1,L2:TList):Boolean;
function FirstListLess(L1,L2:TList):Boolean;
function FirstListLessRec(L1,L2:TList):Boolean;
procedure ListCopy(List1:TList; var List2:TList);
procedure ListCopyRec(L1:TList; var L2:TList);
procedure FindMaxSeq(L:TList; var Num,Len:TCardinal);
procedure Match(L,SubL:TList; var Pos:TCardinal);
procedure InsertSubList(var L:TList; SubList:TList; Num:TCardinal);
procedure DeleteSublist(var L:TList; Num,Len:TCardinal);
procedure CopySublist(L:TList; var SubL:TList; Num,Len:TCardinal);
procedure Concat(var L1,L2:TList);
procedure FindAndDecat(var L1,L2:TList; Klic:TData);
procedure MergeLists(var LSource1,LSource2,LDest:TList);
procedure DecatListToSeq(SourceList:Tlist; var DestList:TUKLL);
procedure InitListOfLists(LL:TUKLL);
procedure ClearListOfLists(var LL:TUKLL);
function LengthListOfLists(LL:TUKLL):TCardinal;
procedure CopyNthList(LL:TUKLL; Num:TCardinal; var L:TList);

implementation

(* ===== Deklarace Operaci ===== *)
(* ===== Jednosmerny Seznam ===== *)

(* >>>>>>> InitList <<<<<<<<< *)
(* Inicializace jednosmerneho seznamu. Nesmi by pouzita pro smazani
seznamu !! Viz procedure Clear *)
procedure InitList(var List:Tlist);
begin
 List.Zac:=nil;
 List.Act:=nil;
 List.Kon:=nil;
 List.Mark:=nil;
 List.MarkUsable:=false;
end; (* procedure InitList *)

(* >>>>>>> InsertFirst <<<<<<<< *)
(* Vloz prvni prvek *)

procedure InsertFirst(var List:TList; Elemt:TData);
var
 PomUk:TUK;
begin
 new(PomUk);
 PomUk^.Data:=Elemt;
 PomUk^.Uk:=List.Zac;
 List.Zac:=PomUk;
 If List.Kon=nil then List.Kon:=PomUk;
end;

(* >>>>>>> InsertLast <<<<<<<<< *)
(* Vloz posledni prvek *)

procedure InsertLast(var List:TList; Elemt:TData);

```

```

var
 PomUk:TUk;
begin
 with List do
 begin
 new(PomUk);
 PomUk^.Uk:=nil;
 PomUk^.Data:=Elem;
 if Kon<>nil
 then (* není prázdný *)
 Kon^.Uk:=PomUk
 else (* je prázdný *)
 Zac:=PomUk;
 (* if *)
 Kon:=PomUk
 end; (* with *)
end; (* procedure *)

(* >>>>>> DeleteFirst <<<<<<<< *)
(* Zruší první prvek *)

procedure DeleteFirst(var L:TList);
var
 PomUk:TUk;
begin
 if L.Zac<>nil
 then begin
 PomUk:=L.Zac;
 if L.Zac=L.Act
 then L.Act:=nil;
 L.Zac:=L.Zac^.Uk;
 if L.Zac=nil
 then L.Kon:=nil;
 if PomUk=L.Mark
 then begin
 L.MarkUsable:=false;
 L.Mark:=nil;
 end; (* if *)
 dispose(PomUk);
 end; (* if *)
end; (* procedure *)

(* >>>>>>> ActSucc <<<<<<<< *)
(* Posune aktivity na další prvek *)
procedure ActSucc(var List:TList);
begin
 with List do
 begin
 if Act<>nil
 then Act:=Act^.Uk
 end;
end;

(* >>>>>>> IsActive <<<<<<<< *)
(* Je seznam aktivní *)

function IsActive(List:TList):Boolean;
begin
 IsActive:=List.Act<>nil
end; (*function *)

```

```

(* >>>>>>> First <<<<<<<<< *)
(* Aktivizace prvního prvku *)

procedure First(var List:TList);
begin
 if List.Zac<>nil
 then List.Act:=List.Zac
 else List.Act:=nil;
end;

(* >>>>>>>CopyFirst>>>>>>>> *)
(* Obsah prvního prvku *)

procedure CopyFirst(L:TList; var Elemt:TData);
begin
 if L.Zac<>nil
 then Elemt:=L.Zac^.Data;
end; (* procedure *)

(* >>>>>>> CopyLast <<<<<<<< *)
(* Obsah posledního prvku *)
procedure CopyLast(List:TList; var Elemt:TData);
begin
 if List.Kon<>nil
 then Elemt:=List.Kon^.Data;
end; (* procedure *)

(* >>>>>>> Copy <<<<<<<< *)
(* Obsah aktívniho prvku *)
procedure Copy(List:TList; var Elemt:TData);
begin
 if List.Act<> nil
 then Elemt:=List.Act^.Data;
end; (* procedure *)

(* >>>>>>> PostInsert <<<<<<< *)
(* Vlož za aktívni prvek *)

procedure PostInsert(var List:TList; Elemt:TData);
var
 PomUk:TUk;
begin
 with List do
 begin
 begin
 if Act<>nil
 then begin
 new(PomUk);
 PomUk^.Data:=Elemt;
 PomUk^.Uk:=Act^.Uk;
 Act^.Uk:=PomUk;
 if Act=Kon (* nový prvek se stane poslední *)
 then Kon:=PomUk;
 end; (* if *)
 end (* with *)
 end; (* procedure *)

(* >>>>>>> PostDelete <<<<<<< *)
(* Zrus prvek za aktívni *)

```

```

procedure PostDelete(var List:TList);
var
 PomUk:TUk;
begin
 with List do
 begin
 if Act<>nil
 then begin
 PomUk:=Act^.Uk;
 if PomUk<>nil
 then begin
 Act^.Uk:=PomUk^.Uk;
 if Kon=PomUk
 then Kon:=Act; (* rusi se posledni prvek*)
 if PomUk=Mark
 then begin
 Mark:=nil; (* puvodni aktivita neobnovitelna *)
 MarkUsable:=false
 end;
 dispose(PomUk);
 end (* if PomUk *)
 end (* if Act *)
 end (* with *)
 end; (* procedure *)
(* >>>>>>> Actualize <<<<<<<< *)
(* Aktualizuj obsah aktivniho prvku *)

procedure Actualize (List:TList; Elemt:TData);
begin
 if List.Act<>nil
 then List.Act^.Data:=Elemt
end; (* procedure *)

(* >>>>>>>>> Clear <<<<<<<< *)
(* Zrus seznam *)

procedure Clear(var List:TList);
var
 PomUk,DelUk:TUk;
begin
 PomUk:=List.Zac;
 while PomUk<>nil
 do begin
 DelUk:=PomUk;
 PomUk:=PomUk^.Uk;
 dispose(DelUk);
 end;
 with List
 do begin
 Zac:=nil;
 Kon:=nil;
 Act:=nil;
 Mark:=nil;
 MarkUsable:=false;
 end; (* with *)
end;

(* >>>>>>>> LengthList <<<<<<< *)
(* Zjisteni delky seznamu *)

```

```

function lengthList(List:Tlist):TCardinal;
var
 n:TCardinal;
 PomUk:TUk;
begin
 n:=0;
 PomUk:=List.Zac;
 while PomUk<>nil
 do begin
 PomUk:=PomUk^.Uk;
 n:=n+1;
 end; (* while *)
 LengthList:=n;
end; (* function *)

(* >>>>>>>> IsEmpty <<<<<<<< *)
(* Test praznosti seznamu *)

function IsEmpty(List:Tlist):Boolean;
begin
 IsEmpty:=List.Zac=nil
end; (* function *)

(* ===== Operace zaznamenavajici ===== a obnovujici aktivitu ===== *)
(* >>>>>>>> MarkAct <<<<<<< *)
(* Procedura zaznamena stav aktivity *)

procedure MarkAct (var List:TList);
begin
 with List
 do begin
 Mark:=Act;
 MarkUsable:=Act<>nil;
 end; (* with *)
end;

(* >>>>>> SetActMarked <<<<<< *)
(* Procedura obnovi aktivitu (pokud to je korektni) *)

procedure SetActMarked (var List:TList);
begin
 with List
 do begin
 if MarkUsable
 then begin
 Act:=Mark;
 end else begin
 Act:=nil;
 end; (* if *)
 end; (* with *)
end;

(* >>>>>>> IsMarkUsable <<<<<<< *)
(* Funkce vraci true, je-li korektni moznost obnovit aktivitu. *)

function IsMarkUsable (List:TList):Boolean;
begin
 IsMarkUsable:=List.MarkUsable;

```

```

end;

(* === Operace typu "najdi prvek obsahujici zadany klic" a === *)
(* ===== zapis nova data (aktualizuj) ===== *)

(* >>>>>> FindAndRewrite <<<<<< *)

procedure FindAndRewrite(List:Tlist; Klic,Elem:TData);
var
 Found:Boolean;
 PomEl:TData;
begin
 First(List);
 Found:=false;
 while IsActive(List) and not Found do begin
 Copy(List,PomEl);
 if PomEl=Klic (* nasel ? *)
 then begin
 Found:=true;
 Actualize(List,Elem)
 end else begin
 ActSucc(List);
 end; (* if *)
 end; (* while *)
end; (* procedure *)

(* >>>>>> FindAndPostInsert <<<<< *)

procedure FindAndPostInsert(var List:Tlist; Klic,Elem:TData);
var
 Found:Boolean;
 PomEl:TData;
 PomMark:TUK;
begin
 PomMark:=List.Act;
 First(List);
 Found:=false;
 while not Found and IsActive(List) do begin
 Copy(List,PomEl);
 if PomEl=Klic
 then begin
 PostInsert(List,Elem);
 Found:=true
 end else begin
 ActSucc(List)
 end; (* if *)
 end; (* while *)
 List.Act:=PomMark;
end; (* procedure *)

(* >>>>>> FindAndDelete <<<<<<< *)

procedure FindAndDelete(var List:TList;Klic:TData);
var
 UkL,UkP:TUK; (* dvojice soubeznych ukazatelu *)
 Found:Boolean;
begin
 if List.Zac=>nil (* je neprazdny? *)

```

```

then begin
 if List.Zac^.Data=Klic (* nasej?*)
 then begin
 UkP:=List.Zac;
 List.Zac:=List.Zac^.Uk;
 if UkP=List.Mark
 then List.MarkUsable:=false;
 if UkP=List.Kon
 then List.Kon:=nil;
 if UkP=List.Act
 then List.Act:=nil;
 dispose(UkP)
 end else begin
 UkL:=List.Zac;
 UkP:=List.Zac^.Uk;
 Found:=false;
 while not Found and (UkP<>nil)
 do begin
 if UkP^.Data=Klic
 then begin (* nasej! *)
 UkL^.Uk:=UkP^.Uk;
 if UkP=List.Mark
 then
 List.MarkUsable:=false;
 if UkP=List.Act
 then List.Act:=nil;
 if UkP=List.Kon
 then List.Kon:=UkL;
 dispose(UkP);
 Found:=true
 end else begin
 UkL:=UkP;
 UkP:=UkP^.Uk
 end (* if Uk *)
 end; (* while *)
 end (* if Zac^.Uk *)
 end (* if Zac<>nil *)
 end; (* procedure *)
(* >>>>> FindAndPostDelete <<<<< *)

procedure FindAndPostDelete(var List:TList; Klic:TData);
var
 PomEl:TData;
 Found:Boolean;
 PomMark:TUK;
begin
 PomMark:=List.Act;
 First(List);
 Found:=false;
 while not Found and IsActive(List)
 do begin
 Copy(List,PomEl);
 if PomEl=Klic
 then begin
 if List.Act^.Uk=PomMark
 then PomMark:=nil;
 PostDelete(List);
 Found:=true
 end else begin
 end;
 end;
end;

```

```

ActSucc(List)
end (* if PomEl *)
end; (* while *)
List.Act:=PomMark;
end; (* procedure *)

(* ===== Porovnavaci operace ===== *)
(* >>>>>>> EquList <<<<<<< *)

function EquList(L1,L2:TList) :Boolean;
var
 PomUk1,PomUk2:TUk;
 Equ:Boolean;
begin
 Equ:=true;
 PomUk1:=L1.Zac;
 PomUk2:=L2.Zac;
 while Equ and (PomUk1<>nil) and (PomUk2<>nil)
do begin
 if PomUk1^.Data=>PomUk2^.Data
 then Equ:=false
 else begin
 PomUk1:=PomUk1^.Uk;
 PomUk2:=PomUk2^.Uk;
 end; (* if *)
 end; (* while *)
 EquList:=Equ and (PomUk1=PomUk2);
end; (* function *)

(* >>>>>> EquListRec <<<<<< *)
(* Rekursivni varianta; Dva seznamy jsou ekvivalentni, jsou-li
oba prazne nebo jsou-li si rovny jejich prvni prvky
a soucasne zbyvajici casti seznamu *)

function EquListRec(L1,L2:TList):Boolean;
var
 PomUk1,PomUk2:TUk;

 function EquUkRec(Uk1,Uk2:TUk)
 :Boolean;
 (* Pomocna funkce, porovnavajici rekuzivne
 dva seznamy dane ukazateli *)
begin
 if (Uk1=nil) and (Uk2=nil)
 then EquUkRec:=true
 else begin
 if (Uk1<>nil) and (Uk2<>nil)
 then begin
 if Uk1^.Data=Uk2^.Data
 then EquUkRec:=
 EquUkRec(Uk1^.Uk,Uk2^.Uk)
 else EquUkRec:=false;
 end else EquUkRec:=false;
 end (* if *)
 end; (* pomocne function *)

 begin (* function EquListRec *)

```

```

PomUk1:=L1.Zac;
PomUk2:=L2.Zac;
EquListRec:=EquUkRec(PomUk1,PomUk2);
end; (* function *)

(* >>>>>> FirstLess <<<<<<< *)
(* Relace usporadani nad 2 seznamy *)

function FirstListLess(L1,L2:TList):Boolean;
var
 PomUk1,PomUk2:TUk;
 NotLess,NotGreater:Boolean;
begin
 NotLess:=true;
 NotGreater:=true;
 PomUk1:=L1.Zac;
 PomUk2:=L2.Zac;
 while (PomUk1<>nil) and (PomUk2<>nil)
 and NotLess and NotGreater
 do begin
 if PomUk1^.Data>PomUk2^.Data
 then begin
 NotGreater:=false;
 end else begin
 if PomUk1^.Data<PomUk2^.Data
 then begin
 NotLess:=false;
 end else begin
 PomUk1:=PomUk1^.Uk;
 PomUk2:=PomUk2^.Uk;
 end; (* *)
 end;
 end;
 FirstListLess:=NotGreater and
 (not NotLess or (PomUk2<>nil));
end; (* function *)

(* >>>>>> FirstListLessRec <<<<< *)
(* rekursivni zapis *)

function FirstListLessRec(L1,L2:TList):Boolean;
var PomUk1,PomUk2:TUk;

 function LessUkRec(Uk1,Uk2:TUk)
 :Boolean;
 (* Pomocna funkce porovnavajici rek.
 dva seznamy. *)
begin
 if (Uk1=nil) and (Uk2=nil)
 then LessUkRec:=false
 else begin
 if (Uk1<>nil) and (Uk2<>nil)
 then begin
 if Uk1^.Data=Uk2^.Data
 then LessUkRec:=
 LessUkRec(Uk1^.Uk,Uk2^.Uk)
 else LessUkRec:=
 Uk1^.Data<Uk2^.Data;
 end
 else LessUkRec:=Uk1=nil;
 end;
end;

```

```

end;
end; (* Pomocne function *)

begin (* function FirstListLessRec *)
PomUk1:=L1.Zac;
PomUk2:=L2.Zac;
FirstListLessRec:=
 LessUkRec(PomUk1,PomUk2);
end; (* function *)

(* >>>>>>>> ListCopy <<<<<<<< *)
(* Vytvoreni kopie seznamu *)

procedure ListCopy(List1:TList; var List2:TList);
var
 PomEl:TData;
begin
 InitList(List2); (* duplikat *)
 First(List1); (* prvni akt. v orig. *)
 while IsActive(List1) do begin
 Copy(List1,PomEl);
 InsertLast(List2,PomEl);
 ActSucc(List1);
 end; (* while *)
end; (* procedure *)

(* Varianta ListCopy bez InsertLast *)
{ InitList(L2);
 First(L1);
 if IsActive(List1)
 then begin
 CopyFirst(List1,PomEl);
 InsertFirst(List2);
 end; (* if *)
 ActSucc(List1);
 First(List2);
 while IsActive(List1) do begin
 Copy(List1,PomEl);
 PostInsert(List2,PomEl);
 ActSucc(List1);
 ActSucc(List2)
 end; (* while *)
(* konec Varianty bez InsertLast *)
}

(* >>>>>>>> ListCopyRec <<<<<<< *)
procedure ListCopyRec(L1:TList; var L2:TList);

procedure CopyRec(UkZac1:TUk; var UkZac2,NewKon:TUk);
(* Pomocna procedura rek. kopirujici seznam za pomoci ukazatelu *)
begin
 if UkZac1=nil
 then begin
 UkZac2:=nil;
 end else begin
 new(UkZac2);
 UkZac2^.Data:=UkZac1^.Data;
 NewKon:=UkZac2;
 CopyRec(UkZac1^.Uk,UkZac2^.Uk,NewKon)
 end
end;

```

```

 end (* if *)
end; (* Pomocne procedure *)

begin (* Procedure ListCopyRec *)
 InitList(L2);
 CopyRec (L1.Zac,L2.Zac,L2.Kon);
end;

(* >>>>>>> FindMaxSeq <<<<<<< *)
(* Nalezeni zacatku a delky nejdelsi neklesajici posloupnosti *)

procedure FindMaxSeq(L:TList; var Num,Len:TCardinal);
var
 N,PomLen,PomOrd,Old,New:integer;
begin
 First(L);
 Len:=0;
 Num:=0;
 N:=1;
 if IsActive(L)
 then begin
 Copy(L,Old);
 ActSucc(L);
 Num:=1;
 Len:=1;
 PomLen:=1;
 PomOrd:=1;
 while IsActive(L) do begin
 Copy(L,New);
 ActSucc(L);
 N:=N+1;
 if (New<Old) or not IsActive(L)
 then begin (* konec nekles. posl.
 nebo seznamu *)
 if New>=Old
 then PomLen:=PomLen+1;
 (* konec *)
 if PomLen>Len
 then begin (* nasla se delsi
 nekles. posloupnost *)
 Len:=PomLen;
 Num:=PomOrd;
 end; (* if PomDelka>Del *)
 PomLen:=1;
 PomOrd:=N;
 Old:=New;
 end else begin (* neni konec *)
 PomLen:=PomLen+1;
 Old:=New
 end (* (New < Old) or .. *)
 end (* while *)
 end (* If IsActive(L) *)
 end; (* procedure *)
(* ===== Operace s podseznamy ===== *)
(* >>>>>>>> Match <<<<<<< *)
(* Nalezeni pozice vyskytu podseznamu v seznamu *)

```

```

procedure Match(L,SubL:TList; var Pos:TCardinal);
var
 UkL,PomUkL,PomUkSub:TUK;
 Shoda,Nasel:Boolean;
begin
 Pos:=0;
 if SubL.Zac<>nil
 then begin
 Nasel:=false;
 UkL:=L.Zac;
 PomUkSub:=SubL.Zac;
 while (UkL<>nil) and not Nasel
 do begin
 if UKL^.Data=PomUkSub^.Data
 then begin
 Shoda:=true;
 PomUkL:=UkL;
 while Shoda and (PomUkL<>nil)
 and (PomUkSub<>nil)
 do begin
 Shoda:=PomUkL^.Data=PomUkSub^.Data;
 PomUkL:=PomUkL^.Uk;
 PomUkSub:=PomUkSub^.Uk;
 end;
 Nasel:=Shoda and (PomUkSub=nil);
 PomUkSub:=SubL.Zac;
 end;
 Pos:=Pos+1;
 UkL:=UkL^.Uk;
 end;
 if not Nasel
 then begin
 Pos:=0;
 end;
 end;
end;
(* >>>>>> InsertSubList <<<<<<< *)

procedure InsertSubList(var L:TList; SubList:TList; Num:TCardinal);
var
 NewSubL:TList;
 PomUk:TUK;
 Poc:TCardinal;
begin
 if SubList.Zac<>nil
 then begin
 InitList(NewSubL);
 ListCopy(SubList,NewSubL);
 if L.Zac=nil
 then begin
 L:=NewSubL;
 end else begin
 if Num=0
 then begin
 NewSubL.Kon^.Uk:=L.Zac;
 L.Zac:=NewSubL.Zac;
 end else begin
 PomUk:=L.Zac;
 Poc:=2;
 end;
 end;
 end;
end;

```

```

while (Poc<=Num)
 and (PomUk^.Uk<>nil)
do begin
 PomUk:=PomUk^.Uk;
 Poc:=Poc+1;
end; (* while *)
if PomUk=L.Kon
 then L.Kon:=NewSubL.Kon;
NewSubL.Kon^.Uk:=PomUk^.Uk;
PomUk^.Uk:=NewSubL.Zac;
end; (* if Num=0 *)
end; (* if *)
end;
end;

(* >>>>>> DeleteSublist <<<<<<< *)

procedure DeleteSublist(var L:TList; Num,Len:TCardinal);
var
 PomUk,DelUk,OverUk:TUk;
 PomNum:TCardinal;
begin
 PomUk:=L.Zac;
 PomNum:=2;
 while (PomNum<Num) and (PomUk<>nil)
 do begin
 PomUk:=PomUk^.Uk;
 PomNum:=PomNum+1;
 end; (* while *)
 if PomUk<>nil
 then begin
 if Num<2
 then OverUk:=L.Zac
 else OverUk:=PomUk^.Uk;
 while (OverUk<>nil) and (Len>0)
 do begin
 DelUk:=OverUk;
 OverUk:=OverUk^.Uk;
 if L.Mark=DelUk
 then begin
 L.MarkUsable:=false;
 end; (* if *)
 if L.Act=DelUk
 then begin
 L.Act:=nil;
 end; (* if *)
 if L.Kon=DelUk
 then begin
 if Num<2
 then L.Kon:=nil
 else L.Kon:=PomUk;
 end; (* if L.Kon=DelUk *)
 dispose(DelUk);
 Len:=Len-1;
 end; (* if *)
 if Num<2
 then L.Zac:=OverUk
 else PomUk^.Uk:=OverUk;
 end; (* while *)
end; (* procedure *)

```

```
(* >>>>>>>> CopySublist <<<<<< *)

procedure CopySublist(L:TList; var SubL:TList; Num,Len:TCardinal);
var
 PomUk,NewUk:TUK;
 PomNum:TCardinal;
begin
 begin
 InitList(SubL);
 if Len<>0
 then begin
 PomUk:=L.Zac;
 PomNum:=1;
 while (PomNum<Num) and (PomUk<>nil)
 do begin
 PomUk:=PomUk^.Uk;
 PomNum:=PomNum+1;
 end; (* while *)
 if PomUk<>nil
 then begin
 new(SubL.Zac);
 NewUk:=SubL.Zac;
 NewUk^.Data:=PomUk^.Data;
 PomUk:=PomUk^.Uk;
 Len:=Len-1;
 while (PomUk<>nil) and (Len>0)
 do begin
 new(NewUk^.Uk);
 NewUk:=NewUk^.Uk;
 NewUk^.Data:=PomUk^.Data;
 PomUk:=PomUk^.Uk;
 Len:=Len-1;
 end; (* while *)
 NewUk^.Uk:=nil;
 SubL.Kon:=NewUk;
 end; (* if PomUk<> nil *)
 end; (* if Len <> 0 *)
 end; (* procedure *)

(* === Procedure pro rozdelovani === *)
(* ===== a spojovani seznamu ===== *)

(* >>>>>>>> Concat <<<<<< *)

procedure Concat(var L1,L2:TList);
begin
 if L1.Zac=nil
 then begin
 L1:=L2;
 L1.Act:=nil;
 L1.MarkUsable:=false;
 end else begin
 L1.Kon^.Uk:=L2.Zac;
 if L2.Kon<>nil
 then L1.Kon:=L2.Kon;
 end; (* if *)
 InitList(L2);
end; (* procedure *)

(* >>>>>>> FindAndDecat <<<<<< *)
```

```

procedure FindAndDecat(var L1,L2:TList; Klic:TData);
(* Najde-li se zadany klic, rozdeli se seznam tak, ze druhý zacina klicem *)
var PomUk:TUK;
 MarkUs,Nasel:Boolean;
begin
 InitList(L2);
 PomUk:=L1.Zac;
 Nasel:=false;
 while (PomUk<>nil) and not Nasel
 do begin
 if PomUk^.Data=Klic
 then begin
 Nasel:=true;
 L2.Zac:=PomUk;
 L2.Kon:=L1.Kon;
 L1.Act:=nil;
 if L1.Zac=L2.Zac
 then begin
 InitList(L1)
 end else begin
 PomUk:=L1.Zac;
 MarkUs:=false;
 while PomUk^.Uk<>L2.Zac
 do begin
 if PomUk=L1.Mark
 then begin
 MarkUs:=true;
 end; (* if *)
 PomUk:=PomUk^.Uk;
 end; (* while *)
 end; (* procedure *)
 end;
 MarkUs:=MarkUs or (PomUk=L1.Mark);
 L1.Kon:=PomUk;
 L1.Kon^.Uk:=nil;
 L1.MarkUsable:=
 L1.MarkUsable and MarkUs;
 end; (* if L1.Zac=L2.Zac *)
 end else begin
 PomUk:=PomUk^.Uk;
 end; (* if PomUk^.Data=Klic *)
end; (* while *)
end; (* while *)
(* >>>>>>>> MergeLists <<<<<<< *)

procedure MergeLists (var LSource1,LSource2,LDest:TList);
var
 UkSou1,UkSou2,UkDest:TUK;
 Dat:TData;

procedure CompWriteMove
 (var UK1,UK2:TUK; var Dat:TData);
(* Pomocna procedura, ktera porovna dva prvky,
 mensi prvek zapise a posune se dal v seznamu *)
begin
 if Uk1<>nil
 then begin
 if Uk2<>nil
 then begin

```

```

if Uk1^.Data<Uk2^.Data
then begin
 Dat:=Uk1^.Data;
 Uk1:=Uk1^.Uk;
end else begin
 Dat:=Uk2^.Data;
 Uk2:=Uk2^.Uk;
end; (* if Uk1^.Data<Uk2^.Data *)
end else begin (* else Uk2<> nil *)
 Dat:=Uk1^.Data;
 Uk1:=Uk1^.Uk;
end; (* else Uk2<> nil *)
end else begin (* else Uk1<> nil *)
 Dat:=Uk2^.Data;
 Uk2:=Uk2^.Uk;
end; (* else Uk1<> nil *)
end; (* Pomocna procedure *)

begin (* procedure MergeLists *)
 InitList(Ldest);
 UkSOU1:=LSource1.Zac;
 UkSOU2:=LSource2.Zac;
 if (UkSOU1<>nil) or (UkSOU2<>nil)
 then begin
 new (LDest.Zac);
 UkDest:=LDest.Zac;
 CompWriteMove
 (UkSOU1,UkSOU2,UkDest^.Data);
 while (UkSOU1<>nil) or (UkSOU2<>nil)
 do begin
 CompWriteMove(UkSOU1,UkSOU2,Dat);
 if Dat<>UkDest^.Data
 then begin
 new(UkDest^.Uk);
 UkDest:=UkDest^.Uk;
 UkDest^.Data:=Dat;
 end; (* if Dat<> *)
 end; (* while *)
 UkDest^.Uk:=nil;
 LDest.Kon:=UkDest;
 end; (* if *)
 Clear(LSource1);
 Clear(LSource2);
end; (* procedure *)

(* >>>>>> DecatListToSeq <<<<<< *)

procedure DecatListToSeq (SourceList:Tlist; var DestList:TUKLL);
var
 UkToSeg:TUKLL;
 PomUk,NewUk:TUK;
 Old:TData;
 TooGreater:Boolean;
begin
 PomUk:=SourceList.Zac;
 DestList:=nil;
 while PomUk<>nil
 do begin
 if DestList=nil

```

```

then begin
 new(DestList);
 UkToSeg:=DestList;
end else begin
 new(UkToSeg^.Next);
 UkToSeg:=UkToSeg^.Next;
end; (* if DestList=nil *)

UkTOSeg^.Next:=nil;
InitList(UkToSeg^.List);
new(UkToSeg^.List.Zac);
NewUk:=UkToSeg^.List.Zac;
Old:=PomUk^.Data;
NewUk^.Data:=Old;
PomUk:=PomUk^.Uk;
TooGreater:=true;
while (PomUk<>nil) and TooGreater
do begin
 if PomUk^.Data<Old
 then begin
 TooGreater:=false;
 end else begin
 new(NewUk^.Uk);
 NewUk:=NewUk^.Uk;
 Old:=PomUk^.Data;
 NewUk^.Data:=Old;
 PomUk:=PomUk^.Uk;
 end; (* if PomUk^.Data < Old *)
end; (* while (PomUk^.Data<Old) *)
NewUk^.Uk:=nil;
UkToSeg^.List.Kon:=NewUk;
end; (* while PomUk <> nil*)
end; (* procedure *)

(* ====== Procedure pro praci ===== *)
(* ====== se seznamem seznamu ===== *)

(* >>>>> InitListOfLists <<<<<< *)

procedure InitListOfLists(LL:TUKLL);
begin
 LL:=nil;
end;

(* >>>>> ClearListOfLists <<<<<< *)

procedure ClearListOfLists(var LL:TUKLL);
var
 DelUkLL:TUKLL;
begin
 while LL<>nil
 do begin
 DelUkLL:=LL;
 LL:=LL^.Next;
 Clear(DelUkLL^.List);
 dispose(DelUkLL);
 end;
end;

(* >>>>> LengthListOfLists <<<<<< *)

```

```
(* Funkce vraci pocet seznamu v seznamu seznamu *)

function LengthListOfLists(LL:TUKLL):TCardinal;
var Num:TCardinal;
begin
Num:=0;
while LL<>nil
do begin
 Num:=Num+1;
 LL:=LL^.Next;
end;
LengthListOfLists:=Num;
end;

(* >>>>>>> CopyNthList <<<<<<< *)

procedure CopyNthList(LL:TUKLL; Num:TCardinal; var L:TList);
var
I:TCardinal;
begin
InitList(L);
if Num>0
then begin
 I:=1;
 while (LL<>nil) and (I<Num)
do begin
 LL:=LL^.Next;
 I:=I+1;
end; (* while *)
if LL<>nil
 then ListCopy(LL^.List,L);
end; (* if *)
end; (* procedure *)

end.(* unit *)
```

### 9.1.2. Dvojsměrný seznam

**Dvojsměrný  
seznam**

```
unit Dlet_un;

interface
(* >>>>>>> Pouzite typy <<<<<< *)
type
 TUk=^TPrek; (* Ukazatel na prvek
 jednosmer. seznamu *)
 TData=integer; (* Datova sl. prvku *)
 TDUk=^TDPrek; (* Ukazatel na prvek
 dvojsmerneho seznamu *)
 TUKLL=^TListOfLists;
 (* Ukazatel na seznam
 seznamu *)

TCardinal=0..MaxInt;

TPrek= record (* Typ prvku jedno-
 smerneho seznamu *)
 Ukk:TUk;
 Data:TData;
end; (* record *)
```

```

TDPrek=record (* Typ prvku dvou-
smerneho seznamu *)
 LUK,PUK:TDUK;
 Data:TData;
end; (* record *)

TList=record (* ATD jednosm. seznam *)
 Zac,Kon,Act,Mark:TUK;
 MarkUsable:Boolean;
 Len:TCardinal;
end; (* record *)

(* Do Mark se zaznamena stav aktivity
operaci Mark.
Promenna MarkUsable je pro test
pouzitelnosti operace ActSetMarked;
nastavi se na false, rysi-li se
zaznamenany prvek.
Len je delka seznamu, zde nevyuzivana..*)

TDList=record (* ATD dvojsm. seznam *)
 Zac,Kon,Act,Mark:TDUK;
 MarkUsable:Boolean;
 Len:TCardinal;
end; (* record *)

TListOfLists=record (* typ seznam seznamu *)
 Next:TUkLL;
 List:TDList;
end;

(*****)
(***Dvojsmerný seznam *****)
(*****)

procedure DInitList(var DList:TDList);
procedure DInsertFirst(var DList:TDList; Elemt:TData);
procedure DInsertLast(var DList:TDList; Elemt:TData);
procedure DDeleteLast(var DList:TDList);
procedure DDeleteFirst(var DList:TDList);
procedure DActSucc(var DList:TDList);
procedure DActPred(var DList:TDList);
function DIsActive(DList:TDList):Boolean;
procedure DFFirst(var DList:TDList);
procedure DLast(var DList:TDList);
procedure DCopyFirst(DList:TDList; var Elemt:TData);
procedure DCopyLast(DList:TDList; var Elemt:TData);
procedure DPostInsert(var DList:TDList; Elemt:TData);
procedure DPostDelete(var DList:TDList);
procedure DPreInsert(var DList:TDList; Elemt:TData);
procedure DPreDelete(var DList:TDList);
procedure DActualize (DList:TDList; Elemt:TData);
procedure DCopy(DList:TDList; var Elemt:TData);
function DLengthList(DList:TDList) :TCardinal;
function DIsEmpty(DList:TDList):Boolean;
procedure DMarkAct(var List:TDList);
procedure DSetActMarked(var List:TDList);
function DIsMarkUsable(List:TDList):Boolean;
procedure DListCopy(DL1:TDList; var DL2:TDList);

```

```

procedure DFindAndRewrite(DList:TDlist;
 Klic,Elem:TData);
procedure DFindAndPostInsert(var DList:TDlist;
 Klic,Elem:TData);
procedure DFindAndDelete
 (var DList:TDList;Klic:TData);
procedure DFindAndPostDelete
 (var DList:TDList; Klic:TData);
procedure DFindAndPreInsert
 (var DList:TDList; Klic,Elem:TData);
procedure DFindAndPreDelete
 (var DList:TDList; Klic:TData);
procedure DClear(var List:TDList);

implementation

(*>>>>>>>>DInitList>>>>>>>>>>>>*)

procedure DInitList(var DList:TDList);
(* Inicializace dvojsmerneho seznamu *)
begin
 DList.Zac:=nil;
 DList.Kon:=nil;
 DList.Act:=nil;
 DList.Mark:=nil;
 DList.MarkUsable:=false;

end;
(*>>>>>>>>DInsertFirst>>>>>>>>>>*)

procedure DInsertFirst(var DList:TDlist;
 Elemt:TData);
(* Vlozeni prvniho prvku *)
var
 DPomUk:TDuk;
begin
 new(DPomUk);
 with DPomUk^ do
 begin
 Data:=Elem;
 LUK:=nil;
 PUK:= DList.Zac;
 end; (* with *)

 if DList.Kon=nil (* je prazdny ? *)
 then
 DList.Kon:=DPomUk;
 else
 DList.Zac^.LUk:=DPomUk;

 DList.Zac:=DPomUk;

end; (* procedure *)

(*>>>>>>>>DInsertLast>>>>>>>>>>>*)

procedure DInsertLast(var DList:TDList;
 Elemt:TData);

```

```

(* Vlozeni posledniho prvku *)
var
 DPomUk:TDUk;
begin
 new(DPomUk);
 with DPomUk^ do
 begin
 Data:=Elem;
 PUk:=nil;
 Luk:= DList.Kon;
 end; (* with *)
 if DList.Kon=nil (* je prazdny? *)
 then DList.Zac:=DPomUk
 else DList.Kon^.PUk:=DPomUk;
 DList.Kon:=DPomUk;
end; (* procedure *)

(*>>>>>>>> DDeleteLast >>>>>>>>>*)

procedure DDeleteLast(var DList:TDList);
(* Zruseni posledniho prvku *)
var
 DPomUk:TDUk;
begin
 with DList do
 begin
 if Kon<>nil (* je prazdny ? *)
 then begin
 DPomUk:=Kon;
 if Zac=Kon (* obsahuje jediny ? *)
 then begin
 Zac:=nil;
 Kon:=nil;
 Act:=nil;
 MarkUsable:=false;
 end else begin
 if Kon=Act (* rusi se aktivni?*)
 then Act:=nil;
 if Kon=Mark
 then MarkUsable:=false;
 Kon:= DPomUk^.Luk;
 Kon^.PUk:=nil;
 end; (* if Zac= *)
 dispose(DPomUk);
 end; (* if Kon<> *)
 end; (* with *)
 end; (* procedure *)

(*>>>>>>>> DDeleteFirst >>>>>>>>*)

procedure DDeleteFirst
 (var DList:TDList);
(* Zruseni prvniho prvku *)
var
 DPomUk:TDUk;
begin
 with DList do
 begin
 if Zac<>nil (* je prazdny ? *)
 then begin

```





```

new(DPomUk);
DPomUk^.Data:=Elem;
DPomUk^.LUk:=Act;
DPomUk^.PUk:=Act^.PUk;
Act^.PUk:=DPomUk;
if Act<>Kon (* není aktivní
 poslední? *)
then DPomUk^.PUk^.LUk:=DPomUk
else Kon:=DPomUk
end (* if *)
end (* with *)
end; (* procedure *)

(* >>>>DPostDelete >>>>>>>>>>>>*)

procedure DPostDelete(var DList:TDlist);
(* Zrusení prvku za aktivním *)
var
 DPomUk:TDUk;
begin
 with DList do
begin
 (* je aktivní? *)
 if Act<>nil
 then begin
 (* má aktivního nasledníka? *)
 if Act^.PUk<>nil
 then begin (* má.. *)
 DPomUk:=Act^.PUk;
 Act^.PUk:=DPomUk^.PUk;
 (* byl růzený poslední? *)
 if DPomUk=Mark
 then MarkUsable:=false;
 if Kon=DPomUk
 then Kon:=Act (* byl.. *)
 else
 DPomUk^.PUk^.LUk:=DPomUk^.LUk;
 dispose(DPomUk)
 end (* if *)
 end (* if *)
 end (* with *)
end; (* procedure *)

(*>>>>>>>DPreInsert>>>>>>>>>>>>*)

procedure DPreInsert(var DList:TDlist;
 ELEM:TData);
(* Vložení pred aktivní prvek *)
var
 DPomUk:TDUk;
begin
 with DList do
begin
 if Act<>nil (* je aktivní ? *)
 then begin
 new(DPomUk);
 DPomUk^.Data:=Elem;
 DPomUk^.PUk:=Act;
 DPomUk^.LUk:=Act^.LUk;
 Act^.LUk:=DPomUk;
 end
end

```

```

if Act<>Zac (* není aktivní
 první? *)
then DPomUk^.LUk^.PUk:=DPomUk
else Zac:=DPomUk
end (* if *)
end (* with *)
end; (* procedure *)

(* >>>>DPreDelete >>>>>>>>>>*)

procedure DPreDelete(var DList:TList);
(* Zrusení prvku pred aktivním *)
var
 DPomUk:TUk;
begin
 with DList do
 begin
 (* je aktivní? *)
 if Act<>nil
 then begin (* je.. *)
 (* má aktivní predchůdce? *)
 if Act^.LUk<>nil
 then begin (* má .. *)
 DPomUk:=Act^.LUk;
 Act^.LUk:=DPomUk^.LUk;
 (* byl růzený první? *)
 if DPomUk=Mark
 then MarkUsable:=false;
 if Zac=DPomUk
 then Zac:=Act (* byl .. *)
 else
 DPomUk^.LUk^.PUk:=DPomUk^.PUk;
 dispose(DPomUk)
 end (* if Act^.LUk<>nil *)
 end (* if Avt<> nil *)
 end (* with *)
 end; (* procedure *)

(*>>>>>>>>>DActualize>>>>>>>>>*)

procedure DActualize (DList:TList;
 Elemt:TData);
(* Prepis obsahu aktivního prvku *)
begin
 if DList.Act<>nil
 then DList.Act^.Data:=Elemt
end; (* procedure *)

(*>>>>>>>>>DCopy>>>>>>>>>>>>*)

procedure DCopy(DList:TList;
 var Elemt:TData);
(* Obsah aktivního prvku *)
begin
 if DList.Act<>nil
 then begin
 Elemt:=DList.Act^.Data;
 end (* if *)
end; (* DCopy *)

```

```

(*>>>>>>DLenghtList>>>>>>>>>>>>*)

function DLenghtList(DList:TList)
 :TCardinal;
(* Delka seznamu *)
var
 n:TCardinal;
 DPomUk:TList;
begin
 n:=0;
 DPomUk:=DList.Zac;
 while DPomUk<>nil do
 begin
 DPomUk:=DPomUk^.PUK;
 n:=n+1
 end; (* while *)
 DLenghtList:=n
end; (* function *)

(*>>>>>DIsEmpty>>>>>>>>>>>>>>>*)

function DIsEmpty(DList:TList):Boolean;
(* Test praznosti seznamu *)
begin
 DIsEmpty:=DList.Zac=nil
end; (* function *)

(*>>>>>DMarkAct>>>>>>>>>>>>>>>*)

procedure DMarkAct(var List:TList);
(* Zaznam aktivity *)
begin
 with List
 do begin
 Mark:=Act;
 MarkUsable:=Act<>nil;
 end; (* with *)
end; (* procedure *)

(*>>>>>>>>>>DSetActMarked>>>>>>>*)

procedure DSetActMarked
 (var List:TList);
(* Navrat zaznamenane aktivity *)
begin
 with List
 do begin
 if MarkUsable
 then begin
 Act:=Mark;
 end
 else begin
 Act:=nil;
 end; (* if *)
 end; (* with *)
 end; (* procedure *)

```

```

(List:TDLList):Boolean;
(* Zda je zaznamenana aktivita *)
begin
 DLsMarkUsable:=List.MarkUsable;
end;

(*>>>>>>DListCopy>>>>>>>>>>*)

procedure DListCopy(DL1:TDLList;
 var DL2:TDLList);
(* Kopie seznamu *)
var
 Err:Boolean;
 PomEl:TData;
begin
 DInitList(DL2);
 DFirst(DL1);
 while DLsActive(DL1) do begin
 DCopy(DL1,PomEl);
 DInsertLast(DL2,PomEl);
 DActSucc(DL1);
 end; (* while *)
 end; (* procedure *)

(*>>>>>>DFindAndRewrite>>>>>>>*)

procedure DFindAndRewrite(DList:TDLlist;
 Klic,Elem:TData);
(* Najdi klic a prepis obsah prvku *)
var
 Found:Boolean;
 PomEl:TData;
begin
 DFirst(DList);
 Found:=false;
 while DLsActive(DList)
 and not Found do
 begin
 DCopy(DList,PomEl);
 if PomEl=Klic (* nasel ? *)
 then begin
 Found:=true;
 DActualize(DList,Elem)
 end else begin
 DActSucc(DList);
 end; (* if *)
 end; (* while *)
 end; (* procedure *)

(*>>>>>>DFindAndPostInsert>>>>>*)

procedure DFindAndPostInsert
 (var DList:TDLlist;
 Klic,Elem:TData);
(* Najdi a vloz za nej prvek *)
var
 Found:Boolean;
 PomEl:TData;
 PomMark:TDUk;
begin

```

```

PomMark:=DList.Act;
DFirst(DList);
Found:=false;
while not Found
 and DIsActive(DList) do
begin
 DCopy(DList,PomEl);
 if PomEl=Klic
 then begin
 DPostInsert(DList,Elem);
 Found:=true
 end else begin
 DActSucc(DList)
 end; (* if *)
end; (* while *)
DList.Act:=PomMark;
end; (* procedure *)

(*>>>>>>>>>DFindAndDelete>>>>>>>>*)

procedure DFindAndDelete
 (var DList:TDLList;Klic:TData);
(* Najdi a zrus *)
var
 Found:Boolean;
 PomEl:TData;
 PomMark:TDUk;
begin
 PomMark:=DList.Act;
 DFirst(DList);
 if not DIsEmpty(DList)
 then begin (* není prázdný *)
 DCopyFirst(DList,PomEl);
 if PomEl=Klic (* rovná se prvni ? *)
 then begin
 if PomMark=DList.Zac
 then PomMark:=nil;
 DDeleteFirst(DList);
 end else begin
 DActSucc(DList);
 Found:=false;
 while
 DIsActive(DList) and not Found
 do begin
 DCopy(DList,PomEl);
 if PomEl=Klic (* nasel ? *)
 then begin
 if PomMark=DList.Act
 then PomMark:=nil;
 DActPred(DList);
 DPostDelete(DList);
 Found:=true
 end else begin
 DActSucc(DList)
 end (* if PomEl = Klic *)
 end (* while *)
 end (* PomEl= Klic *)
 end; (* if notDIsEmpty ..*)
 DList.Act:=PomMark;
end; (* procedure *)

```

```
(*>>>>>DFindAndPostDelete>>>>>>*)
procedure DFindAndPostDelete
 (var DList:TDLList; Klic:TData);
(* Najdi a zrus prvek za nim *)
var
 Found:Boolean;
 PomEl:TData;
 PomMark:TDUk;
begin
 PomMark:=DList.Act;
 DFirst(DList);
 Found:=false;
 while not Found and DIsActive(DList)
 do begin
 DCopy(DList,PomEl);
 if PomEl=Klic (* nasel ? *)
 then begin
 if PomMark=DList.Act^.PUK
 then PomMark:=nil;
 DPostDelete(DList);
 Found:=true
 end else begin
 DActSucc(DList)
 end (* if *)
 end; (* while *)
 DList.Act:=PomMark;
end; (* procedure *)

(*>>>>>>DFindAndPreInsert>>>>>>*)

procedure DFindAndPreInsert
 (var DList:TDLList; Klic,Elem:TData);
(* Najdi a vloz pred nej prvek *)
var
 Found:Boolean;
 PomEl:TData;
 PomMark:TDUk;
begin
 PomMark:=DList.Act;
 DFirst(DList);
 Found:=false;
 while not Found and DIsActive(DList)
 do begin
 DCopy(DList,PomEl);
 if PomEl=Klic (* nasel ? *)
 then begin
 DPreInsert(DList,Elem);
 Found:=true
 end else begin
 DActSucc(DList)
 end (* if *)
 end; (* while *)
 DList.Act:=PomMark;
end; (* procedure *)

(*>>>>>> DFindAndPreDelete >>>>*)

procedure DFindAndPreDelete
 (var DList:TDLList; Klic:TData);
```

```
(* Najdi a zrus prvek pred nim *)
var
 Found:Boolean;
 PomEl:TData;
 PomMark:TDUk;
begin
 PomMark:=DList.Act;
 DFirst(DList);
 Found:=false;
 while not Found and DIsActive(DList)
 do begin
 DCopy(DList,PomEl);
 if PomEl=Klic (* nasel ? *)
 then begin
 if PomMark=DList.Act^.LUk
 then PomMark:=nil;
 DPreDelete(DList);
 Found:=true;
 end else begin
 DActSucc(DList)
 end (* if *)
 end; (* while *)
 DList.Act:=PomMark;
end; (* procedure *)

(* >>>>>>>>> DClear <<<<<<<<< *)

procedure DClear(var List:TDList);
(* Zrus seznam *)
var
 PomUk,DelUk:TDUk;
begin
 PomUk:=List.Zac;
 while PomUk<>nil
 do begin
 DelUk:=PomUk;
 PomUk:=PomUk^.PUk;
 dispose(DelUk);
 end; (* while *)
 with List
 do begin
 Zac:=nil;
 Kon:=nil;
 Act:=nil;
 Mark:=nil;
 MarkUsable:=false;
 end; (* with *)
end; (* procedure *)

(*<<<<<<<<<<<<<<<<<<<<<<<<<<<<*)
end. (* unit *)
```

### 9.1.3. Kruhový seznam

### 9.1.3. Kruhový seznam

unit Oslet\_un;  
interface

(\* >>>>>>> Pouzite typy <<<<<<< \*)

type

```

TUk=^TPrvek; (* Ukazatel na prvek jednosmer. seznamu *)
TData=integer; (* Datova slozka prvku *)

TDUk=^TDPPrvek; (* Ukazatel na prvek dvojsmerneho seznamu *)
TUKLL=^TListOfLists;
(* Ukazatel na seznam seznamu *)

TCardinal=0..MaxInt;

TPrvek= record (* Typ prvku jednosmerneho seznamu *)
 Uk:TUk;
 Data:TData;
end; (* record *)

TDPPrvek=record (* Typ prvku dvousmerneho seznamu *)
 LUk,Puk:TDUk;
 Data:TData;
end; (* record *)

TList=record (* ATD jednosm. seznam *)
 Zac,Kon,Act,Mark:TUk;
 MarkUsable:Boolean;
 Len:TCardinal;
end; (* record *)

(* Do Mark se zaznamena stav aktivity
operaci Mark.
Promenna MarkUsable je pro test pouzitelnosti operace ActSetMarked;
nastavi se na false, rusi-li se zaznamenany prvek. Len je delka
seznamu, nevyuzivana..*)

TDList=record (* ATD dvojsm. seznam *)
 Zac,Kon,Act,Mark:TDUk;
 MarkUsable:Boolean;
 Len:TCardinal;
end; (* record *)

TListOfLists=record
 Next:TUKLL;
 List:TDList;
end; (* record *)

(*****)

procedure InitList(var List:Tlist);
procedure InsertFirst(var List:TList; Elemt:TData);
procedure ActSucc(var List:TList);
function IsActive(List:TList):Boolean;
procedure First(var List:TList);
procedure PostInsert(var List:TList; Elemt:TData);

procedure PostDelete(var List:TList);
(* atd*)

(* * Uzazky operaci pro jednosm. seznam *)
(* ***** s hlavickou *****)

procedure InitListHead(var List:Tlist);
function LengthListHead(List:TList):TCardinal;

```

```

procedure InsertFirstHead(var List:TList; Elemt:TData);
procedure FindAndDeleteHead(var List:TList; Klic:TData);
procedure DeleteFirstHead(List:TList);
(* atd*)

(**** Operace pro kruhovy seznam ****)
(***** s hlavickou *****)

procedure InitListCircHead (var List:TList);
procedure InsertFirstCircHead (var List:TList; Elemt:TData);
procedure FirstCircHead(var List:TList);
procedure SuccCircHead(var List:TList);
function LengthListCircHead(List:TList):TCardinal;
procedure ListCopyCirc(List1:TList; var List2:TList);
procedure Copy(List:TList; var Elemt:TData);

implementation

(* >>>>>>>> InitList <<<<<<<<< *)

procedure InitList(var List:Tlist);
begin
 List.Zac:=nil;
 List.Act:=nil;
 List.Kon:=nil;
 List.Mark:=nil;
 List.MarkUsable:=false;
end; (* procedure InitList *)

(* >>>>>>> InsertFirst <<<<<<<< *)

procedure InsertFirst(var List:TList; Elemt:TData);
var PomUk:TUk;
begin
 new(PomUk);
 PomUk^.Data:=Elemt;
 PomUk^.Uk:=List.Zac;
 List.Zac:=PomUk;
 If List.Kon=nil then List.Kon:=PomUk;
end;

(* >>>>>>> ActSucc <<<<<<<<< *)

procedure ActSucc(var List:TList);
begin
 with List do
 begin
 if Act<>nil
 then Act:=Act^.Uk
 end;
end;

(* >>>>>>> IsActive <<<<<<<<<< *)

function IsActive(List:TList):Boolean;
begin
 IsActive:=List.Act<>nil
end; (*function *)

```

(\* >>>>>>> First <<<<<<<<< \*)

```
procedure First(var List:TList);
begin
 if List.Zac<>nil
 then List.Act:=List.Zac
 else List.Act:=nil;
end;
```

(\* >>>>>>> PostInsert <<<<<<< \*)

```
procedure PostInsert(var List:TList;
 Elemt:TData);
var
 PomUk:TUk;
begin
 with List do
 begin
 if Act<>nil
 then begin
 new(PomUk);
 PomUk^.Data:=Elemt;
 PomUk^.Uk:=Act^.Uk;
 Act^.Uk:=PomUk;
 if Act=Kon
 then Kon:=PomUk;
 end; (* if *)
 end (* with *)
 end; (* procedure *)
```

(\* >>>>>>> PostDelete <<<<<<< \*)

```
procedure PostDelete(var List:TList);
var
 PomUk:TUk;
begin
 with List do
 begin
 if Act<>nil
 then begin
 PomUk:=Act^.Uk;
 if PomUk<>nil
 then begin
 Act^.Uk:=PomUk^.Uk;
 if Kon=PomUk
 then Kon:=Act;
 if PomUk=Mark
 then begin
 Mark:=nil;
 Markusable:=false
 end;
 dispose(PomUk);
 end (* if PomUk *)
 end (* if Act *)
 end (* with *)
end; (* procedure *)
```

(\* Ukazky operaci pro jednosm. seznam \*)
(\*\*\*\*\*\* s hlavickou \*\*\*\*\*)

```
(*>>>>>>>InitListHead>>>>>>>>>>*)

procedure InitListHead(var List:Tlist);
(* Hlavicka bude obsahovat -maxint. Prvkiem seznamu by mohl byt take zaznam
s variantou. Jednou variantou by mohla byt hlavicka a druhou vlastni datova
strukturna seznamu *)
begin
 InitList(List);
 InsertFirst(List,-maxint);
end; (* procedure *)

(*>>>>>>LengthListHead>>>>>>>>>>*)

function LengthListHead(List:TList):TCardinal;
var N:TCardinal;
begin
 N:=0;
 First(List);
 ActSucc(List);
 while IsActive(List) do begin
 ActSucc(List);
 N:=N+1
 end; (* while *)
 LengthListHead:=N;
end; (* procedure *)

(*>>>>>>>InsertFirstHead>>>>>>>>>*)

procedure InsertFirstHead(var List:TList; Elemt:TData);
begin
 First(List);
 PostInsert(List,Elemt)
end; (* procedure *)

(*>>>>>>>FindAndDelete>>>>>>>>>*)

procedure FindAndDeleteHead(var List:TList; Klic:TData);
(**)
var
 UkL,UkP:TUk; (* dvojice soubeznych ukazatelu *)
 Found:Boolean;
 PomEl:TData;
begin
 UkL,UkP:=List.Zac;
 UkP:=List.Zac^.Uk;
 Found:=false;
 while not Found and (UkP<>nil) do begin
 if UkP^.Data=Klic
 then begin (* nasej! *)
 UkL^.Uk:=UkP^.Uk;
 if UkP=List.Kon
 then List.Kon:=UkL;
 dispose(UkP);
 Found:=true
 end else begin
 UkL:=UkP;
 end
 end
end;
```

```

UkP:=UkP^.Uk;
end (* if Uk *)
end; (* while *)
end; (* procedure *)

(*>>>>>>>DeleteFirstHead>>>>>>>>>*)

procedure DeleteFirstHead(List:TList);
begin
 First(List);
 PostDelete(List)
end; (* procedure *)

(**** Operace pro kruhovy seznam *****)
(***** s hlavickou *****)

(*>>>>>>>InitListCircHead>>>>>>>>*)

procedure InitListCircHead
 (var List:TList);
var
 PomUk:TUk;
begin
 New(PomUk);
 PomUk^.Data:=-maxint;
 PomUk^.Uk:=PomUk;
 List.Zac:=PomUk;
 List.Act:=List.Zac;(* nulova aktivita
 ukazuje na hlavicku ! *)
end; (* procedure *)

(*>>>>>>InsertFirstCircHead>>>>>>*)

procedure InsertFirstCircHead(var List:TList;Elem:TData);
(* Vloz za hlavicku *)
var
 PomUk:TUk;
begin
 New(PomUk);
 PomUk^.Data:=Elem;
 PomUk^.Uk:=List.Zac^.Uk;
 List.Zac^.Uk:=PomUk;
end; (* procedure *)

(*>>>>>>FirstCircHead>>>>>>>>>*)

procedure FirstCircHead(var List:TList);
begin
 List.Act:=List.Zac^.Uk;
 if List.Act=List.Zac then List.Act:=nil;
end;

(*>>>>>> SuccCircHead >>>>>>>>>*)

procedure SuccCircHead(var List:TList);
begin
 if List.Act<>nil
 then begin
 List.Act:=List.Act^.Uk;
 end;

```

```

if List.Act=List.Zac (* ukazuje na
 hlavicku ? *)
then List.Act:=List.Act^.Uk
end (* if *)
end; (* procedure *)

(*>>>>>LengthListCircHead>>>>>>>>*)

function LengthListCircHead(List:TList):TCardinal;
var
 n:TCardinal;
 PomUk:TUk;
begin
 n:=0;
 PomUk:=List.Zac^.Uk;
 while PomUk<>List.Zac do begin
 n:=n+1;
 PomUk:=PomUk^.Uk
 end; (* while *)
 LengthListCircHead:=n
end; (* function *)

(*>>>>>>ListCopyCirc>>>>>>>>>>>*)

procedure ListCopyCirc(List1:TList; var List2:TList);
(* S pouzitim ukazatelu *)
var
 PomUk1,PomUk2:TUk;
begin
 if List1.Zac=nil
 then begin
 List2.Zac:=nil;
 List2.Act:=nil;
 List2.Kon:=nil
 end else begin
 new(PomUk2);
 List2.Zac:=PomUk2;
 List2.AcT:=PomUk2;
 PomUk1:=List1.Zac;
 PomUk2^.Data:=PomUk1^.Data;
 PomUk1:=PomUk1^.Uk;
 while PomUk1<>List1.Zac do begin
 new(PomUk2^.Uk);
 PomUk2^.Uk:=PomUk2^.Uk;
 PomUk2^.Data:=PomUk1^.Data;
 PomUk1:=PomUk1^.Uk;
 end; (* while , PomUk1 ukazuje na
 zacatek *)
 PomUk2^.Uk:=List2.Zac; (* uzavreni
 kruhu *)
 end (* if List1.Zac..*)
end; (* procedure *)

(* >>>>>>>>> Copy <<<<<<<<< *)

procedure Copy(List:TList; var Elemt:TData);
begin
 if List.Act<> nil
 then Elemt:=List.Act^.Data;

```

```
end; (* procedure *)
end.
```

## 9.2. Příklady operací nad binárními stromy

Příklady  
operací nad  
binárními  
stromy

### Ukázky některých příkladů ze stromových etud

Pozn. Příklady slouží k základní orientaci. V této podobě nejsou odladěny.

#### DUPLIKÁT BINÁRNÍHO STROMU S UKAZATELI NEREKURSÍVNĚ

- Binární strom je implementovaný pomocí ukazatelů. Napište nerekursivní zápis procedury pro vytvoření kopie (duplicátu) daného stromu. Zásobník nemusíte implementovat.

```

type
 TUk =^TUzel; (* typ uzlu *)
 TUzel=record
 Data:Tdata;
 LUK,PUk:TUk;
 end;

procedure Nejlev (UkOr:TUk; var UkKo:TUk);

(* procedura pracuje se dvěma zásobníky ukazatelů StackOr a StackKo
 prochází levou diagonálou originálu a souběžně vytváří kopii
 revise 24.6.1994 *)

var PomUk:TUk; (* lokální pomocný ukazatel pro kopii*)

begin
 if UkOr<>nil then begin (* ošetření prvního uzlu diagonály před cyklem
 *)
 new(UkKo); (* vytvoření prvního diagonály (kořene) *)
 UkKo^.Data:=UkOr^.Data;(* a jeho naplnění *)
 Push(StackOr, UkOr); (* uložení do zásobníku pro pozdější návrat *)
 Push(StackKo, UkKo);
 PomUk:=UkKo; (* umístění pomoc. uk. na "kořen" *)
 UkOr:=UkOr^.LUK; (* posun v originálu doleva *)
 PomUk^.LUK:=nil; (* nilování nejlevějšího na diagonále kopie *)

 while UkOr<>nil do begin (* průchod druhým a dalšími prvky diagonály *)
 new(PomUk^.LUK); (* vytvoření levého prvku kopie *)
 PomUk:=PomUk^.LUK; (* posun v kopii doleva *)
 PomUk^.Data:=UkOr^.Data; (* naplnění uzlu kopie *)
 Push(StackOr, UkOr); (* uk. uzlu originálu do zásobníku *)
 UkOr:=UkOr^.LUK; (* posun v originálu doleva *)
 Push(StackKo, PomUk); (* uk. uzlu kopie do zásobníku *)
 end; (* while *)

 PomUk^.LUK:=nil; (* nilování nejlevějšího na diagonále kopie *)

 end else begin (* originál prázdný, kopie taky... *)
 UkKo:=nil;
 end; (* if *)
end; (* procedure Nejlev *)

procedure TreeCopy (UkOr:TUk; var UkKo:TUk);
(* procedura vytvoří kopii - duplikát zadaného binárního stromu procedura
 pracuje se dvěma zásobníky téhož typu: StackOr a StackKo *)
var PomUk:TUk; (* pomocný ukazatel pro kopii *)
begin
```

```

SInit(StackOr);
SInit(StackKo);
Nejlev(UkOr, UkKo);

while not SEmpty(StackKo) do begin
 Top(StackOr, UkOr); Pop(StackOr);
 Top(StackKo, PomUk); Pop(StackKo);
 Nejlev(UkOr^.PUk, PomUk^.PUk);
end; (* while *)
end; (* procedure *)

```

#### DUPLIKÁT BINÁRNÍHO STROMU S INDEXY V POLI NEREKURSÍVNĚ (UDPP)

- Binární strom implementovaný (s využitím "uživatelského DPP") v poli DPP je dán indexem. Napište nerekusivní zápis procedury pro vytvoření kopie (duplicátu) daného stromu, který nebude obsahovat terminální uzly (listy) originálního stromu. Zásobník nemusíte implementovat.

```

type
 TUk=0..1000; (* hodnota 0 je užívána ve významu nil *)
 TUzel= record (* typ uzlu *)
 Data:Tdata;
 LUk, PUk:TUk;
 end;

 TDPP=array[1..1000] of TUzel; (* pole pro UDPP *)

var DPP:TDPP;

procedure Nejlev(UkOr:TUk; var UkKop:TUk);
(* procedura pracuje se dvěma zásobníky ukazatelů *)
var PomUkKop:TUk;
begin
 if (UkOr=0) or (DPP[UkOr].LUk=0) and (DPP[UkOr].PUk=0) then
 (*POZOR! funguje jen při zkráceném vyhodnocování booleovských
 výrazů. Jinak je index UkOr mimo rozsah. Nad úpravou
 se zamyslete v rámci domácí přípravy.*)
 UkKop:=0 (* kořen resp. pravý syn nilový nebo list *)
 else begin
 newDPP(UkKop); (* vytvoř kořen nebo pravého syna - první uzel diagonály *)
 DPP[UkKop].Data:=DPP[UkOr].Data; (* kopírování dat *)
 PushOr(UkOr); (* uzel v originálu do zás. *)
 UkOr:=DPP[UkOr].LUk; (* posun v originále doleva *)
 PomUkKop:=UkKop;

 while (DPP[UkOr].LUk<>0) or (DPP[UkOr].PUk<>0) do begin
 (* cyklus diagonály nezpracuje nejlevější uzel, je-li to list *)
 PushKop(PomUkKop); (* uzel kopie do zásobníku *)
 newDPP(DPP[PomUkKop].LUk); (* vytvoř neterminální řadový *)
 (* uzel diagonály kopie *)
 UkKop:=DPP[PPomUkKop].LUk; (* posun v kopii doleva *)
 DPP[PomUkKop].Data := DPP[UkOr].Data; (* kopírování dat *)
 PushOr(UkOr); (* uzel originálu do zásobníku *)
 UkOr:=DPP[UkOr].LUk; (* posun v originálu doleva *)
 end (*while*)
 PushKop(PomUkKop); (* uložení nejlevějšího uzlu diagonály kopie *)
 end

```

```

DPP [PomUkKop] . LUK := 0; (* nilování nejlevějšího uzlu diagonály kopie *)
end; (* if *)
end; (* procedure Nejlev *)

```

```

procedure RedukovanaKopie (KorOr:TUk; var KorKop:TUk);
(* procedura pracuje se dvěma zásobníky ukazatelů; každý je realizován
 samostatným souborem operací; všimněme si, že má-li každý zásobník
 "své" operace, které se liší i jménem (pak operace nemusí mít
 parametr, rozlišující o který zásobník v operaci jde; takovému
 parametru se říká "parametr instance" *)
var
 UkOr, UkKop:TUk;
begin
 SInitOr; SInitKop;
 Nejlev(KorOr, KorKop);
 while not SEmptyOr do begin
 TopOr(UkOr); TopKop(UkKop);
 PopOr; PopKop;
 Nejlev(DPP[UkOr].PUk, DPP[UkKop].PUk);
 end; (* while *)
end; (* procedure *)

```

---

**PRŮCHOD POSTORDER BINÁRNÍM STROMEM A VLOŽENÍ PRVŮ DO  
JEDNOSMĚRNÉHO SEZNAMU**

---

Při průchodu "postorder" vložte všechny uzly daného BS do jednosměrného seznamu. Pro seznam využijte výhradně operace ADT.

```

type
 TUzel=record
 LUK, PUk:TUk;
 Data:integer;
 end;

procedure Nejlev(UkTree:TUk);
begin
 while UkTree<>nil do begin
 PushUk(UkTree);
 UkTree:=UkTree^.LUK;
 PushBool(true); (* vložení příznaku "zleva" *)
 end; (* while *)
end; (* procedure *)

procedure PostOrder (UkTree:TUk; var List:TList);
var
 Zleva : Boolean;

begin
 SInitBool; (* inicializace zásobníku booleovských hodnot *)
 SInitUk; (* inicializace zásobníku ukazatelů *)

 InitList(List);

```

```

InsertFirst(List,0); (* vytvoření hlavičky *)
First(List); (* první je aktivní *)

Nejlev(UkTree);
while not SEmptyUk do begin
 TopBool(Zleva); PopBool;
 TupUk(UkTree);
 if Zleva then begin
 PushBool(false); (* vložení příznaku "příště přijde zprava" *)
 Nejlev(UkTree^.PUK);
 end else begin
 PopUk;
 PostInsert(List, UkTree^.Data); (* postupné vkládání do seznamu *)
 SuccList(List); (* postup aktivity *)
 end; (*if*)
end; (*while*)

DeleteFirst(List); (* zrušení nepotřebné hlavičky *)
end; (* procedure *)

```

Pozn.U průchodu PostOrder udává aktuální počet prvků zásobníku po každém vložení ukazatele uzlu do zásobníku vzdálenost daného uzlu od kořene. U jiných průchodů to neplatí

---

#### ALGORITMUS PRO ZRUŠENÍ BINÁRNÍHO STROMU BEZ PRŮCHODU POSTORDER.

---

```

procedure ZrusStrom(var kor:TUkUz);
(* předpokládá se rušení neprázdného stromu, tedy kor<>nil *)
begin
 InitStack(S); (* inicializace zásobníku S pro ukazatele na uzel *)

 repeat (* cyklus rušení *)
 if kor=nil
 then begin
 if not SEmpty(S)
 then begin
 kor:=TOP(S);
 POP(S);
 end;
 end else begin (* jde doleva, likviduje a pravé strká do zásobníku *)
 if kor^PUK <>nil
 then PUSH(S,Kor^.PUK);
 PomPtr:=Kor; (* uchování dočasného kořene pro pozdějsí zrušení *)
 Kor:=Kor^.LUK; (* posud doleva *)
 dispose(PomPtr); (* zrušení starého uchovaného kořene *)
 end;
 until (Kor=nil) and SEMPTY(S)

```

Obdobný algoritmus s využitím mechanismu Nejlev

```

procedure ZrusUzel (var kor:TUkUz);
(* tato verze s cyklem repeat předpokládá rušení neprázdného BS *)
var
 uz:TUkUz;

```

```

procedure nejlev (kor:TUKUz);
begin
 while kor<>nil do begin
 Push(S,kor);
 kor:=kor^.LUk;
 end;

begin;
 SInit(S);
 uz:=kor;
 repeat
 nejlev(uz); (* cestou doleva plní zásobník *)
 uz:=Top(S);Pop(S); (* uz
 if uz^.PUK <> nil
 then uz:=uz^.PUk;
 dispose(uz);
 until SEMPTY(S);
end;

```

**9.3.  
Protokol  
operace  
Insert pro  
AVL strom**

### 9.3. Protokol operace Insert pro AVL strom.

Následující ukázka je protokol programu v Pascalu EC (1983) implementující nereuirzívní zápis operace Insert pro AVL strom.  
(viz násl. strana).

## PŘÍLOHA A

PROGRAM AVLSTROM pro demonstraci nerekurzivního zápisu operace INSERT  
v AVL stromu

```

PROGRAM AVLSTROM(INPUT,OUTPUT);
(*
PROGRAM DEMONSTRUJE VYTVARENI
AVL - STROMU S VYROVNANAVANIM
NEREKURZIVNI METODOU. JEHO
CINNOST JE NAPLNI PROCEDURY
ZARAD.

DATUM: 29. 5. 1983
AUTOR: P. LAMPA, K. SOLNICKY
VUT BRNO FE, III-EP
*)

TYPE
 PUZEL="TUZEL";
 TSTAV=(LEVY_DELSI,STEJNE,PRAVY_DELSI); (* UKAZATEL NA UZEL STROMU *)
 TPODSTROM=(LEVY,PRAVY); (* TYP PODSTROMU UZLU *)
 TKOREN=ARRAY [TPODSTROM] OF PUZEL; (* UKAZATELE NA LEVY A PRAVY
 PODSTROM *)
 TPOLOZKY=RECORD
 PREDCHOZI: PUZEL; (* UKAZATEL NA PREDCHOZI UZEL *)
 STRANA: TPODSTROM; (* TYP PODSTROMU VE KTEREM JSME *)
 END;

 TZASOBNIK=ARRAY[1..16] OF TPOLOZKY; (* ZASOBNIK PRO CESTU STROMEM *)

 TUZEL=RECORD
 HODNOTA: INTEGER; (* HODNOTA UZLU *)
 KOREN: TKOREN; (* UKAZATELE NA PODSTROMY *)
 STAV: TSTAV; (* VYROVNANOST PODSTROMU *)
 END;

VAR
 HLAVICKA: PUZEL; (* VRCHOL STROMU *)
 DATA: INTEGER; (* UKLADANA HODNOTA *)
 NALEZENOK: BOOLEAN;

PROCEDURE ZARAD(DATA: INTEGER; VAR P: PUZEL; VAR NASEL: BOOLEAN);
(* TATO PROCEDURA ZARADI DO STROMU S HLAVICKOU P ZADANA DATA.
 POKUD JIZ VE STROMU TYTO DATA JSOU, NABUDE HODNOTY FALSE,
 JINAK DATA ZARADI, ZAPISE CESTU DO ZASOBNIKU A V PRIPADE
 POTREBY UPRAVI STROM NA AVL STROM. *)
VAR
 PP, (* POMOCNY UKAZATEL *)
 PREDPOSLEDNI: PUZEL; (* NEJBLIZSI VYSSI UZEL NEZ HLEDANY *)
 STRANA: TPODSTROM; (* TYP PODSTROMU NEJBLIZSIMU VYSSINU UZLU *)
 ZASOBNIK: TZASOBNIK;
 VRCHOL: INTEGER; (* VRCHOL ZASOBNIKU *)
(* NASLEDUJICI PROCEDURY SLOUZI K UPRAVE OKOLI UZLU PODSTROMU.
 NA OBRAZCICH JE ZNAZORNEN STAV PODSTROMU PRED A PO UPRAVE. *)
PROCEDURE LL_ROT(VAR P: PUZEL);
 (*
 3 2
 2 => 1 3
 1
 *)
 VAR
 P1: PUZEL;

```

```

BEGIN
 P1:=P^,KOREN[LEVY];
 P^,KOREN[LEVY]:=P1^,KOREN[PRAVY];
 P1^,KOREN[PRAVY]:=P;
 P^,STAV:=STEJNE;
 P:=P1;
END; (* LL_ROT *)

PROCEDURE RR_ROT(VAR P:PUZEL);
(* 1 2 2
 2 => 1 3
 5 *)
VAR
 P1: PUZEL;
BEGIN
 P1:=P^,KOREN[PRAVY];
 P^,KOREN[PRAVY]:=P1^,KOREN[LEVY];
 P1^,KOREN[LEVY]:=P;
 P^,STAV:=STEJNE;
 P:=P1;
END; (* RR_ROT *)

PROCEDURE LR_ROT(VAR P:PUZEL);
(* 3 2 2
 1 => 1 3
 2 *)
VAR
 P1,P2: PUZEL;
BEGIN
 P1:=P^,KOREN[LEVY];
 P2:=P1^,KOREN[PRAVY];
 P1^,KOREN[PRAVY]:=P2^,KOREN[LEVY];
 P2^,KOREN[LEVY]:=P1;
 P^,KOREN[LEVY]:=P2^,KOREN[PRAVY];
 P2^,KOREN[PRAVY]:=P;
 IF P2^.STAV = LEVY_DELSI THEN P^.STAV:=PRAVY_DELSI
 ELSE P^.STAV:=STEJNE;
 IF P2^.STAV = PRAVY_DELSI THEN P1^.STAV:=LEVY_DELSI
 ELSE P1^.STAV:=STEJNE;
 P:=P2;
END; (* LR_ROT *)

PROCEDURE RL_ROT(VAR P:PUZEL);
(* 1 2
 3 => 1 3
 2 *)
VAR
 P1,P2: PUZEL;
BEGIN
 P1:=P^,KOREN[PRAVY];
 P2:=P1^,KOREN[LEVY];
 P1^,KOREN[LEVY]:=P2^,KOREN[PRAVY];
 P2^,KOREN[PRAVY]:=P1;
 P^,KOREN[PRAVY]:=P2^,KOREN[LEVY];
 P2^,KOREN[LEVY]:=P;
 IF P2^.STAV = PRAVY_DELSI THEN P^.STAV:=LEVY_DELSI
 ELSE P^.STAV:=STEJNE;
 IF P2^.STAV = LEVY_DELSI THEN P1^.STAV:=PRAVY_DELSI
 ELSE P1^.STAV:=STEJNE;
 P:=P2;
END; (* RL_ROT *)

```

```

(* PRO UPLNE POCHOPENI TECHTO PROCEDUR JE NEJLEPE SI JEJICH
CINNOST ODSIMULOVAT NA PAPIROVEM POCITACI *)

PROCEDURE NOVY_UZEL(DATA: INTEGER; VAR P: PUZEL);
(* PROCEDURA VYTVORI NOVY UZEL A NASTAVI JEMO
POLOZKY NA PRISLUSNE HOTNOTY *)

BEGIN
 NEW(P);
 WITH P^ DO
 BEGIN
 HODNOTA:=DATA;
 KOREN[LEVY]:=NIL;
 KOREN[PRAVY]:=NIL;
 STAV:=STEJNE;
 END;
END; (* NOVY_UZEL *)

PROCEDURE VYBALANCOVANI;
(* PROCEDURA PROVEDE VYBALANCOVANI STROMU *)

VAR
 BALANCOVAT: BOOLEAN;
 P: PUZEL;

BEGIN
 BALANCOVAT:=TRUE; (* BUDEM BALANCOVAT *)
 WHILE BALANCOVAT AND (VRCHOL > 1) DO
 BEGIN
 P:=ZASOBNIK[VRCHOL].PREDCHOZI;
 IF ZASOBNIK[VRCHOL].STRANA=LEVY THEN (* PRIDALI JSME
 LEVY UZEL *)
 CASE P^.STAV OF
 PRAVY_DELSI:
 (* UZEL MEL PRAVY PODSTROM
 A TAK SE SROVNAL *)
 BEGIN
 P^.STAV:=STEJNE;
 BALANCOVAT:=FALSE;
 END;
 STEJNE:
 (* UZEL BYL KONCOVY
 A JESTE TO UJDE *)
 P^.STAV:=LEVY_DELSI;
 LEVY_DELSI:
 (* UZEL JIZ MEL LEVY PODSTROM
 A TAK JEJ MUSIME SROVNAT *)
 BEGIN
 ZASOBNIK[VRCHOL-1].PREDCHOZI:=P;
 IF P^.KOREN[LEVY]^ .STAV=LEVY_DELSI THEN
 LL_ROT(PREDCHOZI^.KOREN[STRANA]);
 ELSE
 LR_ROT(PREDCHOZI^.KOREN[STRANA]);
 P^.STAV:=STEJNE;
 BALANCOVAT:=FALSE;
 END;
 END; (* CASE *)
 END;
END;

```

```

ELSE (* PRIDALI JSME PRAVY UZEL *)
CASE P^.STAV OF
 LEVY_DELSI:
 (* UZEL MEL LEVY PODSTROM
 A TAK SE SROVNAL *)
 BEGIN
 P^.STAV:=STEJNE;
 BALANCOVAT:=FALSE;
 END;
 STEJNE:
 (* UZEL BYL KONCOVY
 A JESTE TO UJDE *)
 P^.STAV:=PRAVY_DELSI;
 PRAVY_DELSI:
 (* UZEL JIZ MEL PRAVY PODSTROM
 A TAK JEJ MUSIME SROVNAT *)
 WITH ZASOBNIK[VRCHOL=1] DO
 BEGIN
 IF P^.KOREN[PRAVY]^,STAV=PRAVY_DELSI THEN
 RR_ROT(PREDCHOZI^,KOREN[STRANA]);
 ELSE
 RL_ROT(PREDCHOZI^,KOREN[STRANA]);
 P^.STAV:=STEJNE;
 BALANCOVAT:=FALSE;
 END;
 END; (* CASE *)
 VRCHOL:=VRCHOL+1;
 END; (* WHILE *)
END; (* VYBALANCOVANI *)

BEGIN
 (* INICIALIZACNI PRIKAZY *)
 VRCHOL:=0;
 STRANA:=LEVY;
 PREDPOSLEDNI:=P;
 PP:=P^.KOREN[LEVY];
 IF PP=NIL THEN NOVY_UZEL(DATA,P^.KOREN[LEVY]) (* ZALOZENI STROMU *)
 ELSE
 (* ZARAZENI DO STROMU *)
 BEGIN
 NASEL:=FALSE;
 WHILE (PP<>NIL) AND NOT NASEL DO
 BEGIN
 (* ZAPISEME CESTU DO ZASOBNIKU *)
 VRCHOL:=VRCHOL+1;
 ZASOBNIK[VRCHOL].STRANA:=STRANA;
 ZASOBNIK[VRCHOL].PREDCHOZI:=PREDPOSLEDNI;
 PREDPOSLEDNI:=PP;

 (* ROZHODNEME SE KUDY DALE PUJDEM *)
 IF PP^.HODNOTA=DATA THEN NASEL:=TRUE
 ELSE
 IF DATA<PP^.HODNOTA THEN
 BEGIN
 STRANA:=LEVY;
 PP:=PP^.KOREN[LEVY];
 END;
 END;
 END;

```

```

 ELSE
 BEGIN
 STRANA:=PRAVY;
 PP:=PP^.KOREN[PRAVY];
 END;
 END; (* WHILE *)
IF NOT NASEL THEN
BEGIN
 NOVY_UZEL(DATA,PP); (* VYTVORENI NOVEHO UZLU *)
 PREDPOSLEDNI^.KOREN[STRANA]:=PP; (* NAVAZANI DO STROMU *)
 VRCHOL:=VRCHOL+1;
 ZASOBNIK[VRCHOL].STRANA:=STRANA;
 ZASOBNIK[VRCHOL].PREDCHOZI:=PREDPOSLEDNI;
 VYBALANCOVANI;
END;
END; (* IF *)
ZARAD:=NASEL;
END; (* ZAHAD *)

BEGIN
(* HLAVNI PROGRAM *)
NOVY_UZEL(MAXINT,MLAVICKA);
WHILE NOT EOF(INPUT) DO
BEGIN
 READLN(DATA);
 ZARAD(DATA,MLAVICKA,NALEZENO);
END;
END.

```

**9.4. Ukázka studentského projektu - operace Insert v AVL stromu v hazyce C.**

Autor: "Autor: Peter Javorsky, [xjavor07@stud.fit.vutbr.cz](mailto:xjavor07@stud.fit.vutbr.cz)", zmní semestr IAL, 2005.

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct AVL_Tree_struct {
 struct AVL_Tree_struct *left, *right;
 int data;
 int bal; // balancia
} AVL_Tree;

AVL_Tree terminator;
// zarazka

void *test_alloc(void *x) //testuje, ci pri alokaci pamate nedoslo miesto
{
 if(x == NULL) {
 fprintf(stderr, "Chyba - nedostatok pamate\n");
 exit(-1);
 }
 return x;
}

void RotLL(AVL_Tree *parent, AVL_Tree *node, AVL_Tree **my_tree)
{

```

```

AVL_Tree *A = node, *B = node->right;
A->right = B->left;
B->left = A;

if(parent == NULL) {
 *my_tree = B;
} else {
 if(parent->right == node)
 parent->right = B;
 else
 parent->left = B;
}

A->bal = 0;
B->bal = 0;
}

void RotRR(AVL_Tree *parent, AVL_Tree *node, AVL_Tree **my_tree)
{
 AVL_Tree *A = node, *B = node->left;

 A->left = B->right;
 B->right = A;

 if(parent == NULL) {
 *my_tree = B;
 } else {
 if(parent->right == node)
 parent->right = B;
 else
 parent->left = B;
 }

 A->bal = 0;
 B->bal = 0;
}

void RotDLR(AVL_Tree *parent, AVL_Tree *node, AVL_Tree **my_tree)
{
 AVL_Tree *A = node, *B = node->right, *C = node->right->left;

 if(A == &terminator || B == &terminator || C == &terminator)
 return;

 B->left = C->right;
 A->right = C->left;
 C->right = B;
 C->left = A;

 if(parent == NULL) {
 *my_tree = C;
 } else {
 if(parent->right == node)
 parent->right = C;
 else
 parent->left = C;
 }

 if(C->bal == 1) {
 B->bal = -1;
 A->bal = 0;
 } else {
 B->bal = 0;
 A->bal = 1;
 }
 C->bal = 0;
}

void RotDRL(AVL_Tree *parent, AVL_Tree *node, AVL_Tree **my_tree)
{
 AVL_Tree *A = node, *B = node->left, *C = node->left->right;

 if(A == &terminator || B == &terminator || C == &terminator)
 return;

 B->right = C->left;
}

```

```

A->left = C->right;
C->left = B;
C->right = A;

if(parent == NULL) {
 *my_tree = C;
} else {
 if(parent->right == node)
 parent->right = C;
 else
 parent->left = C;
}

if(C->bal == 1) {
 B->bal = 1;
 A->bal = 0;
} else {
 B->bal = 0;
 A->bal = -1;
}
C->bal = 0;
}

int *AVL_Insert(int data, AVL_Tree **my_tree)
{
 AVL_Tree *node = *my_tree, *critical = NULL, *critical_parent = NULL, *node_parent =
NULL;
 int *ins = NULL;

 if((*my_tree) == &terminator) {
 (*my_tree) = (AVL_Tree*)test_alloc(malloc(sizeof(AVL_Tree)));
 (*my_tree)->left = &terminator;
 (*my_tree)->right = &terminator;
 (*my_tree)->data = data;
 (*my_tree)->bal = 0; // balanced
 // vytvoril novu vetvu ...

 return &(*my_tree)->data;
 }
 // pri prazdnom zozname nic nerobime

 while(1) {
 if(node->data == data) {
 return &node->data; // nasiel
 } else if(node->data > data) {
 if(node->left == &terminator)
 break;

 if(node->bal != 0) {
 critical = node;
 critical_parent = node_parent;
 }
 // potencialna kriticka noda

 node_parent = node;
 node = node->left;
 // chod dolava
 } else {
 if(node->right == &terminator)
 break;

 if(node->bal != 0) {
 critical = node;
 critical_parent = node_parent;
 }
 // potencialna kriticka noda

 node_parent = node;
 node = node->right;
 // chod doprava
 }
 }
 // hľada uzol

 if(critical != NULL)
 node = critical;
 while(1) {

```

```

 if(data < node->data) {
 node->bal++;
 if(node->left == &terminator) {
 ins = AVL_Insert(data, &node->left);
 // vlozi novy vyvazeny koren
 break;
 } else
 node = node->left;
 } else {
 node->bal--;
 if(node->right == &terminator) {
 ins = AVL_Insert(data, &node->right);
 // vlozi novy vyvazeny koren
 break;
 } else
 node = node->right;
 }
 }
 // vlozi uzol

 if(critical == NULL)
 return ins;
 if(critical->bal == -2) {
 if(critical->left->bal == -1)
 RotLL(critical_parent, critical, my_tree);
 else
 RotDLR(critical_parent, critical, my_tree);
 } else if(critical->bal == 2) {
 if(critical->right->bal == 1)
 RotRR(critical_parent, critical, my_tree);
 else
 RotDRL(critical_parent, critical, my_tree);
 }
 // ak sme rozvazili strom, rotujeme vetvy

 return ins;
}

void AVL_Erase(AVL_Tree *tree)
{
 if(tree->left != &terminator)
 AVL_Erase(tree->left);
 if(tree->right != &terminator)
 AVL_Erase(tree->right);

 printf("Rekurzivne odstraneny prvok %d...\n", tree->data);
 free(tree);
 // rekurzivne uvolni strom (post-order)
}

int Tree_Height(AVL_Tree *tree) // rekurzivna vyska stromu
{
 if(tree == &terminator)
 return 0;

 int lh = Tree_Height(tree->left);
 int rh = Tree_Height(tree->right);

 if(lh > rh)
 return lh + 1;
 else
 return rh + 1;
}
/*
 1 medzera 16-1, pozicia dalsi=4, 12
 2 3 medzera 8-1, pozicia dalsi=2,
 4 5 6 7 medzera 4-1, pozicia dalsi=1
 8 9 0 1 2 3 4 5 medzera 2-1, pozicia dalsi=0

 10 medzera 16-1, pozicia dalsi=4, 12
 20 medzera 8-1, pozicia dalsi=2,
 30 medzera 4-1, pozicia dalsi=1
 40 50 60 70 medzera 2-1, pozicia dalsi=0
 80 90 00 10 20 30 40 50

```

```

vyska stromu=4
počet detí smerom dole s medzerami=15 = 2^4-1
pozícia prvej vetvy=16/2=8
pozícia dalsia=8

*/
char space_char;

void PrintTree_r(AVL_Tree *tree, int len, int height, int pos, int medzera, int isright, int
isrightmost)
{
 if(!height)
 return;

 if(height == 1) {
 if(!isright) { // ak není právý (pravé uz suu odsadene)
 for(int i = len; i < pos; i++)
 putchar(' ');
 }
 // odsadi ...
 if(tree == &terminator) {
 for(int i = 0; i < len; i++)
 putchar(space_char);
 } else {
 char fmt[20];
 sprintf(fmt, "%0%dd", len);
 printf(fmt, tree->data);
 }
 // vypíše obsah stromu ...

 if(isrightmost)
 putchar('\n');
 // ak je najpravší, odriadkuje
 } else {
 if(tree != &terminator) {
 PrintTree_r(tree->left, len, height - 1, pos - medzera / 2, medzera / 2,
isright, false);

 if(height == 2) {
 for(int i = len; i < medzera; i++)
 putchar(' ');
 // odsadi ...
 }

 PrintTree_r(tree->right, len, height - 1, pos + medzera / 2, medzera /
2, true, isrightmost);

 if(height == 2 && !isrightmost) {
 for(int i = len; i < medzera; i++)
 putchar(' ');
 // odsadi ...
 }
 } else if(height > 0) { // pre učely správneho zarovnávania musíme tláciť i
casti stromu ktoré už nici neobsahujú
 PrintTree_r(&terminator, len, height - 1, pos - medzera / 2, medzera /
2, isright, false);

 if(height == 2) {
 for(int i = len; i < medzera; i++)
 putchar(' ');
 // odsadi ...
 }

 PrintTree_r(&terminator, len, height - 1, pos + medzera / 2, medzera /
2, true, isrightmost);

 if(height == 2 && !isrightmost) {
 for(int i = len; i < medzera; i++)
 putchar(' ');
 // odsadi ...
 }
 }
 }
}

```

```

 }

 }

int pow2(int x) // mocnina dvoch
{
 int y = 1;
 for(;x > 0; x --, y *= 2)
 ;
 return y;
}

/*
 1
 x 7
 x x x 4

*/
void PrintTree(AVL_Tree *tree)
{
 int vyska = Tree_Height(tree);
 int mocnina2na_vyska = pow2(vyska);
 int pos_1 = mocnina2na_vyska / 2;
 int medzera = mocnina2na_vyska;

 for(int i = 1; i <= vyska; i++)
 PrintTree_r(tree, 2, i, pos_1 * 2, medzera, false, true);
 // vypise strom ...
}

int main()
{
 char ch;
 space_char = '.';
 printf("NAHRADNY PROJEKT Z IAL\n"
 "Tento program zobrazuje rotacie AVL stromu pri postupnom\n"
 "vkladani 10tich nahodnych novych prvkov. Ak sa vkladany prvak\n"
 "v strome uz nachadza, nic sa nedeje.\n"
 "Autor: Peter Javorosky, xjavor07@n\n"
);
 system("pause");
 printf("\n");

 do {
 ch = '\0';
 AVL_Tree *tree = &terminator;
 for(int i = 0; i < 10; i++) {
 int x = rand() % 20;
 printf("Pridany prvak %d...\n", x);
 AVL_Insert(x, &tree);

 printf("-----\n");
 PrintTree(tree);
 printf("-----\n\n");
 system("pause");
 system("cls");
 }
 // vlozi 15 nahodnych poloziek ...

 system("cls");
 AVL_Erase(tree);
 printf("\nFrajete si vytvorit iny strom? (A/N):");

 }while ((ch=getchar()) && (ch == 'a' || ch == 'A') && printf("Ano.\n"));

 printf(" Nie\n");
 system("pause");
 // vymaze cely strom rekurzivne
 }
 return 0;
}

```

