VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

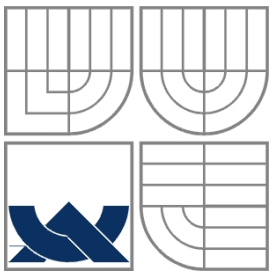
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ALGORITMY ŘAZENÍ V JAZYCE C

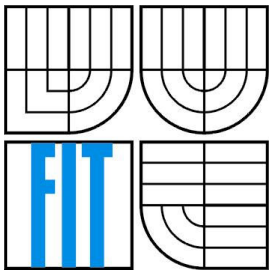
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE JIŘÍ VANĚK
AUTHOR

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ALGORTIMY ŘAZENÍ V JAZYCE C

SORTING ALGORITHMS IN C LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE — Jiří Vaněk
AUTHOR

VEDOUCÍ PRÁCE — Prof. Ing. Jan Maxmilián Honzík, CSc.
SUPERVISOR

BRNO 2007

Zadání bakalářské práce/4999/2006/xvanek26

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2006/2007

Zadání bakalářské práce

Řešitel: **Vaněk Jiří**
Obor: Informační technologie
Téma: **Algoritmy řazení v jazyce C**
Kategorie: Alg. a datové struktury

Pokyny:

1. Seznamte se detailně s textem kapitoly 5 o řazení ve studijní opoře pro předmět IAL a s metodami řazení v této kategorii v dostupné literatuře
2. Modifikujte text kapitoly zapsané pro pascalovský jazyk tak, aby zachoval co nejvíce (i v detailech) původní obsah, ale aby se vztahoval k zápisu příkladů a algoritmů v jazyce C
3. Převeďte všechny příklady a algoritmy do jazyka C a odlaďte je v prostředí vytrvořeném pro tento účel
4. Navrhněte demonstrační animační program pro vybrané algoritmy s využitím grafického rozhraní (ListMerge-sort, Merge-sort, Radix-sort).
5. Navrhněte a vytvořte další vhodné kontrolní otázky a příklady.
6. Navrhněte příklady vhodné pro formulářově orientovanou písemnou zkoušku ze znalostí tohoto okruhu.

Literatura:

- Honzík,J.M.: Aloritmy. Studijní opora předmětu Algoritmy. FIT VUT v Brně

Při obhajobě semestrální části projektu je požadováno:

1. Přepsaní a odladěné algoritmy metod řazení uvedených ve studijní opoře.
2. Demonstrační program s animací principů vybraných řadicích metod.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Honzík Jan M., prof. Ing., CSc.**, UIFS FIT VUT
Datum zadání: 1. listopadu 2006
Datum odevzdání: 15. května 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Jaroslav Zendulka, CSc.
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Jiří Vaněk**
Id studenta: 84116
Bytem: Dolní Pálava 289/46, 678 01 Blansko
Narozen: 16. 03. 1985, Brno
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
bakalářská práce

Název VŠKP: Algoritmy řazení v jazyce C
Vedoucí/školitel VŠKP: Honzík Jan M., prof. Ing., CSc.
Ústav: Ústav informačních systémů
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:
tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (1 ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☐ ihned po uzavření této smlouvy
   - ☒ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

....

Nabyvatel Autor

Abstrakt

Tato práce se zabývá řadicími algoritmy z pohledu jejich studia. Jejím účelem je poskytnout studentům materiály, které jim mohou pomoci v pochopení těchto algoritmů. Práce se skládá z přepsání části studijní opory předmětu Algoritmy do jazyka C, vytvoření programu pro testování algoritmů z opory a vytvoření animace demonstrující činnost vybraných řadicích algoritmů.

Klíčová slova

Řadicí algoritmy, Bubble sort, Merge sort, ListMerge sort, Radix sort, Quick sort

Abstract

This thesis deals with the sorting algorithms in the context of studying them. The purpose is to provide the students with materials, which can help them with the understanding of these algorithms. The work consists of the rewriting of the study-supporting materials into the C language, the creation of a test program for the appropriate algorithms and the creation of an animation to demonstrate the functionality of the selected algorithms.

Keywords

Sorting algorithms, Bubblesort, Mergesort, ListMergesort, Radixsort, Quicksort

Citace

Vaněk Jiří: Algoritmy řazení v jazyce C, bakalářská práce, Brno, FIT VUT v Brně, 2007

Algoritmy řazení v jazyce C

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Prof. Ing. Jana Maxmiliána Honzíka, CSc.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jméno Příjmení
30.7.2007

Poděkování

Děkuji vedoucímu své bakalářské práce, panu Prof. Ing. Janu Maxmiliánu Honzíkovi, CSc., za cenné rady a ochotu vždy pomoci nejen s touto prací.

© Jiří Vaněk, 2007.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů..

Obsah

Úvod

Tato práce si klade za cíl vytvořit nové studijní materiály pro předmět Algoritmy, vyučovaný v bakalářském studijním programu „Informační technologie“ na FIT VUT v Brně. V prvních letech existence této fakulty (2002-2004) zde byl jako výchozí jazyk pro výuku programování a algoritmizace používán Pascal, dříve nejběžnější studentský jazyk. Od roku 2004 fakulta přechází od Pascalu k dnes více použitelnému jazyku C a většina předmětů, u kterých je to možné, se orientuje právě na tento jazyk. U předmětu „IAL - Algoritmy“ k této výměně jazyků nedošlo. „Tento předmět používá algoritmický 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“ [1]. IAL se tak stal jediným předmětem, který studenty seznamuje s jazykem pascalovského typu, což může být pro některé studenty nepříjemné, pokud nemají s takovým jazykem dřívější zkušenost. Přicházela kritika ze strany studentů, a reakcí na ni byl návrh na vytvoření nové studijní opory, která by zachovala obsah původní opory, ale byla by psána ve studentům známém jazyce C.

Studijní opora byla tématicky rozdělena na čtyři části, mojí prací je vytvořit část zabývající se řadicími algoritmy. Doplňkem této části studijní opory bude demonstrační animační program pro vybrané algoritmy (ListMerge sort, Merge sort, Radix sort). Po spojení všech částí tak studenti dostanou k dispozici novou studijní oporu, která jim umožní studovat předkládané algoritmy paralelně v Pascalu i v C. Společně s animacemi ke každému tématu tak vznikne kolekce praktických studijních materiálů, které usnadní pochopení studovaných algoritmů.

V první kapitole je vysvětlen smysl této práce z pohledu aktuálnosti jazyka C a významu probíraných algoritmů. Druhá kapitola představuje podrobněji dílčí části mojí práce: novou studijní oporu, program pro ladění a testování algoritmů z této opory a animaci demonstrující vybrané algoritmy. Jsou zde definovány požadavky na zpracování jednotlivých částí. Třetí kapitola obsahuje postup práce, popisuje jednotlivé etapy tvorby výsledného díla. Ve čtvrté kapitole se zabývám teorií potřebnou k pochopení algoritmů uvedených v animaci, pak rozebírám tyto samotné algoritmy a vysvětluji jejich použití v animaci. Pátá kapitola popisuje implementaci animace, následuje závěr.

1 Smysl této práce

Jak bylo řečeno v úvodu, cílem této práce je vytvořit studijní materiály v jazyce C. Nabízí se však otázka: Není jazyk C, stejně jako Pascal, také již zastaralý? Na to odpovídá první část této kapitoly.

1.1 Význam jazyka C

V dnešní době se jazyk C může jevit jako ne příliš aktuální. Většina softwarových projektů je dnes psána v objektově orientovaných jazycích, které nabízejí modernější a účinnější nástroje pro vývoj aplikací splňujících dnešní požadavky. V komerční oblasti jsou dnes nejrozšířenější jazyky jako Java, C++ a C#, které nejsou čistě objektově orientované, ale poskytují všechny potřebné prostředky pro objektový návrh programu a přitom zachovávají některé výhody imperativních jazyků. S dnešním prudkým rozvojem internetu je spojen vzestup popularity a významu webových jazyků jako je například PHP, které ve své poslední specifikaci také podporuje objektovou orientaci. Pro většinu programátorů tak bude v dnešní době pravděpodobně nutností znalost zmíněných, nebo podobných moderních jazyků.

Mohlo by se tedy zdát logické, programování na školách od začátku vyučovat například v Javě, která se dnes pravidelně umísťuje na prvních pozicích statistik popularity i rozšířenosti programovacích jazyků. Mnohé školy, kombinující informatiku s jiným oborem, skutečně vyučují základy programování právě na Javě. Takový přístup je pochopitelný, pokud má škola širší zaměření než jen informatiku. Pro lidi, kteří se však chtějí v budoucnu věnovat především programování, představuje jazyk C velmi dobrou pomůcku k zlepšení jejich programátorských schopností. Z dnešního pohledu je totiž jazyk C poměrně blízký strojovému jazyku. Programátor se musí sám postarat o alokaci a uvolnění paměti, musí dobře zvládat práci s ukazateli, a abstraktní datové typy (např. seznamy, fronty) si musí vytvořit a spravovat sám. V moderních programovacích jazycích bývají tyto záležitosti před uživatelem skryty, což umožňuje věnovat se více samotné funkcionalitě vytvářeného programu, avšak opravdové efektivity programování v takovém jazyce je možné dosáhnout tehdy, když programátor chápe principy skryté za moderními nástroji, které jazyk nabízí. Znalost jazyka C usnadní pochopení procesů jako je vznik nového objektu, předávání reference na objekt, použití tříd popisujících abstraktní datové typy a mnoho dalších. Detailní pochopení a schopnost vlastní implementace těchto mechanismů už dnes zřejmě není nezbytnou výbavou každého programátora, přesto však přináší jasné výhody pro každého, kdo chce v programování dosáhnout vysoké kvality. A to by mělo být cílem každého studenta informatiky. Proto je jazyk C vhodný a velmi užitečný pro výukové účely.

Další argument hovořící pro jazyk C je zcela zřejmý: syntaxe všech zmíněných jazyků (Java, C++, C#, PHP) vychází právě z C, takže student začínající na C nebude mít větší problém zvyknout si na syntaxi některého z těchto jazyků, pokud se na něj zaměří.

1.2 Význam řadicích algoritmů

1.2.1 Algoritmy obecně

Algoritmy patří k programování zcela neoddělitelně, úlohou programátora je vždy nalézt vhodný algoritmus, a zapsat jej ve vhodném jazyce. Samotný algoritmus je na jazyce nezávislý, jedná se pouze o postup definující kroky k dosažení jistého cíle. Psaní programů v kterémkoli jazyce je tedy v jádru jen zapisování algoritmů, které zůstávají pro všechny jazyky stejné. Některá programovací paradigmata (např. logické) sice vyžadují zcela jiný přístup, ale u většiny běžně používaných jazyků se nic zásadního nemění. Každý začínající programátor se společně se svým prvním programovacím jazykem učí i základy algoritmizace. Ty bývají na počátku snadné a zdají se velmi přirozené, proto není zřejmá potřeba podrobnějšího studia algoritmů. S narůstající obtížností řešených úloh ale přestává být řešení na první pohled zřejmé a bývá často nutné použití složitějších postupů. U nich se již vyplatí prostudovat, jaká řešení byla u podobných problémů dříve nalezena a použita. Studium algoritmů podává přehled užitečných metod, často známých desítky let, a přitom stále velmi aktuálních a používaných. Zde už nelze namítnout, že moderní programovací jazyky nabízejí mocné nástroje, které automaticky řeší řadu složitých problémů. Bez schopnosti algoritmického myšlení se žádný programátor neobejde, protože při psaní v každém jazyce nastane dřív či později situace vyžadující nalezení složitého algoritmu a obecné studium algoritmizace má v takovém případě významnou roli.

1.2.2 Algoritmy řazení

1.2.2.1 Řazení a třídění

Než začneme mluvit o řadicích algoritmech, je nutné jasně definovat dva lehce zaměnitelné, avšak významem rozdílné pojmy:

„Třídění je rozdělování položek homogenní datové struktury do skupin (tříd) se (zadanými) shodnými vlastnostmi (atributy).“

„Řazení je uspořádání položek dané lineární homogenní datové struktury do sekvence podle relace uspořádání nad zadanou vlastností (klíčem) položek.“

Definice převzaty ze studijní opory [1], kde jsou tyto pojmy blíže vysvětleny.

1.2.2.2 Historie řadicích metod

Úloha automatizovaného seřazení či roztřídění souboru dat byla řešena daleko před vznikem prvních elektronických počítačů. V roce 1889 sestrojil Herman Hollerith mechanický třídící stroj, založený na děrných štítcích. Díry v štítku představovaly číslice, podle vyděrovaných čísel se otevíraly příslušné zásobníky, do kterých štítky padaly a tím byly tříděny. Stroj významně urychlil vyhodnocování statistických dat, například při sčítání lidu v USA v roce 1890.

Stejného principu třídění se na podobných mechanických strojích využívalo i k řazení. Proto se pro řazení vžil anglický výraz „sorting“, a přestože většina počítačových řadicích algoritmů princip třídění nevyužívá, anglický termín sorting jim zůstal.

Informace této podkapitoly čerpány ze studijní opory [1] a z internetu [2].

1.2.2.3 Význam řadicích algoritmů dnes

Důležitost algoritmů řazení je zřejmá už z faktu, že řazení bylo jednou z prvních úloh, kterou řešily první počítače i mechanické stroje před nimi. Dnes jsou tyto metody neméně významné, prakticky každá dnešní aplikace potřebuje někdy seřadit určitá data. Zvláště velký význam má použitá řadicí metoda ve velkých databázových systémech, kde objem řazených dat bývá značně velký a maximální možná rychlost řadicí metody je proto nezbytná. S dnešním rozvojem internetu vzrostl význam vyhledávacích metod, a vyhledávání se mnohonásobně zefektivní, pokud se vyhledává v již seřazených datech. V oblasti internetu se tedy nachází další významné uplatnění řadicích algoritmů.

Rozumět řadicím algoritmům je důležité pro každého programátora nejen proto, aby dokázal řazení dat ve svých programech implementovat s dostatečnou efektivitou, ale také aby se seznámil se zajímavými a praktickými programovacími principy, které jsou použitelné i v jiných algoritmech. Jedná se například o práci s více zanořenými cykly, která je pro řazení typická, nebo o rekurzi.

2 Dílčí části této práce

2.1 Studijní opora

Hlavním cílem této práce je přepsání studijní opory předmětu Algoritmy do jazyka C. Tento úkol byl první a základní myšlenkou celé práce. Vytvoření nové opory vyžaduje modifikaci původního textu, tak aby zachoval co nejvíce původní obsah, ale aby se vztahoval k uvedeným algoritmům zapsaným v C. V této části opory se neustále pracuje s poli čísel a pro tuto práci je důležité, jakým způsobem se tato pole indexují. Zatímco v Pascalu je možné dolní index pole nastavit na libovolnou hodnotu, a většinou se tedy volí začátek pole na indexu jedna, v jazyce C začínají indexy vždy od nuly. To je jedna z odlišností, na kterou bylo nutné vzít ohled při modifikaci textu opory. Dále bylo nutné v opoře upravit například texty týkající se definování vlastních datových typů, práci s řetězci, nebo předávání a návrat parametrů funkcí. Tyto a další odlišnosti mezi jazyky C a Pascal musely být v novém textu zohledněny.

Studijní opora předkládá studentům mnoho konkrétních příkladů, krátkých úseků kódu, ukazujících možnou implementaci uváděných algoritmů. Tyto příklady musely být přepsány do jazyka C, odladěny a testovány tak, aby všechen kód uvedený v opoře byl ozkoušený a funkční.

Během práce na nové opoře se také očekává odstranění chyb a nedostatků ze stávající opory.

2.2 Program pro ladění a testování algoritmů z opory

Při přepisu jednotlivých algoritmů z opory jsem potřeboval program, ve kterém bych mohl tyto algoritmy ladit a testovat. Algoritmy v opoře jsou jen nekompletními úseky kódu, bylo nutné vytvořit program, do kterého by se daly lehce vkládat za účelem jejich testování. Vzhledem k tomu, že všechny tyto řadicí algoritmy mají stejný typ vstupních a výstupních dat, nebylo složité program pro tento účel navrhnout. Jako vstup očekává řadicí algoritmus pole čísel, v praxi samozřejmě budeme řadit spíše větší položky, které budou kromě klíčů k řazení obsahovat i nějaká užitečná data. Pro výukové účely nám postačí pouze číselné položky. Různé algoritmy s těmito vstupními daty nakládají různým způsobem, ale výstup je opět jednotný: pole se stejnými položkami, tentokrát však seřazenými.

Je tedy potřeba mít program, který připraví vstupní pole, zavolá zvolenou řadicí metodu s tímto vstupem a uloží výstup. Některé algoritmy potřebují ke své činnosti určité datové typy, které jazyk C standardně neposkytuje. Proto musí tento program také připravit k použití typy jako

zásobník nebo lineární seznam. Program musí být co nejjednodušší, nepotřebuje grafické rozhraní ani jiné efekty, které by odváděly pozornost od samotných algoritmů. Navrhnul jsem tedy jednoduchou konzolovou aplikaci, v jejímž kódu se dá snadno orientovat a dělat změny, přidávat nové algoritmy či ladicí informace. Mně tento program posloužil k odladění všech řadicích algoritmů z opory, stejně dobře může sloužit budoucím studentům k experimentům s těmito algoritmy, přidávání nových algoritmů, nebo jen k uvedení úseků kódu z opory do souvislosti a ověření jejich funkčnosti.

Úseky kódu, které neobsahují přímo řadicí metody, ale jiné algoritmy z tohoto oboru (například MacLarenův algoritmus), nemají stejný formát vstupů a výstupů, a proto nebyly do tohoto testovacího programu zahrnuty. Byly však odladěny v samostatných programech, žádný kód nebyl přidán do studijní opory bez otestování jeho funkčnosti.

Zdrojový kód programu by měl být studentům, spolu se studijní oporou, poskytnut ke studiu. Ovládání programu je zpřístupněno pouze při spuštění programu přes parametry příkazového řádku, což umožňuje snadné použití programu v testovacích skriptech.

2.3 Demonstrační animační program

Další praktickou studijní pomůckou může být animační program, který by jednoduchým a snadno pochopitelným způsobem demonstroval práci některých zajímavých algoritmů. Úkolem této práce není srovnávat rychlost a jiné charakteristiky jednotlivých metod, ale pomoci k pochopení jejich principů. Proto ani animace nemá sloužit k nějakému porovnávání, ale má co nejlépe vystihovat podstatu dané metody. Ovládání má být co nejjednodušší a intuitivní.

Souběžně s touto prací vzniká v letošním roce jiná bakalářská práce, celá věnovaná programu pro animovanou demonstraci řadicích algoritmů. Animace, která vznikne, má být také poskytnuta jako studijní materiál k předmětu Algoritmy, proto nemusí moje animace demonstrovat většinu základních řadicích algoritmů, které jsou předmětem zmíněné cizí práce. Místo toho se můžu zaměřit na podrobnější zobrazení některých zajímavých, a méně známých metod. Moje zadání mi předepisuje vytvořit animaci pro ListMerge sort, Merge sort a Radix sort. Já jsem do animace přidal také Bubble sort a Quick sort, důvody tohoto rozšíření jsou popsány v kapitole 4 u těchto konkrétních algoritmů.

Způsobem demonstrace se snažím v animaci co nejvíc přiblížit výkladu daného algoritmu ve studijní opoře. U jednodušších metod, které mají v opoře uveden kompletní kód, zobrazuje animace řazené pole ve své skutečné podobě a se všemi indexy i jinými prostředky, které metoda využívá. U složitějších metod, popisovaných v opoře pouze pseudokódem, naznačuje animace pouze abstraktně, jaké operace metoda provádí.

Zdrojový kód tohoto animačního programu není vhodné předkládat studentům ke studiu, protože většinou neobsahuje skutečné implementace řadicích algoritmů, ale pouze jistou interpretaci práce těchto metod.

3 Postup práce

3.1 Studijní opora a testovací program

Prvním krokem a začátkem celé této práce bylo prostudování té části studijní opory, kterou jsem měl zpracovávat. Již při prvním pročítání jsem začal postupně přepisovat uváděné příklady do jazyka C. Tak jsem se seznamoval s textem opory a přepisování jednotlivých algoritmů mi pomohlo tyto metody detailněji pochopit. Jak přibývalo hotových algoritmů, objevila se potřeba jednotného testovacího prostředí, a tak vznikla první verze programu pro ladění a testování. První etapou vzniku tohoto celku bylo tedy přepsání příkladů z opory a jejich odladění v programu vytvořeném pro tento účel.

Druhou etapou byl opětovný průchod celou studijní oporou, přičemž jednotlivé algoritmy byly již přepsané do jazyka C, a tak mohl být modifikován i text vztahující se k nim. V této chvíli bylo také možné přepsaný kód zrevidovat a upravit, aby vstupy a výstupy byly opravdu jednotné. Na konci této etapy byl text nové opory hotový, a testovací program byl dokončen a uveden do podoby, ve které může být předložen studentům k jejich vlastním experimentům.

3.2 Animace

3.2.1 Prototyp

První verze animace vznikla v zimním semestru třetího ročníku v rámci projektu do předmětu „Tvorba uživatelských rozhraní“. Tato verze si kladla za cíl navrhnout jednoduché a snadno použitelné uživatelské rozhraní. Druhým důležitým cílem bylo vytvořit systém zobrazování detailního průběhu řazení, který by byl univerzálně použitelný pro metody odlišných principů. V této fázi se tolik nejednalo o samotné řadicí algoritmy, pouze některé z požadovaných algoritmů byly implementovány, a to často v silně zjednodušených verzích.

3.2.2 Výběr prostředí

Důležitá byla v této fázi volba jazyka a nástroje pro realizaci animace. Volil jsem mezi jazykem C/C++ s nějakou grafickou knihovnou či toolkitem, a Javou.

Jazyk C není pro tvorbu GUI aplikací navržený a jeho použití k tomuto účelu je díky různým knihovnám možné, ale poněkud náročné. Nabízelo by se například prostředí WIN API, ale tuto možnost jsem zamítnul pro značnou a zbytečnou složitost.

Samotný jazyk C++ stejně jako C neposkytuje prostředky pro tvorbu grafických uživatelských rozhraní, ale jeho použití pro tento účel už je mnohem běžnější, zejména díky platformě .NET v prostředí Microsoft Visual Studio. Návrh GUI aplikace je zde velmi usnadněn možností přímého grafického návrhu, kdy může programátor „skládat“ uživatelské rozhraní své aplikace z připravených komponent jako jsou například menu a tlačítka a přitom má kontrolu nad generovaným kódem.

Java poskytuje už v definici jazyka všechny potřebné nástroje k tvorbě GUI aplikací. Programátor si musí kód pro vytvoření všech oken a tlačítek napsat sám, ale díky připraveným třídám, definujícím všechny potřebné grafické komponenty, je takový návrh poměrně snadný. Velkou výhodou Javy je nezávislost na platformě.

Z těchto možností jsem nakonec zvolil jazyk C++ a .NET Framework v prostředí Microsoft Visual Studio 2005, především kvůli tomu, že jsem tehdy měl s tímto prostředím více zkušeností. Druhým důvodem byla možnost použít v animaci kód řadicích algoritmů ve stejné podobě, v jaké jsou zapsány ve vznikající studijní opoře.

3.2.3 Výběr stylu animace

Existuje mnoho podobných animačních programů pro demonstraci řadicích algoritmů. Hledal jsem příklady takových programů na webu, abych získal přehled o tom, jaké animace na toto téma již existují a jak lze k tomuto úkolu přistupovat. V zásadě se dají rozlišit dva typy těchto animací.

První typ redukuje celé řazení pouze na operace výměny dvou prvků. Většina řadicích algoritmů se dá takto chápat. Když zanedbáme různá porovnávání, průchody cyklem či rekurzivní zanořování, proces řazení je posloupností výměn prvků v řazeném poli. Právě tyto výměny jsou operace kritické z hlediska časové náročnosti programu. Proto animace, která zobrazuje pouze výměny, může dobře posloužit pro porovnání rychlosti různých metod. Významná vlastnost řadicích metod je velikost skoku, s jakým se prvky posunují ke své konečné pozici. U pomalých metod se prvek posunuje pouze o jednu pozici, rychlejší metody mívají tento posun větší. Tato charakteristika řadicí metody se dá na tomto typu animace také dobře pozorovat.

Druhý typ animací řadicích algoritmů zobrazuje podrobně všechny děje, ke kterým během řazení dochází. Ukazuje všechna porovnávání, posuny indexů, a další mechanizmy, které vedou k samotným výměnám prvků. Takto podrobné zobrazení vede ke zkreslení představy o rychlosti algoritmu, ale mnohem lépe pomůže pochopit, jak vlastně algoritmus pracuje.

Cílem mojí práce je pomoci v pochopení řadicích algoritmů, proto je jasnou volbou druhý typ animace, který podrobně ukazuje celý průběh řazení a každou metodu animuje individuálním způsobem vystihujícím charakter právě této metody. Některé algoritmy v mé animaci nejsou sice zobrazeny zcela podrobně se všemi kroky, zachovávají si však svůj specifický charakter.

3.2.4 Návrh uživatelského rozhraní

Na ovládání animace existuje jediný požadavek, a tím je jednoduchost použití. Inspiroval jsem se moderními přehrávači médií, které také kladou důraz na jednoduchost. Dominantní jsou tlačítka pro spuštění a zastavení animace, jejich funkčnost doplňují tlačítka pro manuální krokování. Použití těchto tlačítek i ostatních ovládacích prvků, kterých není mnoho, je velmi intuitivní. Blíže je ovládání popsáno v uživatelském manuálu (dostupný na CD - příloha 4).

3.2.5 Konečná verze animace

Po ukončení práce na studijní opoře a testovacím programu jsem se vrátil k prototypu animace. Uživatelské rozhraní bylo stále vyhovující, zvolený styl podrobné demonstrace bylo vhodné zachovat. Nyní bylo nutné v animaci prezentovat řadicí metody tak, jak jsou uvedeny ve studijní opoře, a doplnit animaci o složitější metody, nastíněné v opoře pouze pseudokódem. Přitom bylo nutné zcela přepracovat systém vykreslování animace, a tak z původního prototypu zbylo jen uživatelské rozhraní. Animace metod byla implementována znovu, a byly přidány některé užitečné vlastnosti jako možnost změny velikosti okna animace. Při roztáhnutí na celou obrazovku se tak dá animace bez problémů používat pro výklad na přednáškách.

4 Řadicí algoritmy

Tato kapitola probírá teorii řadicích algoritmů, použitých v animačním programu. Přímo na teoretický výklad jednotlivých metod pak navazuje popis konkrétního použití dané metody v mojí práci.

4.1 Teorie řadicích algoritmů – obecně

Na vstupu řadicího algoritmu je vždy kolekce s lineárním uspořádáním prvků, datová struktura obsahující položky stejného typu. Pokud jsou položky takové kolekce porovnatelné podle některé ze svých vlastností, pak je možné tuto vlastnost použít jako klíč pro řazení této kolekce. Na výstupu řadicího algoritmu jsou všechny položky kolekce uspořádány podle daného klíče. Podle toho, jakým způsobem je možné přistupovat k jednotlivým položkám řazené kolekce, rozlišujeme typy řazení na sekvenční a nesekvenční.

4.1.1 Rozdělení podle přístupu k řazeným datům

Sekvenční řazení se používá tehdy, když datová struktura neumožňuje přímý přístup ke svým položkám, a tak se položky kolekce musí zpracovávat sekvenčně jedna po druhé. Někdy se tyto metody označují jako vnější řazení, řazení souborů nebo seznamů. Historicky to byly významné algoritmy, protože paměťové média prvních počítačů často umožňovala pouze sekvenční přístup. Dnes je použití těchto metod spíše výjimečné, proto se těmto algoritmům moje animace nevěnuje.

Nesekvenční řazení využívá přímého přístupu k libovolnému prvku kolekce. Označuje se jako vnitřní řazení, nebo řazení polí. Umožňuje využití mnoha různých principů řazení a nabízí tak větší efektivitu řazení. Těmito algoritmy se zabývá největší část studijní opory a celá moje animace.

4.1.2 Rozdělení podle principu řazení

Máme tedy řadicí algoritmy rozděleny z hlediska typu zpracovávaných dat. Dále se budeme zabývat metodami řazení polí. Ty se dají rozdělit podle principu, na jakém pracují. Než budeme probírat jednotlivé principy, je nutné zmínit pojem „stabilita“ řazení.

„Řadicí metodu lze označit za stabilní, jestliže zachová relativní uspořádání duplicitních klíčů souboru.“

Definice převzata [3]. Tento pojem je také podrobněji a na příkladech vysvětlen ve studijní opoře.

4.1.2.1 Princip výběru

Algoritmy založené na principu výběru jsou velmi populární mezi začínajícími programátory, protože jsou jednoduché a dají se snadno odvodit bez znalosti pokročilejších programovacích technik. Tyto metody rozdělují pole na seřazenou a neseřazenou část. Na počátku tvoří celé pole neseřazenou část a seřazená část je prázdná. V každém kroku metoda projde celou neseřazenou část a nalezne v ní minimum (nebo maximum, záleží na konkrétní implementaci). Nalezený extrém se zařadí na konec do seřazené části, neseřazená část se o tento prvek zmenší. Výběr extrému a jeho přesun se opakuje, dokud se nevyprázdní neseřazená část pole. Jednotlivé algoritmy založené na tomto principu se liší způsobem výběru extrémního prvku v nesetříděné části. Zástupcem těchto algoritmů v mé animaci je Bubble sort.

4.1.2.2 Princip vkládání

Princip vkládání je velmi blízký výše uvedenému principu výběru, pro začátečníka je také snadno pochopitelný. Pole je opět chápáno jako seřazená a neseřazená část. V seřazené části je tentokrát hned od začátku jeden prvek, typicky první prvek pole. Algoritmus odebere první prvek z neseřazené části, a vloží ho do seřazené na správné místo, tak aby tato část zůstala seřazená. Takto jsou všechny prvky z neseřazené části postupně vloženy do seřazené části. V té se nakonec nachází celé pole, a řazení je úspěšně ukončeno. Jednotlivé implementace se liší způsobem nalezení pozice pro vložení prvku do seřazené části. To lze zefektivnit například pomocí binárního vyhledávání. Tyto algoritmy nemají v mojí animaci žádného zástupce.

4.1.2.3 Princip rozdělování

Tento princip už je poněkud náročnější na pochopení, ale metody pracující tímto způsobem patří mezi ty nejrychlejší, proto je význam těchto algoritmů značný. Řazené pole je rozděleno na levou a pravou část takovým způsobem, aby všechny prvky v levé části byly menší nebo rovny hodnotě jistého prvku v tomto poli a všechny prvky v prvé části byly větší než tento prvek. Na každou ze vzniklých částí aplikujeme znova stejný proces rozdělení. Takto dělíme pole na stále menší části, až v poslední fázi máme části se dvěma prvky a problém se redukuje na uspořádání těchto dvou prvků. Nejjednodušší způsob, jak dosáhnout takového postupného dělení pole, je rekurzivní zanořování. Zástupce tohoto principu v mé animaci je Quick sort.

4.1.2.4 Princip slučování

Algoritmy založené na principu slučování vyhledávají v řazeném poli úseky, které lze již považovat za seřazené posloupnosti. V nejhorším případě, kdy jsou prvky pole seřazeny proti směru vyhledávání posloupností, představuje každý prvek pole jednoprvkovou seřazenou

posloupnost. Algoritmus pak bere vždy dvě posloupnosti a setřídí je dohromady. To je proces, při kterém vznikne ze dvou seřazených posloupností jedna seřazená posloupnost. Vzniklé posloupnosti se znovu mezi sebou setřiďují, dokud není celé pole jediná seřazená posloupnost. Moje animace demonstruje činnost dvou takových algoritmů, Merge sortu a ListMerge sortu.

4.1.2.5 Řazení tříděním

Tento princip je inspirován třídicími stroji z předpočítačové éry. Číslo, které je klíčem pro řazení položek pole, se rozdělí na jednotlivé číslice – hodnoty jednotlivých řádů. Pole se pak řadí nejdříve podle hodnoty klíče v řádu jednotek, pak desítek, stovek, a řád se dále zvyšuje až do řádu největšího čísla v poli. Při použití stabilní řadicí metody bude po seřazení dle největšího řádu pole seřazené kompletně. Důležité je, že není nutné používat žádnou z klasických řadicích metod. Protože číslice na každé pozici (hodnoty všech řádů) mohou nabývat pouze hodnot -9 až 9, může být jejich seřazení realizováno roztříděním číslic do seznamů reprezentujících každou z možných hodnot. Algoritmus řadící pole za pomoci třídění je Radix sort, jeho činnost ukazuje má animace.

4.2 Potřebné datové struktury

Než se začneme zabývat samotnými řadicími algoritmy, je nutné si uvědomit, že metody založené na různých, často velmi odlišných principech, budou mít každá své specifické požadavky. Někdy se jedná o nároky na paměť, jindy je nutným předpokladem prostředí schopné rekurze. Jiným běžným požadavkem může být možnost využívat abstraktních datových typů jako je seznam, fronta nebo zásobník. Protože některé algoritmy, prezentované v mojí práci, těchto typů využívají, uvedu zde nejdřív popis použitých abstraktních datových typů.

4.2.1 Seznam

Kolekce prvků stejného typu, položky jsou lineárně uspořádány. Každý prvek má právě jednoho předchůdce (pouze první prvek předchůdce nemá) a právě jednoho následníka (výjimkou je poslední prvek). První prvek je výjimečný i tím, že je to jediný prvek, ke kterému je umožněn přímý přístup. Ke všem dalším prvkům se musím sekvenčně propracovat pomocí operace, která vrací následníka prvku seznamu.

Implementace tohoto pohybu seznamem může být různá. Ve studijní opoře je v souvislosti s řadicími algoritmy použit takový seznam, jehož prvky mají kromě užitečných dat také ukazatel na další prvek. Seznam je pak provázán těmito ukazateli, k označení seznamu potřebuji pouze ukazatel na první prvek. Konec seznamu rozpoznám podle prvku, jehož ukazatel na další prvek má hodnotu NULL. Tato definice je pak pro naše potřeby doplněna o ukazatel na aktivní prvek. Tím

získáme možnost přímého přístupu nejen k prvnímu, ale i k aktivnímu prvku, a možnost tuto aktivitu posouvat po směru ukazatelů provazujících seznam. Rozšířením definice seznamu o aktivitu se mnohé operace se seznamem výrazně zjednoduší.

Seznam je velmi univerzální datová struktura s širokými možnostmi využití, které jsou dány množinou operací implementovaných pro manipulaci s daty seznamu.

4.2.2 Fronta

4.2.2.1 Teorie

Abstraktní datový typ fronta se dá chápat jako seznam, u kterého jsou striktně omezeny možnosti přístupu k jeho prvkům. Rozlišuje se zde začátek a konec fronty, nové prvky je možné přidávat pouze na konec, zatímco odebrat prvek nebo číst jeho data je umožněno jen pro prvek na začátku. Znamená to, že se jedná o strukturu typu FIFO (First In, First Out), pro kterou platí zásada, že prvky jsou odebírány a zpracovávány v takovém pořadí, v jakém byly do fronty přidány.

Implementace fronty tedy může využít datovou strukturu seznam, nad kterou definuje operace inicializace, vložení nového prvku na konec, čtení a rušení prvku ze začátku. Často je vhodné rozšířit tuto definici o další operace, ale již v této základní podobě je fronta plně použitelná.

4.2.2.2 Použití fronty v mé práci

Frontu využívá algoritmus ListMerge sort, který je ve studijní opoře popsán pouze pseudokódem. Kompletní algoritmus jsem implementoval v testovacím programu a grafické znázornění jeho činnosti ukazuje moje animace. Proto je v těchto programech výše popsaným způsobem implementována i fronta. Operace nad ní jsou rozšířeny o vlastnost aktivního prvku. Jeho čtení však není umožněno, tím by byl charakter fronty porušen. Povoleno je pouze aktivitu nastavovat, posouvat a testovat, zda je seznam aktivní.

4.2.3 Zásobník

4.2.3.1 Teorie

Zásobník je možné, podobně jako frontu, implementovat jako seznam, u něhož jistým způsobem omezíme možnosti manipulace s jeho prvky. Vkládání i odebírání je v případě zásobníku umožněno vždy ze stejného konce, ten se označuje jako vrchol zásobníku. Je to jediné místo

přístupu do této struktury. To vede k vlastnostem struktury typu LIFO (Last In, First Out), kde jsou prvky čteny a odebírány v opačném pořadí, než v jakém do zásobníku přišly.

Je tedy nutné implementovat operace pro inicializaci, vložení prvku na vrchol zásobníku, čtení a odebírání prvku z vrcholu.

4.2.3.2 Použití zásobníku v mé práci

Pro výukové a testovací účely nemusí být soubor testovacích dat příliš velký, tuto velikost navíc známe předem, a proto nemusí být zásobník implementován tak, aby mu byla paměť přidělována skutečně dynamicky. Datovou strukturu s vlastnostmi zásobníku je možné realizovat i na statickém poli. Místo ukazatele na vrchol zásobníku je nutné uchovávat si index pole, který plní roli tohoto ukazatele.

Tímto způsobem je implementován zásobník v mém testovacím programu. Jeho služeb využívá ke své činnosti nerekurzivní Quick sort. Operace pro práci se zásobníkem jsou díky využití statického pole velmi jednoduché a názorné, takže i tato část kódu může být snadno využita ke studiu a experimentům.

4.3 Bubble sort

4.3.1 Teorie

Základní řadicí metoda, snadno se dá odvodit bez hlubších znalostí této problematiky, avšak jedna z nejpomalejších metod. Její vnitřní cyklus postupně porovnává dvojice sousedních prvků, a pokud jsou porovnané prvky seřazeny opačně (vzhledem k sobě navzájem), prohodí se. Tohoto mechanismu se dá využít pro řazení na principu výběru nebo vkládání. Studijní opora ukazuje obě možné implementace, pro animaci jsem zvolil nejtypičtější variantu založenou na principu výběru.

4.3.1.1 Bubble sort na principu výběru

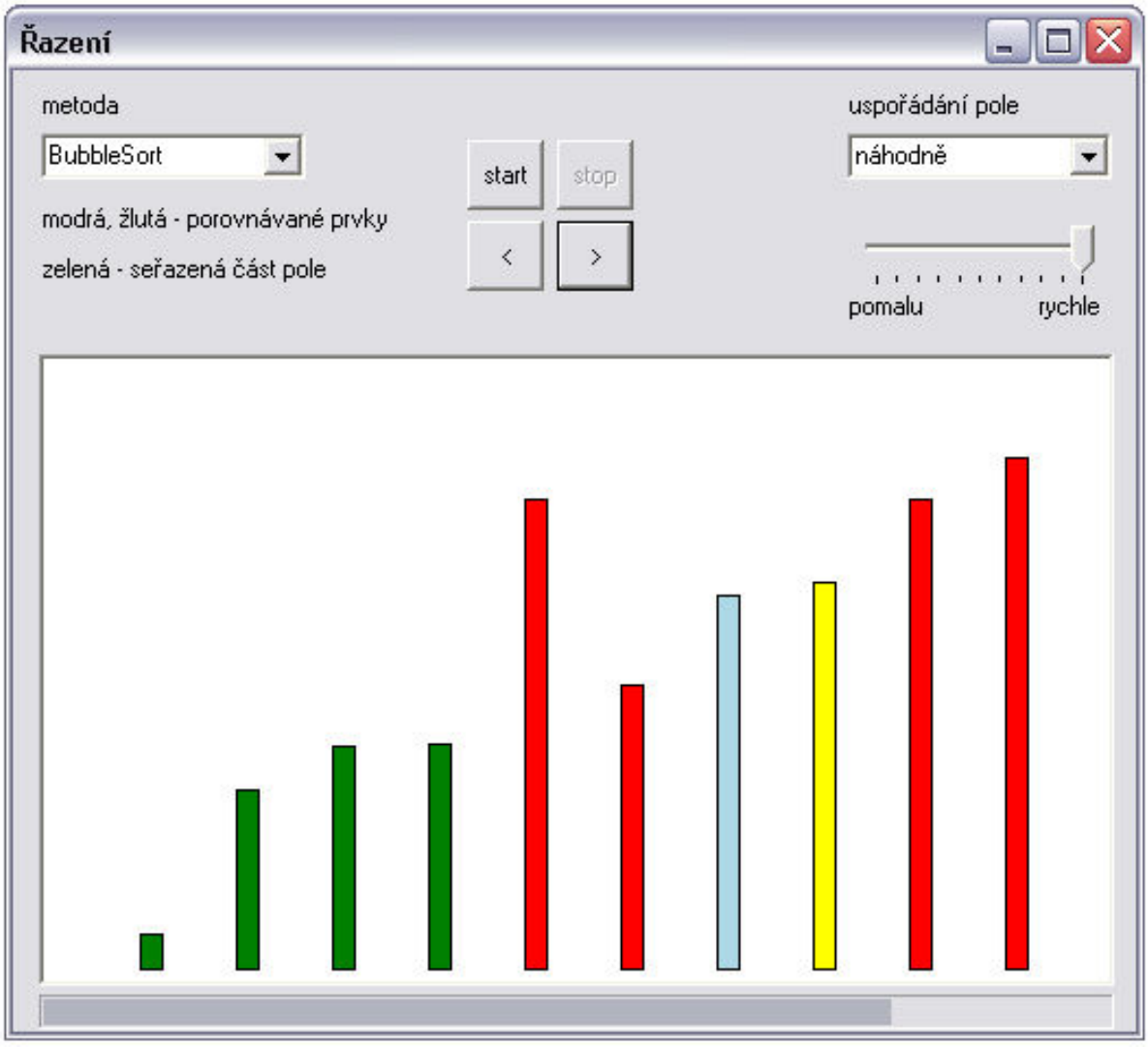
Algoritmus prochází neseřazenou částí pole, začne na jejím konci a porovnává (popřípadě prohazuje) všechny sousední dvojice prvků. Nejmenší prvek z neseřazené části se tak ze své aktuální pozice pohybuje skrz celou tuto část, až se zařadí na konec do seřazené části. Odtud pochází název „Bubble sort“, protože nejmenší prvek jakoby „probublává“ skrz část pole.

Stejný algoritmus lze implementovat mnoha různými způsoby, směr procházení pole bývá často opačný a místo nejmenších prvků „probublávají“ polem ty největší. Existují varianty snižující počet porovnání na nutné minimum, počet výměn prvků však snížit nedokáží a proto nepřináší významné zvýšení rychlosti algoritmu.

4.3.2 Použití Bubble sortu v mojí práci

Studijní opora předkládá studentům dvě varianty zápisu tohoto algoritmu založeného na principu výběru, a jednu na principu vkládání. Vzhledem k jednoduchosti těchto algoritmů je v opoře uveden jejich kompletní kód, který je obsažen i v testovacím programu.

Demonstrace činnosti tohoto algoritmu v animaci nebyla požadována, přesto jsem se rozhodl rozšířit moji animaci o tuto metodu, konkrétně o variantu pracující na principu vyhledávání. Primárním účelem není ukazovat, jak Bubble sort funguje. Tento algoritmus je natolik známý a snadno pochopitelný, že vysvětlovat jeho princip studentům informatiky je většinou zbytečné. Právě této všeobecné znalosti využívá moje animace k tomu, aby uživatele seznámila se svým ovládáním a se způsobem, jakým jsou algoritmy demonstrovány. Po spuštění animace je implicitně zvolena právě tato metoda, a uživatel si tak může na známém algoritmu vyzkoušet práci s animačním programem.



Demonstrační animační program, Bubble sort

Animace zobrazuje deset barevných sloupců, každý z nich symbolizuje jeden prvek řazeného pole. Výška sloupce představuje hodnotu vlastnosti prvku, která slouží jako klíč pro řazení. Tohoto sloupcového zobrazení využívá většina algoritmů v mojí animaci. Žlutou a světle modrou barvou jsou označeny prvky, které se právě porovnávají, zelené sloupce představují prvky v seřazené části, ostatní prvky jsou červené. Animace zobrazuje všechna porovnání, je tedy vidět jak algoritmus postupně prochází neseřazenou částí pole a kde je to potřeba, provádí výměny. Dá se pozorovat, jak nejmenší prvek neseřazené části „probublává" na konec seřazené části a tam se porovnávání zastavuje. Stejně tak lze pozorovat, že použitá varianta algoritmu kontroluje, jestli při průchodu neseřazenou částí došlo k nějaké výměně, a pokud ne, celé pole je již seřazeno a algoritmus končí.

4.4 Quick sort

4.4.1 Teorie

Quick sort je v praxi jedna z nejužitečnějších metod řazení, protože je ve většině případů nejrychlejší. Pracuje na principu rozdělování, implementace ve studijní opoře odděluje od samotného algoritmu funkci „partition", která provede nad daným úsekem pole rozdělení do dvou částí. Celý algoritmus pak obsahuje pouze volání funkce partition a dvojí volání sám sebe, první pro levou část vytvořenou ve funkci partition a druhé pro pravou část. Rekurzí se tak zajistí postupné dělení pole na stále menší části, až do velikosti dvou prvků.

4.4.1.1 Partition

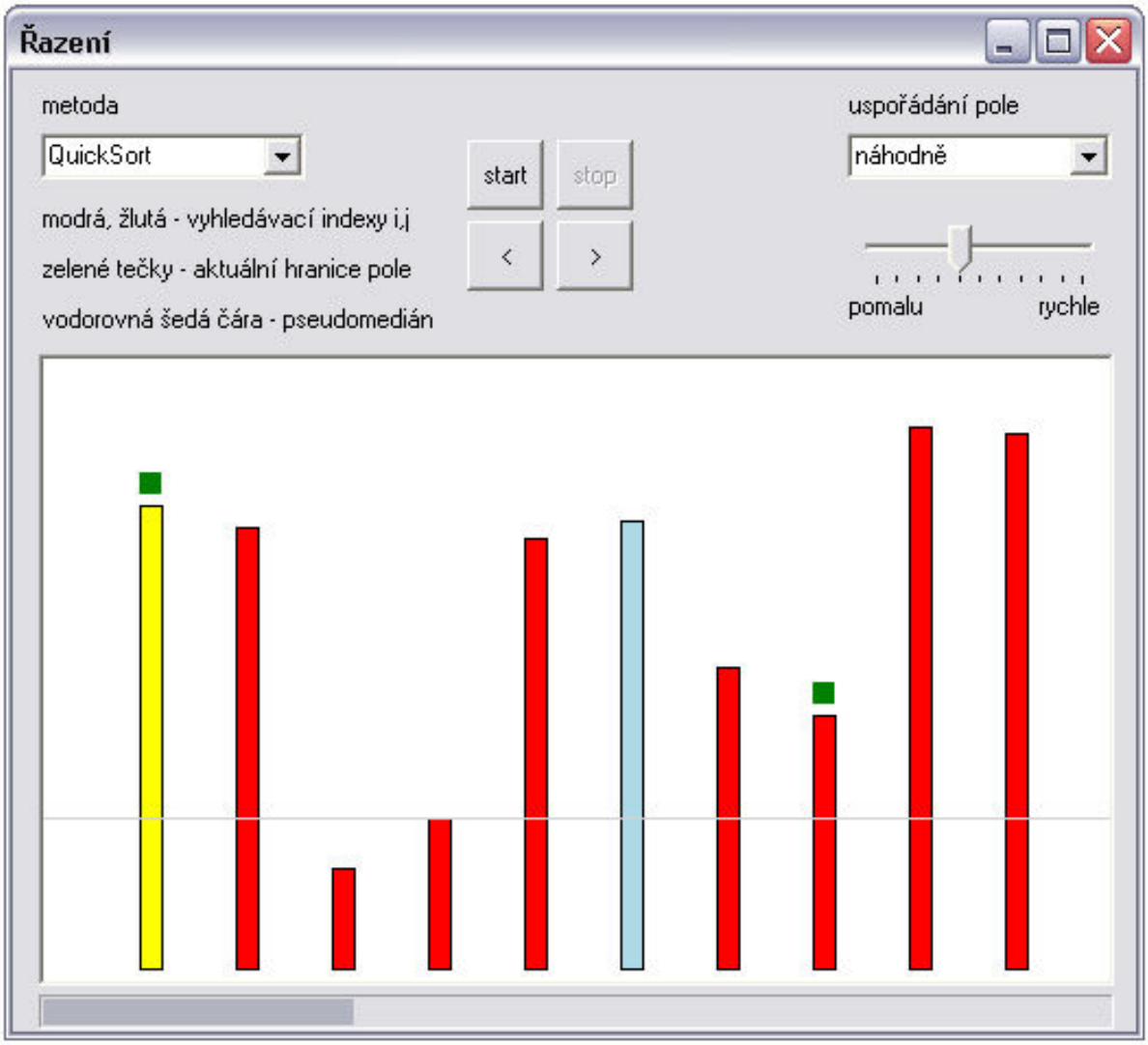
Tato funkce je jádrem algoritmu Quick sort. Řazené pole (nebo jeho úsek) rozdělí na dvě části tak, že v levé části jsou všechny prvky menší nebo rovny hodnotě jistého prvku z daného úseku pole a v pravé části se nachází prvky větší než je tato hodnota. Tím rozdělujícím prvkem by v ideálním případě byl „medián". To je prvek, jehož hodnota je větší než polovina čísel v daném úseku pole a menší než druhá polovina. Určení mediánu by však byla neúnosně náročná operace, proto je místo něj použit „pseudomedián", což je libovolný prvek z daného úseku pole. Algoritmus bude stále pracovat správně, pokles efektivity je zanedbatelný. Typicky se jako pseudomedián volí prvek ze středu aktuálního úseku pole, toho se drží i studijní opora.

K rozdělení jsou potřeba dva vyhledávací indexy, které jsou na počátku inicializovány na levý a pravý okraj aktuálního úseku. Levý index se pak pohybuje doprava, dokud nenalezne prvek, který nepatří do levé části, protože je větší než pseudomedián. Pravý index se naopak pohybuje doleva, dokud nenarazí na prvek menší nebo roven hodnotě pseudomediánu. Nalezené prvky se prohodí, tím se dostanou do té části, do které patří, a indexy se mohou posunout opět o jednu pozici

blíže k sobě. Ve chvíli, kdy se tyto vyhledávací indexy překříží, jsou všechny prvky ve správné části, funkce splnila svůj úkol a končí.

4.4.2 Použití Quick sortu v mojí práci

Moje zadání pro animační program nepožaduje demonstraci tohoto algoritmu, rozhodl jsem se ji však implementovat jako rozšíření. Důvodem je nesporně velký význam Quick sortu, jeho široké použití v praxi a zajímavé programovací techniky, kterých využívá. Přestože se jedná o velmi důležitý algoritmus, začínající studenti mají někdy problém s jeho pochopením, a tak se mu vyhýbají. Proto se v mojí animace snažím o co nejnázornější ukázku jeho činnosti.



Demonstrační animační program, Quick sort

Hodnoty prvků jsou reprezentovány výškou sloupců, stejně jako u Bubble sortu. Zelené tečky nad sloupci označují okraje aktuálního úseku pole. Na nich je možné sledovat, jak se

algoritmus rekurzivně zanořuje, když volá sám sebe pokaždé pro jinou část pole. Žlutou a světle modrou barvou jsou zvýrazněny pozice, na kterých se nachází vyhledávací indexy. Šedá čára, horizontálně procházející skrz zobrazené pole, představuje hodnotu pseudomediánu. Můžeme tak sledovat pohyb vyhledávacích indexů: pohyb levého indexu doprava se zastaví, když sloupec sahá nad hranici pseudomediánu, pohyb pravého indexu doleva skončí nalezením sloupce menšího než pseudomedián. Takto nalezené prvky se prohodí a indexy se posunou o jednu pozici blíž k sobě.

Protože animace zobrazuje takto detailní průběh algoritmu, rychlost metody není zřejmá. Ta spočívá v nízkém počtu výměn prvků, zatímco animace zobrazuje i všechny posuny indexů. Představu o efektivitě metody si tedy student asi nevytvoří, ale může o to lépe pochopit princip, na kterém je algoritmus založen. To je také smyslem celé této práce.

4.5 Merge sort

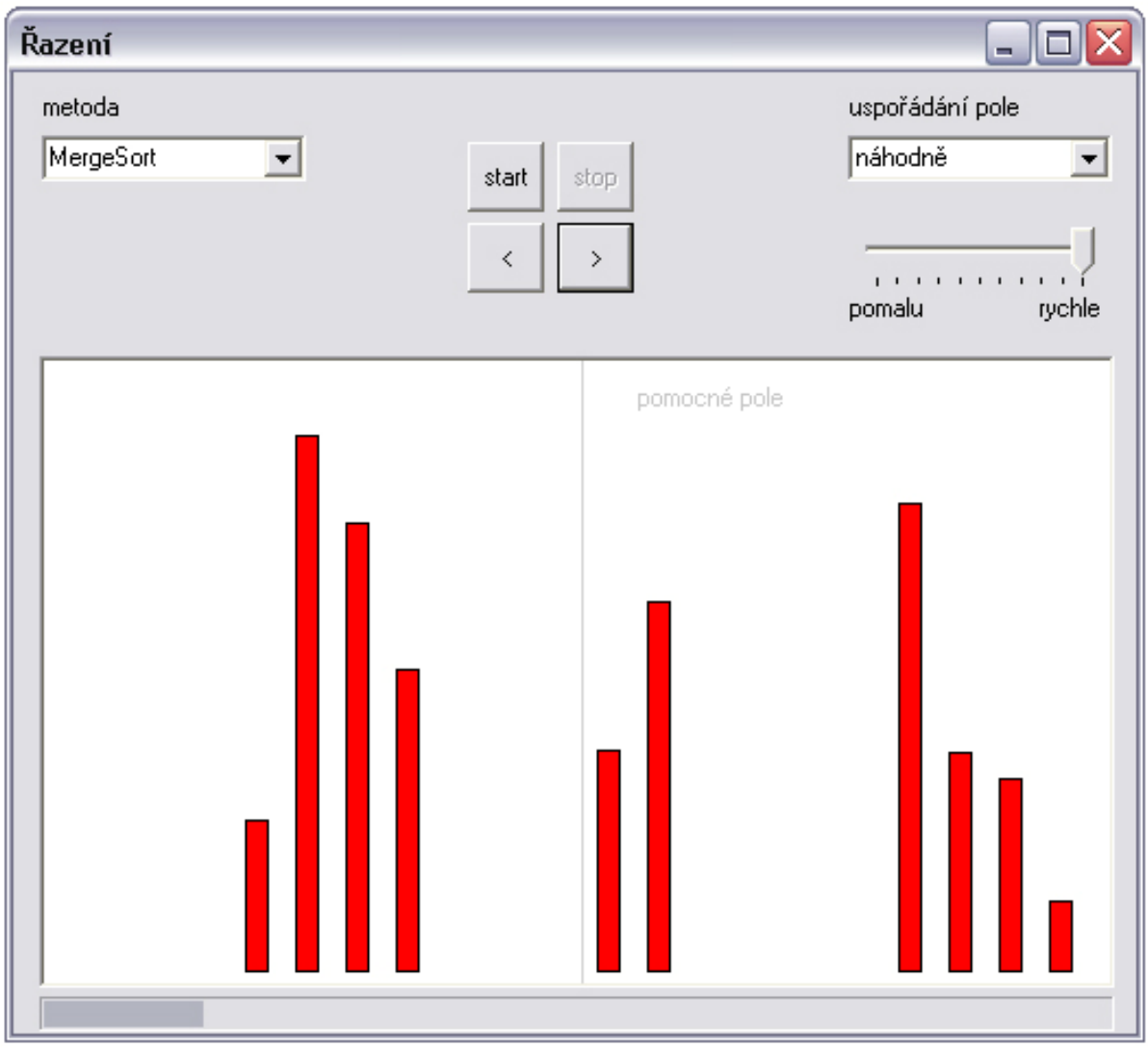
4.5.1 Teorie

Rychlá a zajímavá metoda, avšak poměrně náročná na implementaci. Merge sort pracuje na principu slučování. Vyhledává neklesající posloupnosti v řazeném poli, ty setřiďuje do pomocného pole. K činnosti algoritmu je tedy potřeba dvojnásobný prostor v paměti. Celé pole se postupně přesune do pomocného pole, nově vzniklé neklesající posloupnosti se opět setřiďují a výsledné posloupnosti se ukládají zpět do původního pole. Celý proces přesunu do pomocného pole a zpět se opakuje, dokud není v původním poli jediná neklesající posloupnost, což je požadovaný výsledek řazení.

Neklesající posloupnosti se ve zdrojovém poli vyhledávají z obou stran zároveň, algoritmus v cyklu porovnává prvky, které jsou na okrajích pole a menší z nich vždy přesune do cílového pole na konec nově vznikající posloupnosti. Okraj, na kterém se přesunutý prvek nacházel, se posune do středu pole na vedlejší prvek. Pokud je tento prvek menší než byl předešlý prvek, znamená to, že neklesající posloupnost na této straně skončila, z opačného okraje se tedy postupně přesouvají prvky tak dlouho, dokud neskončí neklesající posloupnost i tam. Takto se okraje pole stále přibližují a nalézané posloupnosti se setřiďují do cílového pole. Tam je potřeba výsledné posloupnosti ukládat střídavě zleva a zprava směrem do středu, aby byly tyto posloupnosti nalezeny, až bude toto pole zdrojové.

4.5.2 Použití Merge sortu v mojí práci

Merge sort je ve studijní opoře popsán pouze slovně s doplňujícími obrázky, kód ani pseudokód uveden není. Testovací program tedy tento algoritmus také neobsahuje. Pro animační program je tento algoritmus součástí mého zadání.



Demonstrační animační program, Merge sort

Výška sloupců opět reprezentuje hodnotu prvků. Vertikální čára uprostřed rozděluje původní pole (vlevo) a pomocné pole (vpravo). Animace se drží úrovně popisu, na jaké je algoritmus představen v opoře. To znamená, že se jedná o poměrně abstraktní zobrazení, kde jsou vidět pouze přesuny prvků z jednoho pole do druhého. V případě této metody je to dostatečně názorné. Je vidět, jak jsou prvky odebírány z obou konců zdrojového pole, jak jejich setříděním vznikají nové a větší seřazené posloupnosti a jak se tyto posloupnosti skládají do cílového pole střídavě zleva a zprava.

Zobrazovat jakékoli další mechanismy tohoto algoritmu je zbytečné, protože princip Merge sortu je touto animací vyjádřen kompletně a konkrétní realizace může vypadat různě.

4.6 ListMerge sort

4.6.1 Teorie

ListMerge sort patří mezi algoritmy, pracující na principu slučování. Položky pole řadí bez přesunu, používá k tomu zřetězených seznamů, odtud název ListMerge sort. Pokud není samotné řazené pole seznamem, a jeho položky nemají ukazatele na další prvek, je nutné vytvořit si pomocné pole, které bude obsahovat pro každý prvek z řazeného pole ukazatel pro zřetězení. Ve statickém poli s přímým přístupem k prvkům nejlépe jako ukazatel poslouží pouhý index do tohoto pole.

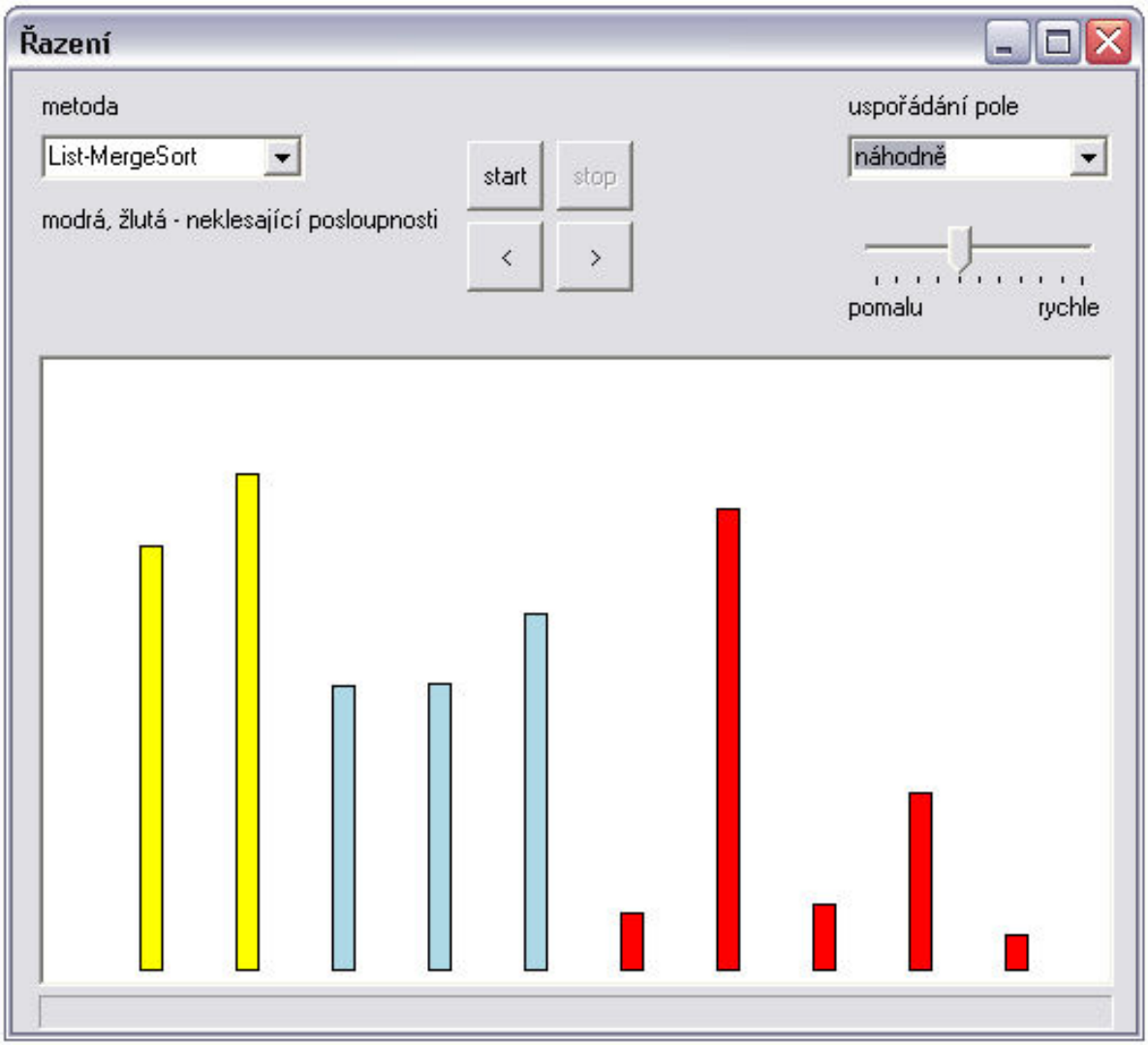
Před začátkem samotného řazení musí algoritmus projít celé řazené pole, vyhledat v něm neklesající posloupnosti a inicializovat pomocné pole ukazatelů tak, aby se nalezené posloupnosti zřetězily do oddělených seznamů. Je nutné uložit si začátky těchto seznamů, k tomu je v tomto případě vhodná struktura typu fronta.

Algoritmus pak ze začátku fronty odebere dva prvky – začátky prvních dvou seznamů obsahujících neklesající posloupnosti. Tyto dvě posloupnosti zatřídí do sebe, vznikne tak jedna nová neklesající posloupnost v jednom seznamu a jeho začátek se vloží na konec fronty. Setřídění dvou seznamů je popsáno například ve studijní opoře[1], funkce „mergeLists“ v úseku „5.6 – Řazení setřiďováním“. Z fronty se vždy odebírají první dva prvky a opakuje se proces setřídění a vložení nového seznamu na konec fronty, dokud fronta neobsahuje pouze jediný začátek seznamu. V tom se nachází již seřazené pole.

Nakonec je třeba podle vzniklého seznamu fyzicky přeskládat prvky řazeného pole. To lze udělat například pomocí MacLarenova algoritmu, který popisuje studijní opora [1].

4.6.2 Použití ListMerge sortu v mojí práci

Studijní opora tento algoritmus popisuje pseudokódem, který je v podstatě úsekem skutečného kódu doplněného slovním popisem. Pro testovací program jsem slovní popis nahradil potřebným kódem, kompletní zdrojový text tohoto algoritmu je zde tedy k dispozici. Demonstrace činnosti ListMerge sortu v animačním programu je požadována.



Demonstrační animační program, ListMerge sort

Způsob sloupcového zobrazení prvků pole zůstal stejný, jako u výše uvedených algoritmů. Problémem při zobrazení této metody je fakt, že prvky zůstávají fyzicky na svých místech a mění se pouze ukazatele a seznamy, do kterých jsou prvky provázány. Zobrazení skutečného stavu řazeného pole a příslušných ukazatelů by se změnilo v nepřehlednou změť šipek, proto je nutné zobrazovat činnost algoritmu na velmi abstraktní úrovni. Pouze na začátku odpovídá zobrazené pole skutečnému poli v paměti. Žlutou barvou je označen první seznam (neklesající posloupnost), modrou barvou druhý. Tyto dva seznamy algoritmus setřídí do jednoho, samotný proces setřídění však animace neukazuje, pouze zobrazí výsledný seznam, jehož prvky označí žlutou barvou. Ten zůstává na místě původních dvou seznamů, a proto je z animace vidět, jak se setřiďování posouvá na vedlejší dva seznamy, a cyklicky prochází pole stále dokola dokud v něm nezůstane pouze jediný seznam. Ten obsahuje seřazené pole.

Kvůli vysoké míře abstrakce této animace není pochopení tohoto algoritmu triviální, spolu se studijní oporou však animace nabízí jednoduchý pohled na tuto metodu.

4.7 Radix sort

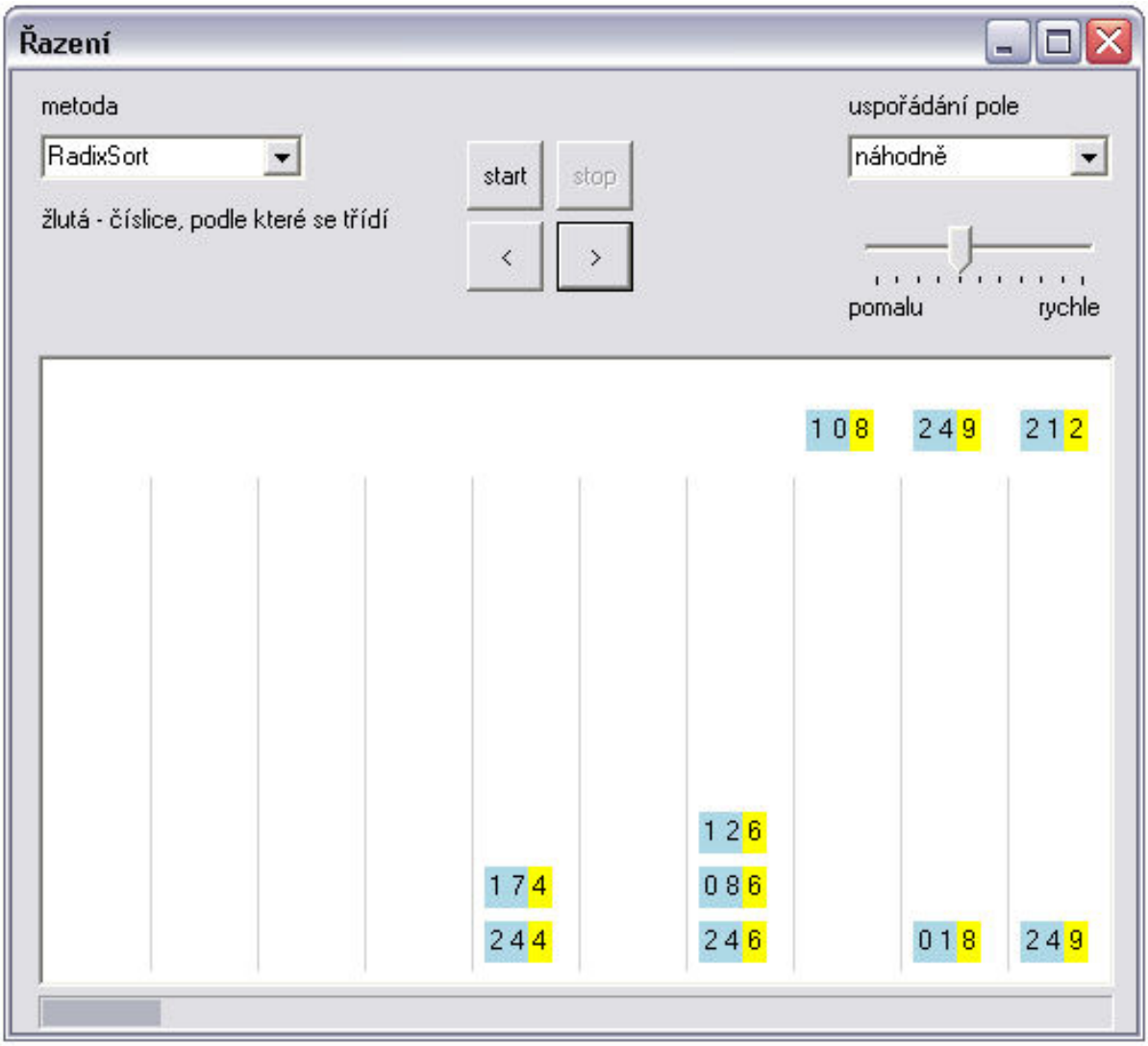
4.7.1 Teorie

Rychlá a zajímavá metoda, jejíž princip je zcela odlišný od ostatních prezentovaných metod. Řazení položek pole se realizuje pomocí jejich třídění. U ostatních řadicích metod je klíčová činnost porovnávání prvků a jejich výměny, Radix sort však nikdy prvky pole neporovnává mezi sebou. Prvky pole se roztřiďují podle hodnot jednotlivých číslic do seznamů (popsáno v této kapitole v úseku 4.1.2.5 – Řazení tříděním). Na začátku algoritmus prochází polem od jeho začátku až do konce tak, jak je pole uloženo v paměti. Každý prvek zařadí do seznamu, do kterého patří podle hodnoty číslice na aktuální pozici. Prvky se nikam nepřesouvají, do patřičných seznamů se řadí pomocí ukazatelů. Když je celé pole roztříděné, projdou se vytvořené seznamy počínaje seznamem pro nejmenší číslici (pokud počítáme i se zápornými čísly, bude to seznam pro číslici -9), až po seznam pro největší číslici (9), a spojí se do jediného seznamu. Ten reprezentuje aktuální seřazení pole. Když se pak v dalším kroku opět prvky roztřiďují, tentokrát podle číslice na pozici posunuté doleva (vyšší řád), berou se v takovém pořadí, v jakém jsou umístěny v celkovém seznamu. Tím je zajištěna stabilita a funkčnost metody.

Roztřídění pole a následné zřetězení do celkového seznamu se opakuje pro každou číslici. Pozice této číslice se posouvá – řád se zvyšuje, dokud není pole roztříděno podle nejvyššího řádu největšího čísla v poli. Když se pak zřetězí výsledný seznam, obsahuje již seřazené pole. Seřadit prvky v původním poli podle pořadí v jakém jsou zřetězeny je možné například pomocí MacLarenova algoritmu (viz. studijní opora [1]).

4.7.2 Použití Radix sortu v mojí práci

Ve studijní opoře je uveden pouze slovní popis s obrázky, kód tedy neobsahuje opora ani testovací program. Moje zadání požaduje animaci tohoto algoritmu.



Demonstrační animační program, Radix sort

Ze sloupcového zobrazení, použitého u ostatních algoritmů, by se uživatel animace nedozvěděl nic o tom, jakým způsobem metoda pracuje. Je nutné zobrazit skutečná čísla, pro demonstrační účely postačí omezení na kladná, nejvýše trojmístná čísla. Žlutou barvou je v každém čísle zvýrazněna číslice, podle které se právě třídí. Svislé čáry oddělují deset sloupců, které symbolizují seznamy pro hodnoty 0 až 9. Na začátku jsou všechna čísla v řadě nad prostorem pro tyto seznamy. Jsou seřazena tak, jak jsou skutečně ve vstupním poli. Postupně se jedno číslo po druhém odebírá z tohoto pole a zatřiďuje se do seznamu, do kterého patří. Když jsou všechny prvky roztříděny, budou se vytvořené seznamy procházet zleva doprava a prvky se z nich budou přesunovat opět do prostoru v horní části zobrazovacího okna, do jedné řady. Tato operace reprezentuje zřetězení seznamů do jednoho celkového seznamu, řada čísel už neodpovídá skutečnému uspořádání pole, ale pořadí, v jakém jsou prvky zřetězeny. Z animace je tak dobře vidět, v jakém pořadí se seznamy prochází a jak výsledné zřetězení vzniká, stejně jako fakt, že při dalších průchodech algoritmus bere prvky v tom pořadí, v jakém se nachází v celkovém seznamu.

Když je výsledný seznam hotov, posune se zvýraznění aktuální číslice o jednu pozici doleva, a znovu se roztřídění a zřetězení opakuje.

Tato demonstrace zobrazuje činnost algoritmu na poměrně abstraktní úrovni, ve skutečnosti se prvky nijak nepřesunují a všechny operace se provádí pouze nad seznamy a ukazateli. Přesuny prvků, které animace ukazuje, vyjadřují tuto práci se seznamy. Vše podstatné z činnosti tohoto algoritmu je tedy zobrazeno, animace plní svůj výukový účel.

5 Implementace animačního programu

5.1 Řadicí metody

První a zásadní otázkou při návrhu animačního programu byla otázka konkrétní implementace řadicích metod. Bylo nutné, aby demonstrace každého uvedeného algoritmu co nejvíce odpovídala popisu, který je uveden ve studijní opoře. U jednodušších metod, které mají v opoře uvedený kompletní kód, byla nejsnazší cesta vzít jejich zdrojové texty tak, jak jsou odladěny v testovacím programu, a doplnit jejich kód o vykreslování potřebných informací. Tímto způsobem je tedy řešena animace Bubble sortu a Quick sortu, u nichž je zdrojový kód v opoře kompletní. Pro ListMerge sort je ve studijní opoře pouze pseudokód, v testovacím programu jsem realizoval celý funkční kód této metody, který pak mohl být také použit pro animaci.

Složitější situace byla s algoritmy Merge sort a Radix sort. Studijní opora žádný kód neuvádí, a i kdyby byl tento kód k dispozici, dal by se těžko ve své původní podobě použít pro animaci, zejména u Radix sortu, který je potřeba zobrazovat zcela jinak, než ostatní metody. Rozhodl jsem se nevytvářet skutečný kód těchto řadicích algoritmů, který by sám o sobě byl opravdu pro řazení použitelný. Místo toho jsem navrhnul kód, který je zdánlivě podobný těmto algoritmům, dokáže také vstupní pole seřadit, ale je navržen přímo pro zobrazování činnosti těchto metod, nikoli pro skutečné řazení. Obsahuje tedy různé mechanismy, díky kterým je usnadněno zobrazování průběhu řazení, ale ztrácí se vlastnosti algoritmů důležité pro jejich efektivitu. Proto se nedají zdrojové texty animačního programu brát jako studijní materiál.

Takovému použití zabraňuje i fakt, že většina algoritmů je v animaci zobrazena velmi podrobně, a tak obsahuje jejich kód více textu pro zobrazování než pro samotné řazení. Proto i metody, jejichž kód je ve svém jádru shodný se skutečnou implementací daného řadicího algoritmu, se ztrácí mezi všemi texty zajišťujícími zobrazení průběhu metody.

5.2 Krokování

Animace musí zobrazovat jednotlivé kroky, které daný algoritmus vykonává nad řazeným polem během procesu řazení. Toho je možné docílit v zásadě dvěma způsoby. Nejsnazší cestou je přidat do standardního kódu řadicího algoritmu funkce, které ve vhodnou chvíli zobrazí aktuální stav řazeného pole. Mezi funkce pro zobrazení se pak vloží jistý čekací čas. Když je zavolána funkce realizující řadicí algoritmus, její průběh je v reálném čase zobrazován a pozastavován, uživatel vidí to, co se skutečně s řazeným polem v dané chvíli děje. Toto řešení však neumožní uživateli manuální krokování animace. Aby animace plnila svůj účel a přispěla k pochopení principů

řadicích metod, musí být uživateli umožněno animaci zastavit, vracet se v ní libovolně zpět či posouvat vpřed. Zastavení a spuštění animace je možné implementovat, krokování oběma směry však v tomto případě nelze zajistit.

Proto jsem se rozhodl pro druhý způsob implementace, který krokování umožní. Funkce vložené do řadicího algoritmu, které by měly zajistit vykreslení aktuálního stavu řazeného pole, se nahradí podobnými funkcemi, které však stav pole přímo nevykreslují, ale ukládají do krokovacího pole. Než tedy začne samotná demonstrace daného algoritmu, funkce realizující tento algoritmus kompletně proběhne, seřadí vstupní pole a všechny kroky, kterými při tom prošla, uloží do krokovacího pole. Animace pak pro zobrazení průběhu metody pouze prochází vytvořeným krokovacím polem a zobrazuje uložené stavy. Posun na další položku tohoto pole může zajistit automatický časovač, stejně tak může uživatel manuálně ovládat pohyb oběma směry.

5.3 Vykreslování

Každá metoda je demonstrována individuálním stylem, přesto najdeme u většiny společné vlastnosti, které je potřeba zobrazit. Především se jedná o aktuální stav samotného řazeného pole, dále například poloha různých indexů a další. Existuje více krokovacích polí, do kterých se tyto informace ukládají. Tato pole jsou společná pro všechny řadicí metody v animaci, zvolená metoda je naplní takovými informacemi, které jsou důležité právě pro tuto metodu. Některá krokovací pole zůstanou u většiny metod nevyužita. Funkce pro vykreslení je jen jedna, volá se při každém kroku nebo při překreslení celého okna. Tato funkce se podle metody, kterou bylo pole seřazeno, rozhodne, jak interpretovat informace uložené v krokovacích polích, a podle toho vykreslí aktuální krok.

6 Závěr

Tato práce vznikla jako součást projektu na podporu výuky předmětu Algoritmy. Výsledným celkem, složeným z bakalářských prací, patřících do tohoto projektu, bude nová studijní opora, upravená pro jazyk C, a kolekce animací vztahujících se k významným tématům tohoto předmětu. V mé práce jsem zpracovával část zabývající se algoritmy řazení.

Danou část studijní opory jsem přepsal, tak jak bylo zadáno. Vzniklý text zachovává v maximální možné míře text původní opory. To je vhodné zejména proto, že předmět bude i nadále vyučován na příkladech psaných v jazyce Pascal a hlavním studijním materiálem zůstane původní pascalovská opora. Pokud student narazí na problém během studia původní opory, nebo bude jen chtít vidět implementaci v jazyce C, bude moci sáhnout po nové studijní opoře a snadno nalezne odpovídající pasáž. Na druhou stranu není vhodné, pokud by novou oporu někdo nepoužil tak, jak bylo zamýšleno, ale bral by ji jako jediný studijní text. Kvůli zachování charakteru původní opory byly totiž na některých místech použity konstrukce, které nejsou pro jazyk C typické ani příliš vhodné. Na tato místa je čtenář vždy upozorněn, stejně jako budou studenti na přednáškách upozorněni, jakým způsobem mohou novou studijní oporu používat.

Pokud bychom podobný projekt realizovali znovu, stála by za zvážení možnost oprostit se od starých studijních materiálů a vytvořit kompletně novou studijní oporu, vztahující se pouze k jazyku C, případně k jinému, modernějšímu jazyku. Vzniklé dílo by pak mohlo sloužit jako samostatná učební pomůcka, použitelná i pro samostudium. Pro předmět algoritmy, který oprávněně zůstane vyučován v jazyce Pascal, by však neměla taková studijní opora větší smysl.

Vhodným doplněním studijní opory je program, obsahující testovací rozhraní pro všechny uvedené řadicí algoritmy. Mě posloužil pro odladění všech těchto algoritmů, stejně tak může být užitečný studentům pro jejich experimenty s programy tohoto zaměření. Obsahuje jednoduchou implementaci abstraktních datových typů fronta a zásobník, všechny algoritmy řazení z opory, rozšířené o kompletní kód ListMerge sortu, a zpracování vstupních a výstupních polí do souborů.

Další součástí této práce je demonstrační animační program pro algoritmy Merge sort, ListMerge sort a Radix sort. Kromě těchto požadovaných algoritmů jsem animaci rozšířil o Bubble sort a Quick sort. Animace zobrazuje práci každého algoritmu individuálním stylem, který vystihuje důležité vlastnosti dané metody. Proto může být animace použita při studiu předmětu algoritmy a splňuje svůj účel, kterým je pomoci pochopení vybraných metod. Animační program vznikal ve dvou fázích, intuitivně psaný prototyp musel být kompletně přepracován. Návrhu výsledné animace předcházelo studium různých variant řadicích algoritmů, možností jejich zobrazení a tvorby uživatelských rozhraní na platformě .NET.

Součástí zadání byl návrh kontrolních otázek, příkladů a testových úkolů. Ty jsou uvedeny v příloze. Zadání bylo ve všech bodech splněno, v některých bodech s účelnými rozšířeními.

Literatura

[1] Honzík, Jan. Algoritmy - Studijní opora, verze 3 - 27.2.2007. Dostupný na URL <https://www.fit.vutbr.cz/study/courses/IAL/private/Opora/Opora-IAL-2007-02-RP-verze-3.pdf>

[2] Wikipedia (online), Dostupný na URL <http://en.wikipedia.org/wiki/Herman_Hollerith>

[3] Sedgewick, Robert. Algoritmy v C, části 1-4. Praha: SoftPress, 2003.

Seznam příloh

Příloha 1. Kontrolní otázky a příklady

Příloha 2. Otázky pro písemnou zkoušku z předmětu Algoritmy

Příloha 3. Kapitola 5 studijní opory předmětu Algoritmy, modifikovaná pro jazyk C

Příloha 4. CD obsahující:

- Technická zpráva
- Kapitola 5 studijní opory předmětu Algoritmy, modifikovaná pro jazyk C
- Program pro ladění a testování algoritmů ze studijní opory
- Zdrojové texty animačního programu, přeložený binární soubor
- Uživatelská příručka k animačnímu programu

Příloha 1 - Kontrolní otázky a příklady

1. Vysvětlete pojem „stabilita řadicí metody“. Kdy je tato vlastnost významná?

2. Od algoritmu Bubble sort je odvozeno mnoho podobných metod, snažících se o zvýšení efektivity. Vysvětlete, v čem spočívá jejich zlepšení a proč nebude vzrůst efektivity výrazný.

3. Studijní opora předkládá variantu Heap sortu, která potřebuje, aby indexy řazeného pole byly v rozsahu 1..N. Upravte tento algoritmus tak, aby zpracovával pole na indexech typických pro jazyk C.

4. ListMerge sort v řazeném poli vyhledá neklesající posloupnosti, které pak setřiďuje. Jaký abstraktní datový typ potřebuje pro uložení začátků těchto posloupností?

5. Radix sort využívá principu řazení podle více klíčů. Jaká metoda je pro toto řazení v Radix sortu použita?

Příloha 2 - Otázky pro písemnou zkoušku z předmětu Algoritmy

Otázky jsou určeny pro formulářově orientovanou písemnou zkoušku, na výběr jsou vždy 4 možnosti, mezi nimiž může být 0..N správných odpovědí. Ty jsou zvýrazněny.

1. Které z těchto algoritmů pracují principiálně bez přesunu položek?

a) Shell sort
b) Merge sort
c) ListMerge sort
d) Radix sort

2. Třípásková přirozená metoda střídavě opakuje fáze

a) rovnoměrné distribuce a setřiďování
b) rozdělování a setřiďování
c) vyhledávání a vkládání
d) rovnoměrné distribuce a vkládání

3. Jak bude vypadat rekurzivní volání v algoritmu Quick sort?

```
void quickSort(int left, int right){ //left,right jsou hranice pole
   int i,j; //i je index jdoucí z levého okraje, j jde zprava
   partition(left,right,&i,&j);
   ...
   ...
}
```

```
a) if (i<j) quickSort(left,i);
   else quickSort(j,right);
```

b) if (left<j) quickSort(left,j);
if (i<right) quickSort(i,right);

c) if (left<i) quickSort(left,j);
if (right>j) quickSort(i,right);

d) if (left<j) quickSort(left,i);
if (right>i) quickSort(j,right);

4. Desetiprvkové pole: 102, 152, 77, 200, 52, 9, 27, 95, 27, 18 bude řazeno Quick sortem.
Jak bude pole vypadat těsně po skončení prvního průběhu funkce `partition`? Řazené pole `array` je globálním objektem.

```
void partition(int left,int right, int* i, int* j ){
    int PM;  //pseudomedián
    *i=left; //inicializace i
    *j=right;//inicializace j
    int x;   //pomocná proměnná
    PM = array[(*i+*j)/2]; //ustavení pseudomediánu
    do{
      while (array[*i]<PM) (*i)++;
      while (PM < array[*j]) (*j)--;

      if (*i<=*j){ //výměna nalezených prvků
        x=array[*i];
        array[*i]=array[*j];
        array[*j]=x;
        (*i)++; //posun indexů
        (*j)--;
      }
    }while (*i<=*j); //cyklus končí, když se indexy i a j překříží
}
```

a) 9, 152, 77, 200, 52, 102, 27, 95, 27, 18
b) 9, 18, 27, 27, 52, 200, 77, 95, 152, 102
c) 18, 27, 77, 27, 9, 52, 200, 95, 152, 102

c) 18, 27, 27, 9, 52, 200, 77, 95, 152, 102

5. Jaké jsou vlastnosti tohoto řadicího algoritmu? Řazené pole `array` je globálním objektem.

```
void sort(int N){ //N je index posledního prvku
  int i,j,tmp;
  for(i=1;i<=N;i++){
    tmp=array[i];
    j=i-1;
    while ((tmp<array[j])&&(j>=0)){//hledej místo a posouvej prvek
      array[j+1]=array[j];
      j--;
    }
    array[j+1]=tmp; //konečné vložení na místo
  }
}
```

a) Metoda je stabilní, chová se přirozeně

b) Metoda je nestabilní, chová se přirozeně

c) Metoda je stabilní, chová se nepřirozeně

d) Metoda je nestabilní, chová se nepřirozeně

Příloha 3 – Kapitola 5 studijní opory předmětu Algoritmy, modifikovaná pro jazyk C

5 ŘAZENÍ

Cíle kapitoly

Kapitolou "řazení" vrcholí témata 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 připočíst cca 6 až 8 hodin.

5.1 Úvod

5.1 Úvodní pojmy a principy

Terminologie

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 zadanou 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 předpočítačové éře, kdy se mechanizované řazení na děrných štítcích provádělo postupným tříděním na mechanických třídicích strojích. Tyto stroje roztřídily (rozdělily) soubor štítků na 10 podsouborů podle hodnoty 0-9 vyděrované v zadaném sloupci. Operátor seřadil ručně tyto podsoubory za sebe do jednoho souboru a třídil je podle dalšího sloupce. Třídil-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 metoda "radix-sort" ("řazení tříděním"). Algoritmy používané později pro řazení na počítačích nevyužívaly princip třídění (*sorting*), ale pojem "sort" - "třídění" jim v názvu zůstal. V anglických názvech se kmen "*sort*" zachovává a v anglické terminologii se s pojmem "*sorting*" pro řazení budeme setkávat. V české terminoilogii se přidržíme termínu "řazení". Ostatně, ani v tělocviku neříkáme "setřiďte 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.

x+y

Přirozenost

Přirozenost řazení je vlastnost algoritmu řazení, jehož *doba* řazení sekvence již uspořádané vzestupně podle hodnoty daného klíče řazena *je menší*, *než doba* řazení náhodně uspořádané sekvence a ta *je menší, než doba* řazení sekvence již seřazené v opačném pořadí hodnoty klíčů.

Smysl řazení

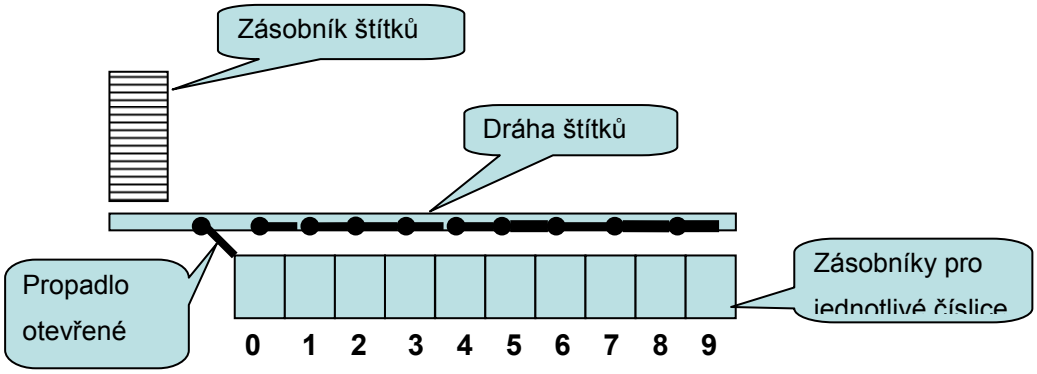
Jako implicitní budeme považovat seřazení podle vzestupného uspořádání klíčů (od nejmenšího k největšímu).

Experimentální hodnoty

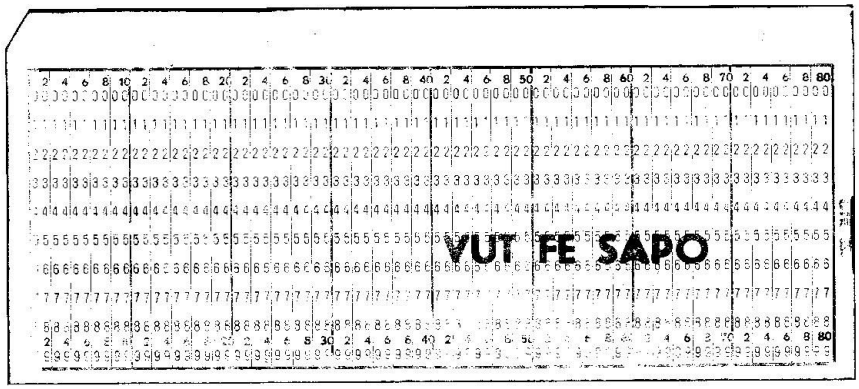
V této kapitole jsou u některých metod uvedeny **experimentální hodnoty**, získané v diplomní práci studenta našeho oboru. Jejich význam slouží jen k relativnímu porovnání metod.

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.

Problém lez řešit třemi způsoby:

Složená relace

a) Vytvoření složené relace uspořádání:

```
//Vrací 0, pokud je druhý starší nebo jsou stejne staří. Jinak vrací 1.
int prvniStarsi(TDatumNarozeni prvni,
                TDatumNarozeni druhy){
  if (prvni.rok!=druhy.rok)
    return (prvni.rok<druhy.rok);
  if (prvni.mesic!=druhy.mesic)
    return (prvni.mesic<druhy.mesic); //rok je shodný
  return (prvni.den<druhy.den); //rok i měsíc shodný
}
```

Pozn. Výsledkem relačních operátorů v jazyce C je typ **int**, který se běžně tímto způsobem používá místo typu **bool**. Proto se u funkcí, které vracejí boolovské hodnoty, budeme držet této konvence a použijeme typ int.

Postupné řazení podle jednoho klíče

b) Neuspořádanou množinu položek lze řadit postupně podle vzrůstající priority jednotlivých klíčů. Podmínkou je použití stabilní řadicí metody! Příklad: Skupinu osob lze seřadit podle stáří tak, ze se:

1. Napřed seřadí podle dne data narozeni
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 řetězec. 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. 84**54**015471).

Příklady k procvičení

a) Napište funkci, která ze zadaného pole osob vytvoří seřazený seznam podle narozenin v roce. Při shodném datu narozenin má starší přednost.

```
typedef struct tDatum {
  int rok;
  int mesic;
  int den;
}TDatumNarozeni;

typedef struct tOsoba {
  char* jmeno;
  TDatumNarozeni datumNarozeni;
}TOsoba;
```

Pozn. Seznam osob seřazený "podle narozenin" je takový seznam, v němž jsou postupně osoby tak, jak slaví v běžném roce narozeniny. Má-li narozeniny více osob v témže dni, mají starší osoby "přednost".

b) Je dán typ

```
typedef enum tObor{
infsys,intsys,pocsys,grasys}TObor;

typedef struct tStudent{
  char* jmeno;
  TObor obor;
```

```
  int rocnik;
  double prumer;
} TStudent;
```

Pozn. Výčtový typ enum se v jazyce C nepoužívá pro vytváření nových typů. Je to sice možné, ale nemá to smysl, protože do proměnné vzniklého typu je možné přiřadit libovolnou hodnotu typu int. V příkladu je takto enum použit pro zachování podobnosti s původní oporou.

Vytvořte aglomerovaný (integrovaný) 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.

Nápověda: Aglomerovaný klíč bude typu řetězec. Průměr lze převést na typ int např. 2.75 ⇨ 275. Výsledný řetězec omezte na 20 znaků.

c) Napište funkci libovolného algoritmu řazení pole, který znáte z prvního ročníku tak, aby se při jejím volání jedním vhodným parametrem ovládala složka, která bude klíčem řazení. Nechť pole je pole prvků typu TOsoba. Pak funkce:

```
void razeni(TPole pole, TXX xx);
```

bude řadit jednou podle složky rok, jindy podle složky mesic 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

```
int prvniStarsi (char* RC1, char* RC2);
```

e) Napište funkci

```
int maDrivNarozeniny (char* RC1,char* RC2);
```

kde RC1 a RC2 jsou rodná čísla. V případě stejně starých osob nebo stejných narozenin má přednost žena před mužem. V případě rovnosti u stejného pohlaví rozhoduje pořadové číslo rodného 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 vytvořím pomocné pole pořadí (tzv. pořadník). Inicializuje se hodnotami shodnými s indexem. Výsledkem řazení je pořadník, v němž jsou uspořádány (seřazeny) indexy prvků řazeného pole.

Nechť jsou dány typy:

```
typedef struct tPolozka{
  TData data;
  TKlic klic;
}TPolozka;

typedef TPolozka TPole[N];
typedef int TPorad[N];
```

Proměnné těchto typů:

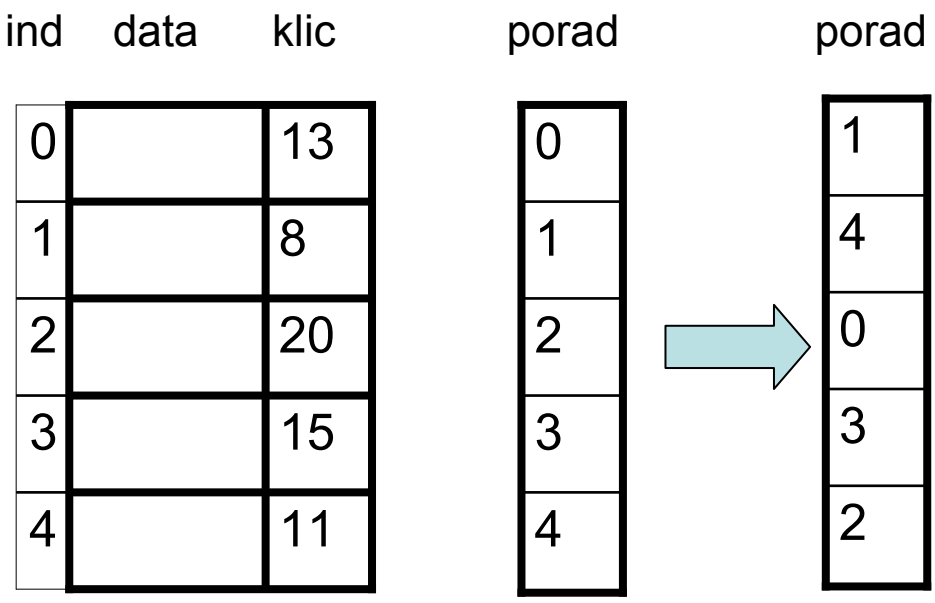
```
TPole pole;
TPorad porad;
```

Pak inicializace má tvar:

```
for(int i=0; i<N; i++) porad[i]=i;
```

Následující obrázek znázorňuje stav pořadníku po inicializaci a po seřazení.

x+y



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]
```

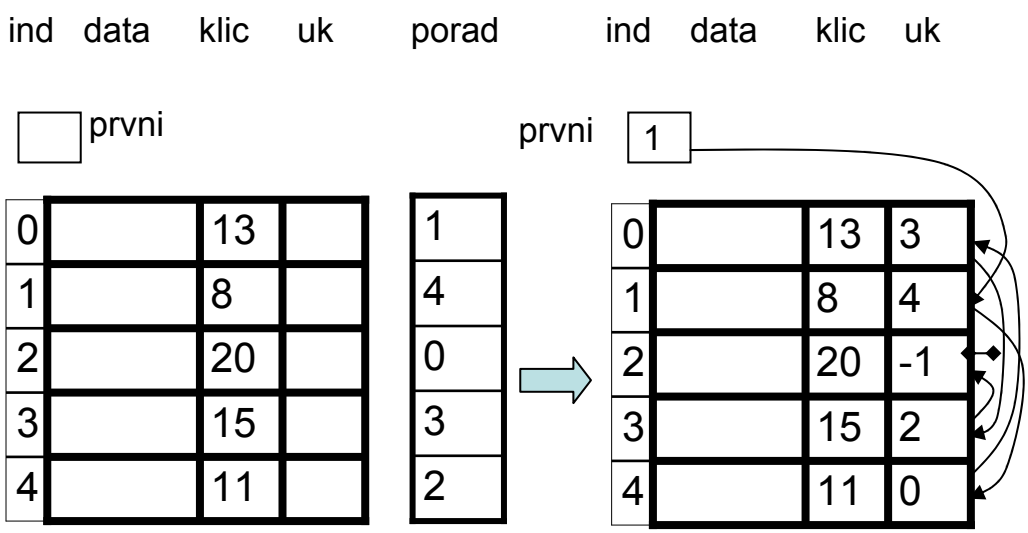
Pole seřazené bez výměny položek lze průchodem vložit do výstupního seřazeného pole vystPole cyklem:

```
for(int i=0;i<N;i++) vystPole[i]=pole[porad[i]];
```

Pole seřazené pomocí „pořadníku“ lze také zřetězit a vytvořit seřazený seznam. Následující obrázek znázorňuje výsledek zřetězení pole seřazeného pomocí pořadníku

z předcházejícího příkladu. Hodnota NULLového ukazatele je reprezentována hodnotou -1.

x+y



Zřetězení provede úsek programu:

```
int prvni=porad[0];
  for(int i=0; i<N; i++)
    pole[porad[i]].uk = porad[i+1];
pole[porad[N-1]].uk = -1;//záporná hodnota ve funkci NULL
```

Zřetězenou seřazenou posloupnost lze převést ze zdrojového pole do cílového pomocí jednoduchého cyklu, který je vhodným příkladem pro domácí procvičení.

MacLarenův algoritmus

MacLarenův algoritmus uspořádá pole seřazené bez přesunu na místě samém (lat. *in situ*) - tedy proces bez pomocného pole.

3

```
int i = 0;
int pom = prvni;
while(i<N-1){
     while(pom<i) pom = pole[pom].uk;
     //Hledání následníka přesunutého na pozici větší než i
     pole[i] :=: pole[pom]; //výměna akt. prvního s akt. minimálním
     pole[i].uk :=: pom; // stejná výměna ukazatelů
```

```
        i++;    // prvních i-1 prvků je již na svém místě
}
```

Pozn. Komentář k MacLarenovu algoritmu.

První prvek seznamu (na který ukazuje proměnná prvni) se vymění s prvkem pole na indexu 0. Tím se nejmenší položka dostane na své místo. Na prvek, který byl z nultého indexu pole odsunut jinam však některý prvek ukazoval. Je třeba ho najít a změnit jeho ukazatel tak, aby místo na nulté 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 (N-2) prvek, kdy cyklus končí.

S ohledem na velkou délku položky je součet času řadicího algoritmu bez přesunu položek (minimálně linearitmický) s časem McLarenova algoritmu (lineární) kratší, než čas samotného řadicího 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:

- Princip výběru (*selection*) - přesouvají maximum/minimum do výstupní posloupnosti
- Princip vkládání (*insertion*) – vkládají postupně prvky do seřazené výst. posloupnosti
- Princip rozdělování (*partition*) – rozdělují postupně množinu prvků na dvě podmnožiny tak, že prvky jedné jsou menší než prvky druhé
- Princip slučování (*merging*) setřiďují postupně seřazené dvě podmnožiny do jedné
- Jiné principy ...

Zavedené konvence

Zavedené konvence

V následujících partiích budou metody řazení v polích vykládány na silně zjednodušené struktuře množiny dat, implementované polem s jednosložkovými položkami, představovanými klíčem typu int:

```
typedef int TA[POCET]; /* Typ řazeného pole */
```

```
TA a; /* Řazené pole */
```

Toto pole bude vstup/výstupním parametrem procedury řazení nebo jejím globálním objektem.

5.2 Řazení na principu výběru

5.2 Řazení na principu výběru (*Select sort*)

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:

```
for (int i=0; i<=N; i++){
  /* najdi minimální prvek pole mezi indexy i a N. Polohu (index) minim a ulož do pomocné proměnné pInd */
  a[i] :=: a[pInd];
}
```

N neoznačuje počet prvků, ale **index posledního prvku**. Toho se budou držet i všechny další uvedené algoritmy, pokud nebude uvedeno jinak.

x+y

Algoritmus má tvar:

```
void selectSort(TA a, int N){
  int pMin,pInd;
  for (int i=0; i<N; i++){
    pInd=i; //poloha pomocného minima
    pMin=a[i]; //pomocné minimum
    for (int j=i+1;j<=N;j++){
      if (pMin>a[j]){
        pMin = a[j];
        pInd = j;
      }
    }
    a[i] :=: a[pInd];
  }
}
```

Hodnocení metody

$\sum$

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

```
void bubbleSort(TA a, int N){
  int i=1;
  int konec;
  do{
    konec = 1; //booleovská hodnota TRUE
    for (int j=N; j>=i; j--){
      if (a[j-1]>a[j]){   //porovnání sousedních dvojic
        a[j] :=: a[j-1]; //výměna
        konec = 0; //booleovská hodnota FALSE
      }
    }
    i++;
  }while ((!konec)&&(i<=N));
}
```

x+y

Jiná varianta zápisu algoritmu Bubble sort.

```
void bubbleSelect(TA a, int N){
  int pokracuj=1;
  int pomN=N;
  while ((pokracuj)&&(pomN>0)){
    pokracuj = 0;
    for(int i=0; i<pomN; i++){ //cyklus porovnávání dvojic
      if(a[i+1]<a[i]){
        a[i] :=: a[i+1];
        pokracuj = 1; //došlo k výměně – nelze skončit
      }
    }
    pomN--;
  }
}
```

∑

Hodnocení metody:

- Bublinový 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 seřazené!**
- 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 přinášejí zajímavé programovací náměty, ale z pohledu zlepšení nemají výrazný účinek.

- **Ripple-sort** si pamatuje polohu první výměny a je-li větší než 0 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 – heapu je binární hromada, založená na binárním stromu.

Významnou operací nad hromadou je její **rekonstrukce** poté, co se poruší pravidlo hromady v jednom uzlu.

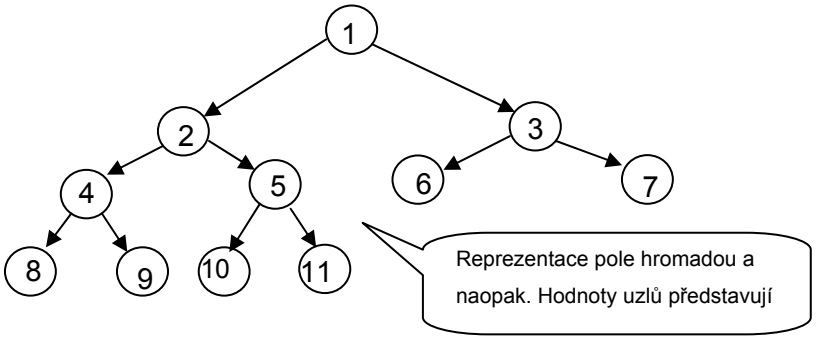
Nejvýznamnějším případem je **porušení hromady v kořeni**. Operaci, která znovuustaví hromadu porušenou v kořeni říkáme „**sift**“ (prosetí), nebo také „zatřesení hromadou“. Spočívá v tom, že prvek z kořene postupnými výměnami propadne na „své“ místo a do kořene se dostane prvek, splňující pravidla hromady. 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 odebírá 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 linearitmickou složitostí $n*\log_2 n$ **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

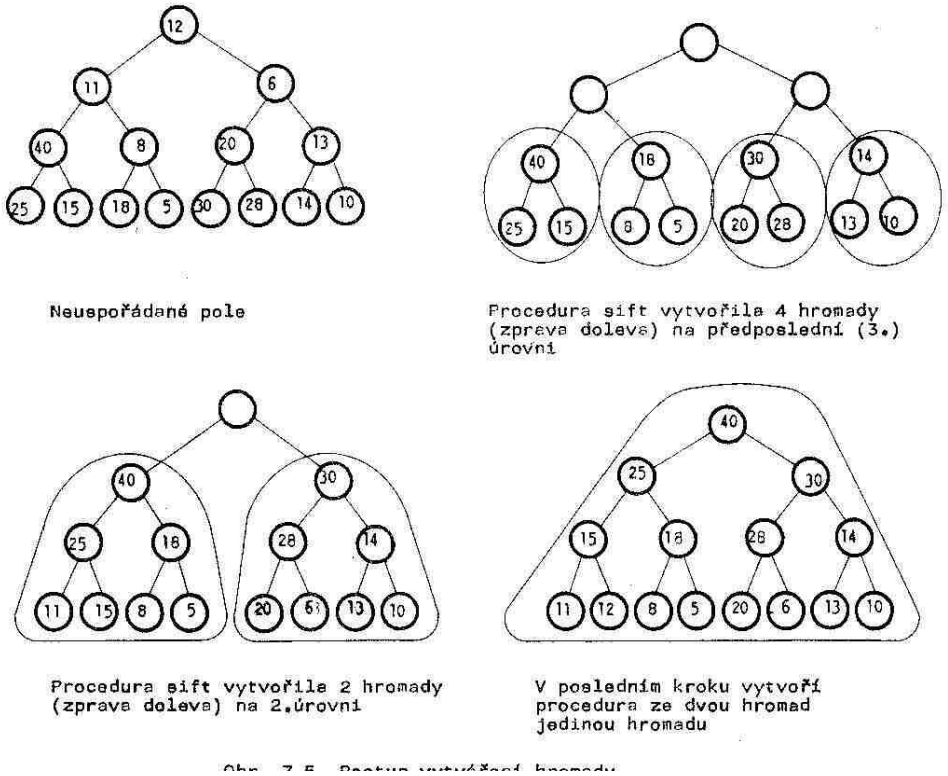
1	2	3	4	5	6	7	8	9	10	11



Mějme proceduru SiftDown, která znovuustaví hromadu porušenou v kořeni - proseje prvek v kořeni, který jako jediný porušuje 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 „otcovský uzel“ a postupuje se ustavováním hromad po všech uzlech doleva a nahoru až k hlavnímu kořeni, jak to ilustroval předcházející obrázek. Má-li pole N prvků, pak nejnižší a nejpravější uzel odpovídající hromady má index **[N/2]**. (Uvažujeme rozsah indexů pole 1..N).

```
/*Ustavení hromady*/
  int left=N/2; // index nejpravějšího uzlu na nejnižší úrovni
  int right=N;
  for(int i=left;i>=1;i--) SiftDown(a,i,right);
/* cyklus opakovaného prosetí "porušeného" kořene do dvou podstromů, které již mají
vlastnosti hromady */
```

Celý algoritmus Heapsortu má pak tvar:

```
void heapSort(TA a, int N){
  TA b; //pomocné pole – viz. Pozn.
  for (int i=0;i<=N;i++) b[i+1]=a[i];//potřebuji pole 1..N+1!
  int Nb = N+1;
 /*Ustavení hromady*/
  int left=Nb/2; // index nejpravějšího uzlu na nejnižší úrovni
  int right=Nb;
  for(int i=left;i>=1;i--) siftDown(b,i,right);
 /*Vlastní cyklus Heap-sortu*/
  for(right=Nb;right>=2;right--){
    b[1] :=: b[right]; // Výměna kořene s akt. posledním prvkem
    siftDown(b,1,right-1); //Znovuustavení hromady
  }
  for (int i=0;i<=N;i++) a[i] = b[i+1];
  //výsledek řazení je třeba vrátit opět v poli a (indexy 0..N)
}
```

Pozn. Protože index synovského uzlu se počítá s pomocí násobení indexu otcovského uzlu, nemůže být žádný prvek na indexu 0. Proto je nejdřív vytvořeno pole b, do kterého se pole a „přeskládá“ na indexy 1 až N+1. Algoritmus je v této podobě použitelný pouze pro výukové účely!

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.

```
void siftDown(TA b, int  left, int right){
 //left je kořenový uzel porušující pravidla heapu, right je velikost pole
  int i = left;
  int j = 2*i; //index levého syna
  int tmp = b[i];
  int cont = (j<=right);
  while (cont){
```

```
      if ((j<right) //uzel má oba synovské uzly
        &&(b[j]<b[j+1])) //a pravý syn je větší
          j++; //nastav jako většího z dvojice synů

      if(tmp>=b[j])//prvek tmp již byl posunut na své místo; cyklus končí
        cont = FALSE;
      else{ //tmp propadá níž, A[j] vyplouvá o úroveň výš
        b[i]=b[j];
        i=j; //syn se stane otcem pro příští cyklus
        j=2*i; //příští levý syn
        cont = (j<=right); //podmínka: "j není terminální uzel"
      }
    }
    b[i]=tmp; //konečné umístění prosetého uzlu
}
```

Závěr

- Heapsort je řadicí metoda s **linearitmickou složitostí**, protože „siftDown“ umí rekonstruovat hromadu (najít extrém mezi N prvky) 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 opakovaně 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 (na indexu 0). Pak má řazení strukturu:

```
for (int i=1; i<=N; i++){
      /* 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 */
}
```

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.

```
void bubbleInsertSort(TA a, int N){
  int i,j,tmp;

  for(i=1;i<=N;i++){
    tmp=a[i];
    j=i-1;
    while ((tmp<a[j])&&(j>=0)){ //hledej místo a posouvej prvek
      a[j+1]=a[j];
      j--;
    }
    a[j+1]=tmp; //konečné vložení na místo
  }
}
```

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.

```
void binaryInsertSort(TA a, int N) {
  int i,j,m,left,right,tmp;
  for(i=1;i<=N;i++){
    tmp=a[i];
    left=0; right=i-1; //nastavení levého a pravého indexu
    while (left<=right){ //stand. bin. vyhledávání
      m=(left+right)/2;
      if (tmp<a[m]) right=m-1;
      else left=m+1;
    }
    for(j=i-1;j>=left;j--)
      a[j+1]=a[j]; //posun segmentu pole doprava
    a[left]=tmp;
  }
}
```

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+1 prvcích typu int v poli H. Napište proceduru, která rekonstruuje (znovuustaví) hromadu porušenou zápisem libovolné hodnoty na index 1<=K<=N

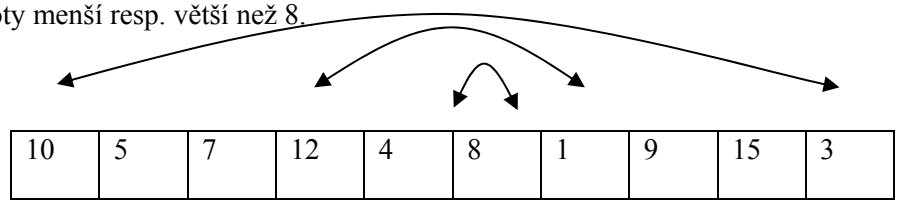
5.4 Quick sort

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 přeskupí 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.

```
void partition(TA a, int left, int right,
               int* i, int* j);
```

Tato procedura rozdělí následující pole na dvě části, **left..j** a **i..right**, které obsahují hodnoty menší resp. větší než 8.



10	5	7	12	4	8	1	9	15	3

Pole po rozdělení

3	5	7	1	4	8	12	9	15	10

left j i right

Algoritmus Quick Sort

Máme-li k dispozici proceduru partition, pak má rekurzívní zápis mechanismu řazení Quick Sort tvar:

```
void quickSort(TA a, int left, int right){
  //Při prvním volání má left hodnotu 0 a right hodnotu N
   int i,j;
   partition(a,left,right,&i,&j);
   if (left<j) quickSort(a,left,j); //Rekurze doleva
   if (i<right) quickSort(a,i,right); //Rek. doprava
}
```

Tajemství vysoké rychlosti řazení Quick sort 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:

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.

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"

```
void partition(TA a, int left, int right,
                                    int* i, int* j){
     int PM;   //pseudomedián
     *i=left; //inicializace i
     *j=right;//         j

     PM = a[(*i+*j)/2]; //ustavení pseudomediánu

     do{
       while (a[*i]<PM) (*i)++;     //hledání prvního zleva
       while (PM < a[*j]) (*j)--; //              zprava

       if (*i<=*j){
         a[*i] :=: a[*j]; //výměna nalezených prvků
```

```
        (*i)++;
        (*j)--;
      }
    }while (*i<=*j); //cyklus končí, když se indexy i a j překříží
}
```

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.

```
void nonRecQuicksort(TA a, int left, int right){
  int i,j;
  TStack S; //deklarace ukazatele na ADT zásobník
  S = (TStack)malloc (sizeof(struct stack));
  stackInit(S); //inicializace zásobníku
  stackPush(S,left);   //vložení levého indexu prvního segmentu
  stackPush(S,right); //vložení pravého indexu prvního segmentu
  while (!stackEmpty(S)){ //vnější cyklus
    //čtení zásobníku v obráceném pořadí
    right = stackTop(S);stackPop(S);
    left = stackTop(S);stackPop(S);
    while (left<right){ //vnitřní cyklus – je co dělit
      partition(a,left,right,&i,&j);
      stackPush(S,i); //uložení intervalu pravé části do zásobníku
      stackPush(S,right);
      right = j; //příprava pravého indexu pro další cyklus
```

```
        }
    }
}
```

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 $T_a(n)\sim=17*n*lg_2(n)$
časová složitost již seřazeného pole je:
$T_{usp}(n)\approx 9*n*lg_2(n)$
časová složitost pro inverzně seřazené pole je:
$T_{inv}(n)\sim=9*n*lg_2(n)$

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 uchová 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_2n 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 rychlostí 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

x+y

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 div 2) a v první etapě docházelo k výměně (n 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.

```
void shellSort(TA a, int N){
    int step,i,j;
    step = N/2; //první krok je polovina délky pole
    while (step > 0){
```

```
        for (i=step;i<=N;i++){ //cykly pro paralelní n-tice
          j = i-step;
          while ((j>=0)&&(a[j]>a[j+step])){
            //bublinový průchod
            a[j] :=: a[j+step];
            j = j-step; //snížení indexu o krok
          }
        }
        step = step/2; //půlení kroku
      }
}
```

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 rekurzi 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 setřiďováním

5.6 Řazení setřiďováním 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, ze karta s nejnižším číslem je nahoře. 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 setřiďujícího dva seznamy do jednoho výsledného seznamu ukazuje následující příklad. Výsledný seznam využívá hlavičky, která je v závěru zrušena.


```
TList MergeLists(TList L1, TList L2){
  TList L3,tmp;
  L3 = newList(); //vytvoření hlavičky
  tmp = L3;
  while ((L1!=NULL)&&(L2!=NULL)){ //cyklus slučování
    if (L1->data < L2->data){ //"Odložení" prvku z L1
      tmp->ptr=L1;
      tmp=L1;
      L1=L1->ptr;
    }else{ //"Odložení" prvku z L2
      tmp->ptr=L2;
      tmp=L2;
      L2=L2->ptr;
    }
  }
  //připojení neprázdného seznamu  L1 nebo L2
  if (L1==NULL) tmp->ptr=L2;
  else tmp->ptr=L1;
  //Zrušení hlavičky
  tmp=L3;
  L3=L3->ptr;
  free(tmp);
  return L3;
}
```

1) Napište úsek programu (funkci), která setřídí prvky typu int dvou seřazených polí do výsledného pole, jehož velikost je dostatečná (větší nebo rovna součtu velikostí zdrojových polí).

2) Navrhněte algoritmus pro vícenásobné setřiďování.

2A) Je dáno více jednorozměrných polí seřazených čísel typu int (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 int. Vytvořte z nich jeden cílový seznam seřazených čísel.

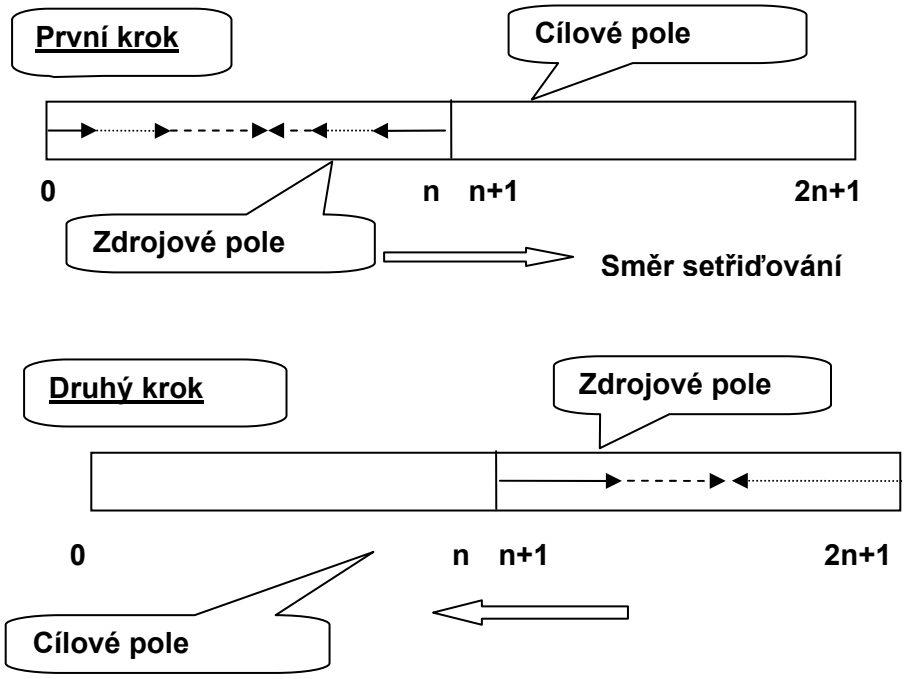
3) Algoritmy příkladu 2 vyžadují nalezení extrému (např. minimálního prvku) z n

prvních prvků nevyčerpaných sekvencí. Navrhněte variantu algoritmu pro příklad 2B, která pro opakované hledání minima použije hromadu (heap).

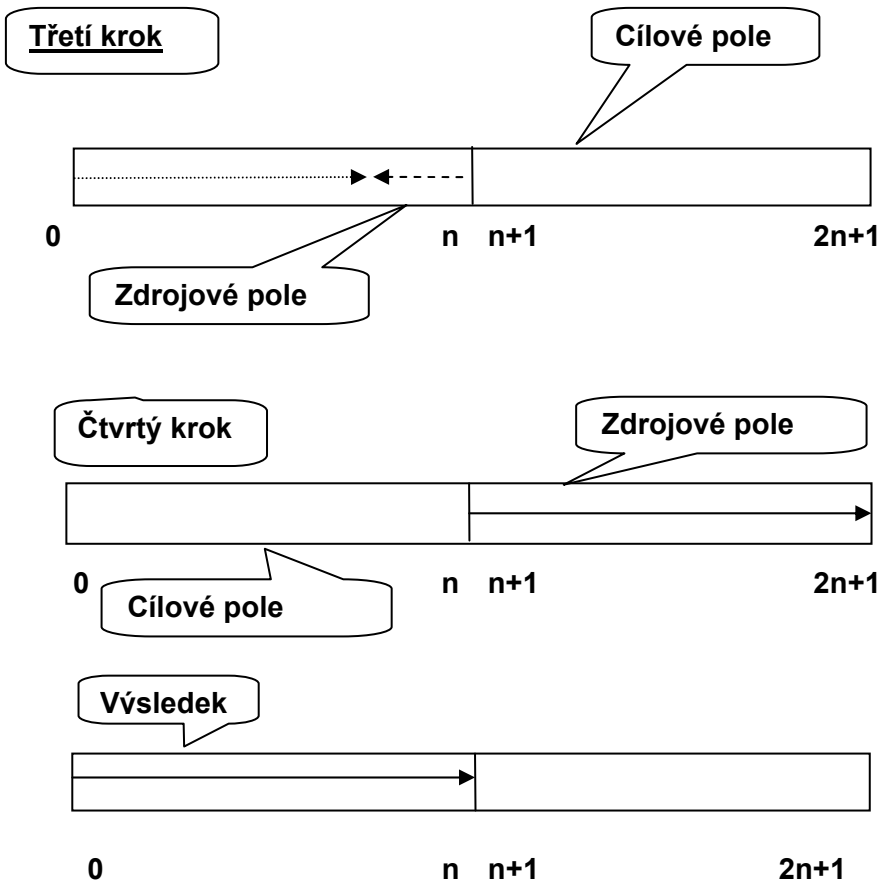
5.7 Megre-Sort

5.7. Merge-Sort je sekvenční metoda využívající přímý přístup k prvkům pole. Postupuje polem zleva a současně zprava a setřiďuje dvě „proti sobě“ postupující neklesající posloupnosti. Výsledek se ukládá do cílového pole a počet vzniklých posloupností se počítá v počitadle. 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



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.

Σ

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. ListMergeSort

5.8. List Merge Sort - řazení polí setřiďováním seznamů - je metoda řazení založená na 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 -1 (NULL). V následujícím obrázku je pět neklesajících posloupností. Čtvrtá z nich má délku rovnu jedné.

x+y

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
A	2	5	9	3	7	10	16	5	8	6	3	11	18	20
Pom	1	2	-1	4	5	6	-1	8	-1	-1	11	12	13	-1

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/IAL/private/Texty/Skripta/ch7.pdf na str.187

textu a str. 28/44 souboru pdf) jen jednu frontu a je tudíž jednodušší.

Algoritmus ListMergeSortu lze v algoritmickým pseudojazykem popsat takto:

```
TList L = initList();   // Inicializace fronty začátků seznamů
```

(* 1 *) Při průchodu polem řazených prvků zřetězíme prvky neklesajících posloupností pomocí pomocného pole indexů (ukazatelů). Přitom index (ukazatel) začátku každé neklesající posloupnosti vložíme do seznamu L. Index posledního prvku každé posloupnosti má hodnotu -1 (NULL). Výsledkem tohoto bloku jsou zřetězené seznamy a seznam L naplněný jejich začátky.

```
do{ // Předpokládá se pole o nenulové délce - alespoň jeden seznam
    copyFirst(L,&zac1); // Čtení začátku prvního seznamu
    deleteFirst(L); first(L);
    if (active(L)){//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])
        insertLast(L,zac1);//Uložení výsledného začátku do fronty
      else insertLast(L,zac2);
```

(* 2 *) Setřiď seznamy zac1 a zac2; koncový index nastav na -1.

```
  }
}while (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 (proceduru), 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 při setřiďování se u shodných prvků musí do výstupní

posloupnosti vložit prvek první posloupnosti a začátek vytvořené neklesající posloupnosti se vloží na začátek seznamu. Experimentálně byly naměřeny hodnoty uvedené v následující tabulce.

n	256	512
SP	2	6
NUP	32	74
OSP	22	48

5.9. Radix sort

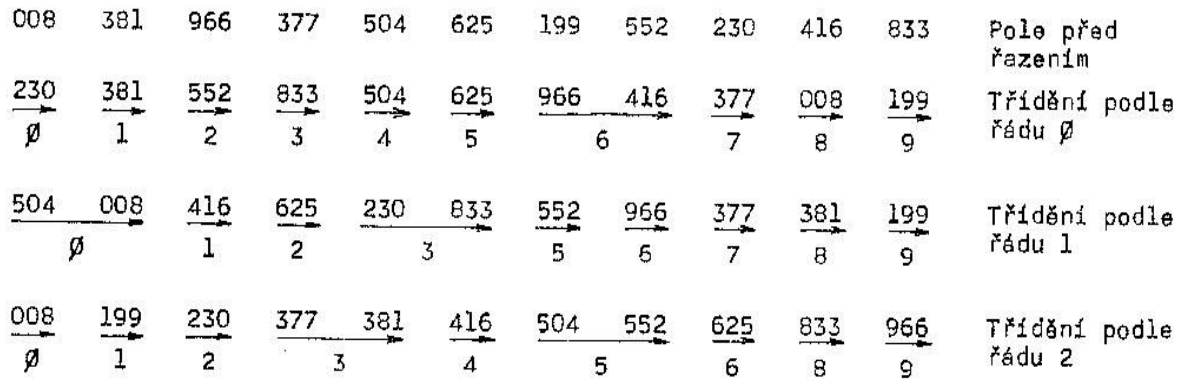
5.9. Radix-sort řazení tříděním podle základu - je počítačovou obdobou řazení tříděním na děrnoštítkových třídicích strojích.

Na třídicích strojích je základem desítková číslice na daném řádu 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 třímístných čísel. Na druhém řádku jsou trojmístná čísla seřazena podle hodnoty poslední číslice předcházejícího řádku. Metoda řazení musí být stabilní a proto je uspořádání předchozího řádku významné. Na třetím řádku jsou čísla seřazena podle pořadí předposlední číslice. Na posledním řádku jsou čísla seřazena podle hodnoty první číslice. Poslední řádek 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 řádu k nejvyššímu.

```
x+y
```

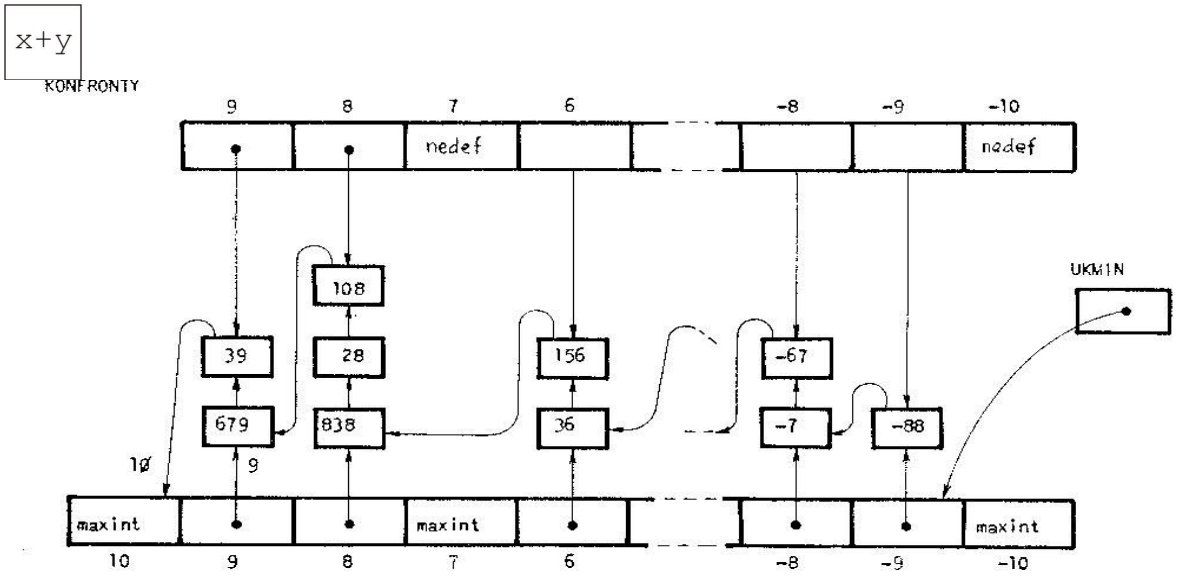


Obr. 7.10. Řazení trojciferných čísel podle základu 10

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 "přihrádky" (podobné těm u děrnoštítkového stroje) pro 20 různých číslic.

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.



Obr. 7.11. Datová struktura po sjednocení front vzniklých tříděním podle nejnižší číslice

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 číslice. 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 znovu 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:

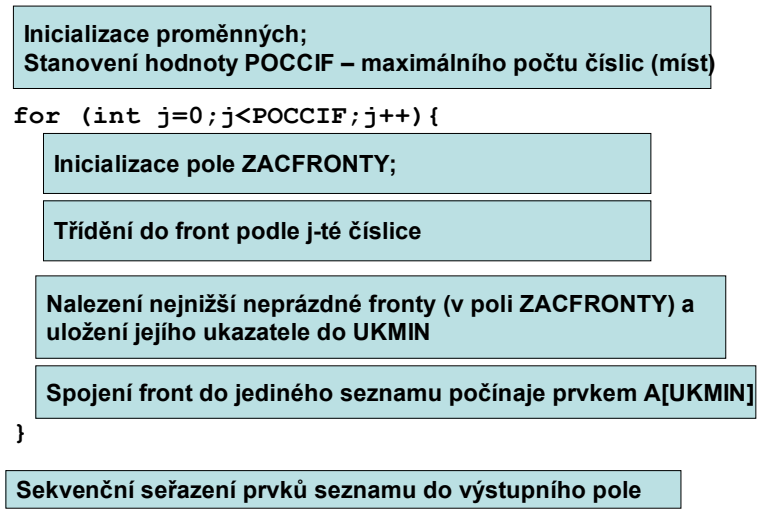
TMAX(n)=(42*POCCIF+15)* n +(16+34*ZAKLAD)POCCIF+15

což vyjadřuje **lineární složitost**! Teoreticky je to tedy nejrychlejší algoritmus.

Experimentálně byly naměřeny tyto hodnoty pro náhodně uspořádaném pole:

n	**256**	**512**	**1024**
NUP	**24**	**60**	**102**

Schéma algoritmu řazení RadixSort



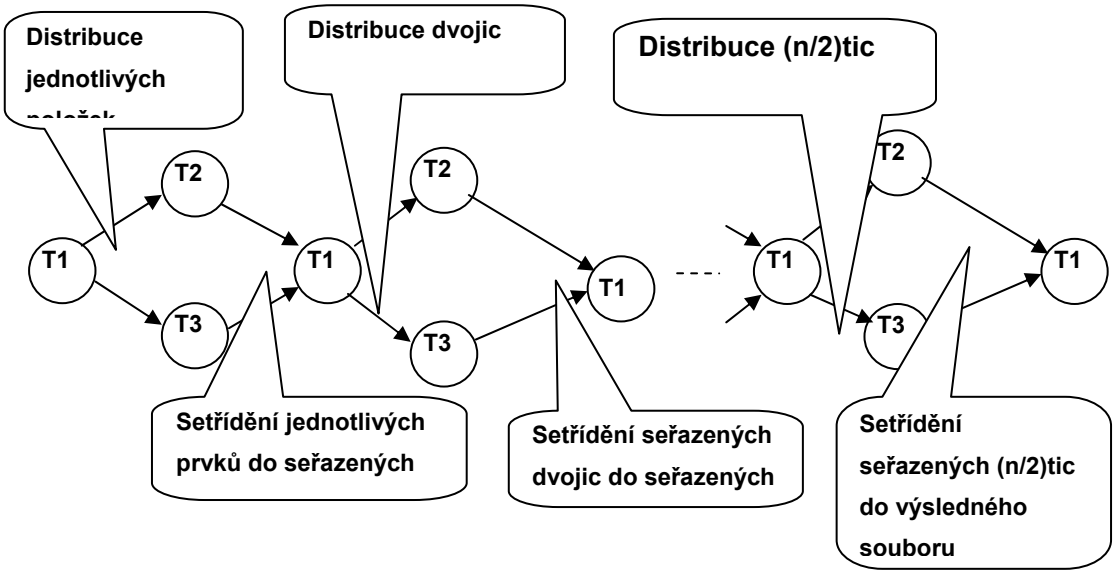
5.10 Sekvenční řazení

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 setřiďování souborů

Přímé setřiďování 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".

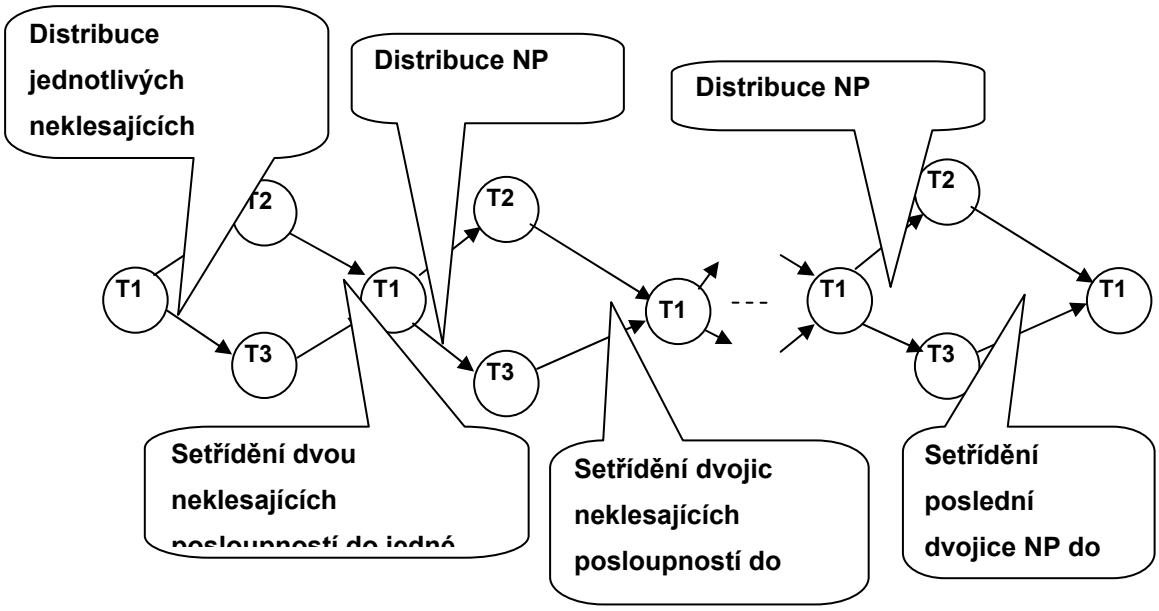
Třípásková přímá metoda



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 setřiďovací. V první fázi se prvky zdrojového souboru rozdělí - distribuují rovnoměrně do dvou soborů 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 distribuují 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 setřiďování do 2^{n+1} - tic tak dlouho, až s vznikne jedna seřazená posloupnost. Je zřejmé, že počet fází lze vyjádřit vztahem $\lceil \lg_2 n \rceil$ (kde závorky $\lceil \rceil$ vyjadřují úpravu hodnoty na nejbližší vyšší (nebo rovné) celé číslo), kde n je počet prvků seřazované posloupnosti.

Třípásková přirozená metoda



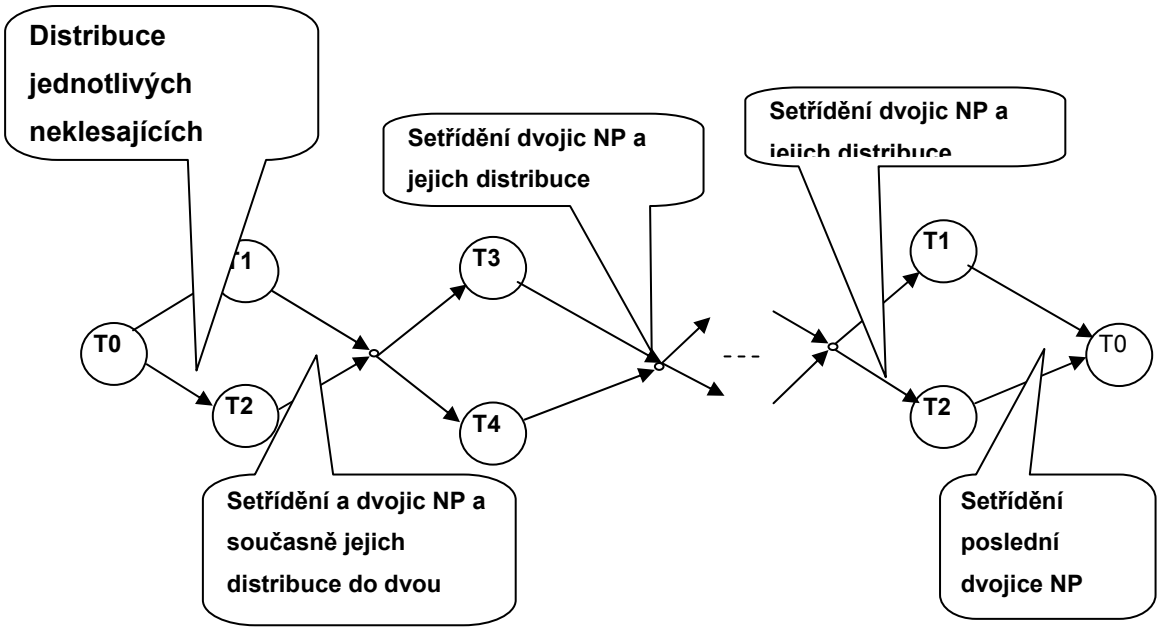
Zásadní rozdíl mezi třípáskovou přímou a přirozenou metodou spočívá v tom, že přirozená metoda distribuuje a setřiďuje neklesající posloupnosti zdrojových souborů. Z toho vyplývá, že hodnota počtu fází má podobný tvar, ale N je počet neklesajících posloupností ve zdrojovém souboru. Znamená to, že již 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 mnohacestného vyváženého setřiďování, 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 rychlost zpracování pro třídění, protože každou fázi redukuje o distribuční krok, který je u mechanických zařízení provázen potřebou převinutí (*rewind*).



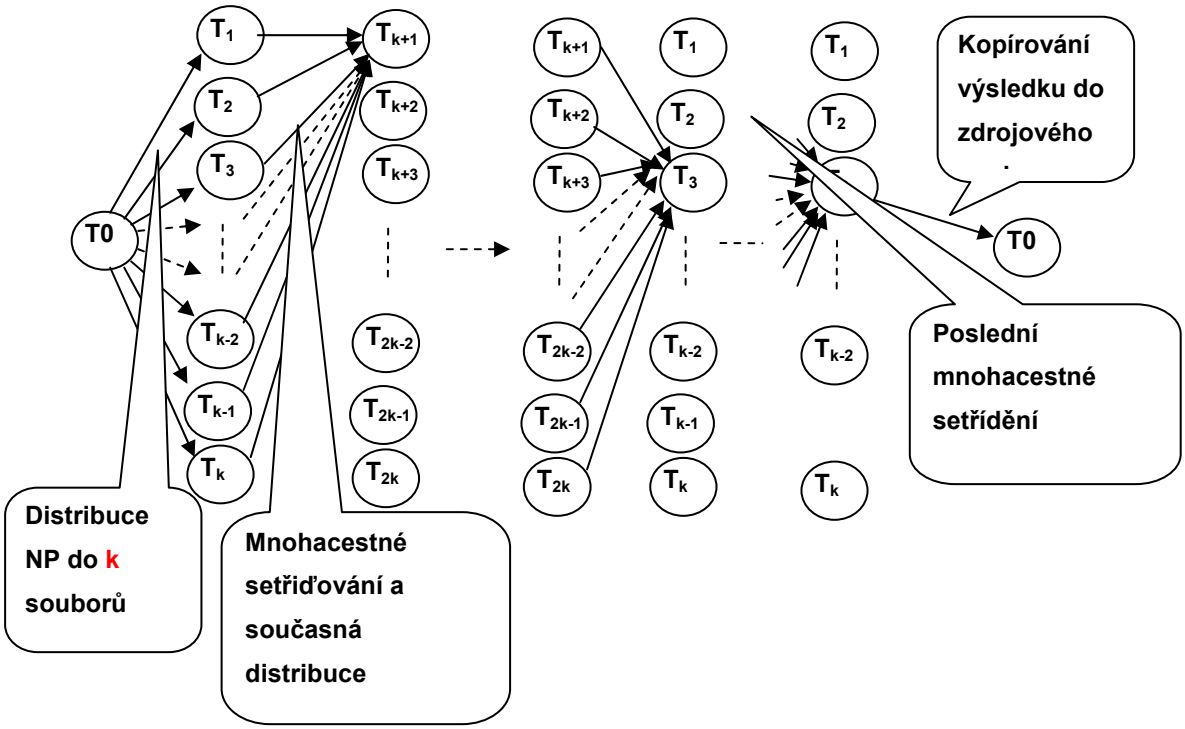
Po prvotní distribuci neklesajících posloupností do dvou pracovních souborů se provádí setříďování každé dvojice posloupností a jejich střídavé ukládání na další dvojici pracovních souborů.

Mnohacestné vyvážené setřiďování

Mnohacestné vyvážené setřiďování (*balanced multiway merging*) je jen rozšířením čtyřpáskové metodu na 2k souborů (k>2) s použitím mnohacestného setřiďování neklesajících posloupností (kde k je počet souborů na zdrojové i cílové straně systému.

Zatímco 4 pásková metoda slučovala dvojice neklesajících posloupností, mnohacestné seřazování, které používalo n zdrojových a n cílových souborů, setřiďovalo 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 setřiďování přebírá hodnotu základu číslo k, vyjadřující počet souborů na jedné straně. Pak je tedy počet fází dán vztahem $\lceil \log_k N \rceil$.

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 skříněmi magnetopáskových jednotek.



Mnohacestné vyvážené setřiďování 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 soborů implementovaných rozsáhlými seznamy.

Polyfázove 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 setřiďování má s použitím K+1 souborů řádově shodnou složitost jako mnohafázové setřiďování s 2K soubory. Vykazuje tedy „úsporu“ K-1 souborů (mechanik).

Metoda je založena na principu postupného setřiďování 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 setřiďování pokračuje tak dlouho, až se vytvoří jediný výsledný cílový seřazený soubor.

Potíží 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 posloupností. 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:

1,1,2,4,8,16,31,61,120,236,...

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	∑
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 posloupností je: 5 a 5+8 .

Pro soustavu 6 pásek (souborů), se budou neklesající posloupnosti z 5 zdrojových souborů setřiďovat do jednoho cílového souboru. Pro inicializační rozdělení se použije Fibonacciho posloupnost 4. řádu, která má 5 počátečních hodnot a má tvar:
0,0,0,0,1,1,2,4,8,16,31,61,120,236,...

f1	f2	f3	f4	f5	f6	∑
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 dáno 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 snižujícího 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.Ukázka 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 nejvýznamnějšími algoritmy vnitřního řazení (řazení polí) i sekvenčního (vnějšího) řazení (řazení souborů) a to v s rekurzívním i nerekurzívním zápisem tam, kde je to účelné.