VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PŘÍPRAVA DOMÁCÍCH ÚLOH PRO PŘEDMĚT ALGORITMY

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE JAKUB ŘEZÁČ
AUTHOR

BRNO 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PŘÍPRAVA DOMÁCÍCH ÚLOH PRO PŘEDMĚT ALGORITMY

PREPARATION OF HOMEWORKS FOR SUBJECT ALGORITHMS

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE — JAKUB ŘEZÁČ
AUTHOR

VEDOUCÍ PRÁCE — ING. BOHUSLAV KŘENA, Ph.D.
SUPERVISOR

BRNO 2007

Abstrakt

Práce se zaměřuje zejména na implementaci domácích úloh pro předmět Algoritmy, jež je vyučován v druhém ročníku na Fakultě informačních technologií Vysokého učení technického v Brně.
Pro vyhodnocování zmíněných domácích úloh je využit systém pro automatizované vyhodnocování domácích úloh. Součástí zadání této bakalářské práce je rovněž modifikování zmíněného systému.

Klíčová slova

Předmět Algoritmy, historie programovacích jazyků, jazyk C, abstraktní datový typ, domácí úlohy, systém pro automatické vyhodnocování domácích úloh, dvousměrně vázaný kruhový lineární seznam, zásobník, obecný graf, orientovaný graf.

Abstract

This bachelor’s thesis is oriented on an implementation of homework for a subject Algorithms, which is taught in second year-class on Faculty of information technology of University of technology in Brno. For homework’s correction is used system for automatized homework correction.
A modification of this system is part of this submission too.

Keywords

Subject Algorithms, history of programming languages, C language, abstract data type, homework, system for automatized homework correction, double bound circular linear list, stack, common graph, oriented graph.

Citace

Jakub Řezáč: Příprava domácích úloh pro předmět algoritmy, bakalářská práce, Brno, FIT VUT v Brně, 2007

Příprava domácích úloh pro předmět algoritmy

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Bohuslava Křeny, Ph. D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jakub Řezáč
25. dubna 2007

Poděkování

Děkuji zejména mému vedoucímu Ing. Bohuslavu Křenovi Ph. D. za trpělivost, výborné vedení a ochotu s jakou se mi při řešení věnoval. Dále bych rád poděkoval všem, kteří byli ochotni přispět mi svým názorem či nápadem nebo jen jinak podpořili mé snažení.

© Jakub Řezáč, 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

1 Úvod 8
2 Operační systémy 10
2.1 Operační systém Unix 10
2.1.1 Historie operačního systému Unix 10
2.1.2 Hlavní rysy operačního systému Unix 11
2.2 Operační systém BSD 13
2.3 Operační systém Linux 13
2.3.1 Historie operačního systému Linux 13
2.3.2 Kompatibilita operačního systému Linux 14
2.3.3 Distribuce operačního systému Linux 14
3 Programovací jazyky 16
3.1 Nejvýznačnější programovací jazyky 16
3.2 Graf četnosti použití 17
3.3 Popis vybraných programovacích jazyků 18
3.3.1 Assembler 18
3.3.2 Fortran 18
3.3.3 Pascal 19
3.3.4 Jazyk C 20
3.3.5 Java 22
3.3.6 Jazyk C# 24
3.3.7 PHP 25
3.3.8 Perl 26
4 Abstraktní datové typy 29
4.1 Vymezení pojmu abstraktního datového typu 29
4.2 Příklady abstraktních datových typů 30
4.2.1 Lineární seznamy 30
4.2.2 Zásobník 33
4.2.3 Orientovaný graf 33
5 Implementace domácích úloh 37
5.1 Pedagogické hledisko 37
5.1.1 Rozsah 37
5.1.2 Složitost 38
5.1.3 Přitažlivost pro studenta 38
5.1.4 Dostupnost materiálů 38
5.1.5 Přínos pro studenta 38
5.2 Zažité konvence 39
5.2.1 Názvy proměnných a funkcí 39
5.2.2 Rozvržení funkcí a jejich volání 40
5.2.3 Komentáře 40
5.2.4 Názvy souborů 41
5.2.5 Výpisy na standardní výstup 41
5.3 Technická stránka 41
5.3.1 Obecné konvence 41

5.3.2 Nové typy domácích úloh 42
6 Systém pro automatické vyhodnocování domácích úloh 45
6.1 Stručný popis 45
6.1.1 Interpreter 45
6.1.2 Porovnání výstupů – skript checker.php 45
6.1.3 Spouštění jednotlivých úloh – skript saferun.pl 45
6.1.4 Hlavní kontrolní skript – checkall.sh 46
6.1.5 Přidělování bodů – skript points.sh 46
6.1.6 Odesílání e-mailů – skripty preparemails.sh a sendmails.sh 47
6.1.7 Odhalování plagiátů – skript findduplicates.php 47
6.2 Modifikace systému 47
6.2.1 Zadání problému a návrhy jeho řešení 47
6.2.2 Použití změn 48
7 Závěr 50
Literatura 51
A. Obsah přiloženého CD 52

1 Úvod

Již ze samotného názvu celé bakalářské práce vyplívá, že veškeré mé snažení je přímo spojeno s předmětem Algoritmy. Tento předmět je vyučován jako povinný ve třetím semestru bakalářského studia studijního programu Informační technologie na Fakultě informačních technologií VUT v Brně. Každý student se v průběhu svého studia rozličně profiluje, ovšem snad všemi obory prostupují pojmy, kterými jsou datová struktura a algoritmus. Znalost problematiky, kterou poskytuje právě předmět Algoritmy, je dle mého názoru naprosto elementární disciplínou pro člověka, jež se chce orientovat v informačních technologiích. Pokud se letmo podíváme na projekty zadávané v rámci ostatních předmětů na naší fakultě, zjistíme, že k jejich vypracování bývá ve velkém množství případů chápána znalost algoritmů a datových struktur jako samozřejmost.

Má bakalářská práce tedy zahrnuje velice zodpovědný úkol. Vypracování zadání domácích úloh jak po stránce syntaktické, tak sémantické, může ovlivnit studenta při řešení problematiky jiných předmětů a může poznamenat jeho celkovou úspěšnost při studiu. Navíc by také mohlo dojít k získání nevhodných programátorských návyků, kterých se později člověk jen nelehko zbavuje, ať už se jedná o technické záležitosti zdrojového textu či jeho přehledné a pochopitelné formátování.

Podrobněji se budu zabývat koncepcí domácích úloh až v příslušné kapitole, ovšem již nyní bych rád zdůraznil jejich celkový charakter. Na rozdíl od jiných povinných předmětů vyučovaných na naší fakultě se snaží tvůrci domácích úloh pro předmět Algoritmy vymyslet pro studenta příklady tak, aby největší důraz padl právě na výukovou stránku věci. Cílem není např. vytvořit síťovou aplikaci, kde student využije konkrétní datovou strukturu nebo navrhnout grafické uživatelské rozhraní pro práci s telefonním seznamem, kde student použije pro řazení algoritmus quicksort. Domácí úlohy pro tento předmět se snaží studenta podnítit k podrobnému pochopení jednotlivých datových struktur a algoritmů, které se pro práci s takovýmito strukturami užívají. Z toho, co jsem právě řekl se může zdát, že tyto domácí úkoly nejsou pro studenta příliš záživné. A právě z toho důvodu se jejich tvůrci musí hluboce zamyslet nad formou zadání a jeho obsáhlostí, nad správnou koncepcí domácích úloh a způsobem, jakým studentovi budou podávány. Student by si dle mého názoru měl bezezbytku uvědomit, jak důležité pro něj může v budoucnu být, jestli správně pochopil jednotlivé domácí úlohy a vypracoval je skutečně samostatně bez plagiátorství. A pokud student tyto předpoklady splní, může sám v budoucnu užívat výhod, které plynou z vlastnictví správně implementovaných algoritmů nad datovými strukturami, v řadě jiných předmětů.

Má bakalářská práce se ale nezabývá jen implementací domácích úloh, ale také problematikou systému užívaného pro jejich opravování. O tomto systému se podrobněji rozepíši v příslušné kapitole, ale zde bych chtěl alespoň nastínit, jaký by měl být dle zadání můj příspěvek tomuto systému. Zmíněný systém se stará o zpracování zadání, bodování vypracovaných domácích úloh a také odhalování případných plagiátů. Mé snažení se týká výlučně části druhé, tedy kontroly domácích úloh s hlediska korektnosti vypracování. Při vyhodnocování nastává několik problémů při chybném vypracování domácí úlohy, jež mohou vézt v nejhorším případě až ke kolapsu celého systému a nutnosti některá studentská řešení opravovat ručně. Mým cílem je odstranit co nejvíce těchto stavů, popřípadě navrhnout nová řešení. Systém pro opravování domácích úloh je napsán pro operační systém GNU/Linux, a právě proto věnuji kapitolu této práce také drobnému přehledu týkajícího se GNU/Linuxu a Unixovým operačním systémům, jejich historii i současnosti.

S algoritmizací bezesporu přímo souvisí programovací jazyky, jejich třídy, určení, složitost apod.. Proto se také v příslušné kapitole věnuji několika vybraným programovacím jazykům a rovněž jejich historii.

2 Operační systémy

Jak jsem již uvedl v úvodu, celý systém pro automatické vyhodnocování domácích úloh je implementován pro operační systém GNU/Linux, tedy pro operační systém, který svůj filozofický základ, ale také některé normy, odvozuje od operačního systému Unix.

Pro důkladné pochopení systému pro automatické vyhodnocování domácích úloh bylo nutné mít alespoň základní znalosti operačních systémů odvozených od Unixu. Proto bych zde rád uvedl několik málo informací o těchto operačních systémech a jejich historii. Informace jsem čerpal z [8] a [4].

2.1 Operační systém Unix

V této podkapitole se zaměřím na historii vývoje operačního systému Unix a zmíním některé důležité milníky od chvíle jeho vzniku. Také připomenu důležité rysy jeho architektury a vůbec jeho filozofie.

2.1.1 Historie operačního systému Unix

2.1.1.1 Operační systém Multics, prvopočátky Unixu

Prvopočátek systému Unix se datuje do roku 1965, kdy se v Bellových laboratořích ve spolupráci s Massachussets Institute of Technology (MIT) a General Electric Company začal vyvíjet nový operační systém Multics. Cílem vývoje bylo vytvořit interaktivní systém s multiuživatelským přístupem na více terminálech, velkým výpočetním výkonem, efektivním uložením dat a bezpečným přístupem. Při návrhu byla uplatněna řada zásad, jež měly vést k úspěchu projektu. Mezi ty hlavní patřil například důraz na použití programovacího jazyka vyšší úrovně nebo dodržování metodiky nařizující provedení návrhu před samotnou implementací. Řada technik, jež v Multicsu byly využity, se ujala i v dalších operačních systémech. Mezi nejdůležitější patří segmentace, hierarchický souborový systém nebo stránkování na žádost.

I přes ambiciózní a propracovaný návrh nebyl nový operační systém programátorsky příliš přívětivý. Možná právě proto se časem z vývojářů oddělila skupina pod vedením K. Thompsona, D. Ritchieho a R. Kernighana, která navrhla vlastní operační systém, jež byl později nazván Unix (nejprve Unics podle již dříve zmíněného systému Multics). Nový systém byl multiuživatelský a multiprocesorový. Byl napsán pro šestnáctibitový počítač PDP-11 od firmy Digital equipment corporation (DEC) v křížovém assembleru, jež byl součástí operačního systému Gecos. Unix tedy ještě neexistoval jako plnohodnotný a soběstačný operační systém.

2.1.1.2 Reimplementace operačního systému Unix do jazyka C

Přelomovým rokem se v historii Unixu stal rok 1973, kdy byl systém revolučně reimplementován v novém jazyce C. Jazyk C vytvořil právě D. Ritchie v 70. letech. Nejprve vyvinul zcela beztypový jazyk BCLP, který následně rozšiřoval o strukturované datové typy. Takto vznikl první překladač jazyka C a také interpretovaný jazyk B. Unix byl nyní přenositelný také na jiné platformy, a to i přes fakt, že jazyk C byl původně navržen jako makroassembler pro počítač PDP-11. O 4 roky později (1977) se Unix dočkal své první instalace na jiný počítač, než na PDP. Nyní byl tedy Unix víceúlohový víceuživatelský a zcela přenositelný systém.

Naimplementování Unixu v jazyce C znamenalo pro vývojáře také prolomení jisté psychologické bariéry. Do té doby si totiž málo kdo uměl představit vývoj operačního systému ve vyšším programovacím jazyce. Nyní nestálo nic programátorům v cestě v psaní operačních systémů rychleji, efektivněji a pro různé platformy. Tvorbu větších projektů zatěžoval nyní hlavně nedostatek kvalitních metodik vývoje nebo také jistá neochota či strach z jejich nasazení.

2.1.1.3 Rozšíření operačního systému Unix, normy

Koncem 70. let se začalo objevovat hned několik větví Unixu. Bellovy laboratoře uvolnily Unix mimo své výzkumné centrum a ten se okamžitě začal uplatňovat zejména na universitách k vědeckým účelům. Mezi nejvýznamnější university patří Kalifornská universita v Berkeley (UCB), která distribuuje systém pod názvem Berkeley System Distribution (BSD), rozšiřující ho zejména o síťové služby. Bellovy laboratoře také pokračují ve vývoji Unixu pod názvem System n. V současnosti se jedná o System V Release 4 (SVR4). Bellovy laboratoře také pravidelně vydávají doporučení SVID obsahující základní charakteristiky systému. Těmito doporučeními se vývojáři víceméně řídí.

Jak Unix získával na popularitě, ukázalo se jako nezbytné, aby došlo k jeho standardizaci. Asi nejvýznamnější je v tomto směru norma POSIX (zkratka z *Portable Operating System Interface*), jež se nachází ve vlastnictví Institute of Electrical and Electronics Engineers (IEEE). POSIX se zabývá hlavně rozhraním a přenositelností zdrojových kódů. Také sdružení evropských autorů X/OPEN vytvořilo normu pod názvem XPG3.

Jak se v čase měnily požadavky na výpočetní techniku, měnil se také samotný Unix. Během jeho vývoje došlo ke vzniku mnoha někdy až diametrálně rozdílných modifikací, které často upadly v zapomnění nebo jsou používány pouze experimentálně, i přes to, že v mnoha ohledech předčí dnes masivně rozšířené komerční distribuce. Jako vhodný příklad se dá uvést operační systém Plan9 vyvíjený koncem 80. let v Bellových laboratořích za účelem návrhu distribuovaného síťového operačního systému pro přechod od minipočítačů a počítačů třídy mainframe k mikropočítačům pracujících v síti. Plan9 se svou vnitřní architekturou Unixu příliš nepodobá. I když se jedná o revoluční návrh a řešení operačního systému, nedošlo nikdy k jeho masivnímu rozšíření a Plan9 zůstal nasazen pouze v laboratorním prostředí.

2.1.1.4 Dopad existence operačního systému Unix na dnešní vývoj

Jak se Unix celá desetiletí vyvíjel, přinesl mnoho nových originálních pohledů na architekturu i funkčnost operačních systémů. Jeho prvky můžeme pozorovat nejen u historických operačních systémů, ale také u moderně navrhovaných. Jeden z významných rysů Unixu je pojetí souboru, jež je u některých operačních systémů z něho vycházejících dotažen ještě dále. Jako příklad může sloužit systém GNU/Hurd. Unix má zkrátka stále co nabídnout a pořád dokáže vývojáře inspirovat. Mezi nejznámější operační systémy vycházející z Unixu patří bezesporu Linux a mnohé další volně distribuované systémy. Unix po mnoho let prokazoval svou funkčnost a jeho znalost patří mezi základní vědomosti v oblasti informatiky.

2.1.2 Hlavní rysy operačního systému Unix

V předchozí kapitole jsem uvedl, že operační systém Unix byl hned na počátku své existence navržen jako

- víceúlohový (multiprocesový),
- víceuživatelský (multiuživatelský).

Nyní obě vlastnosti rozeberu podrobněji.

2.1.2.1 Schopnost zpracovávat více úloh najednou

Obě vlastnosti mají co dočinění s faktem, že Unix byl koncipován hned od prvopočátku pro provoz na velkých počítačích. Předpokládalo se, že takové počítače budou obsluhovat hned několik procesů najednou, a právě proto byl Unix navržen jako víceúlohový. V jeden časový okamžik na jednom jedno-jádrovém procesoru ovšem nelze reálně zpracovávat více úloh najednou. Právě proto Unix využívá metody zvané timesharing. Jedná se o cyklické přidělování procesorového času úlohám zpravidla podle nastavených priorit. Čím má proces vyšší prioritu, tím více mu bude poskytnuto procesorového času. Naopak úlohám s menší prioritou je procesorový čas přidělován s menší frekvencí či v mnohem kratších časových jednotkách, než je tomu u úloh s prioritou vyšší. Timesharing spadá plně do režie operačního systému. Jelikož jsou jednotlivé časové okamžiky relativně krátké, obvykle od 30 ms do 150 ms, připadne uživateli zpracovávání úloh souběžné, ovšem jedná se jen o klam. Tento klam ale vede k nutnosti zavést v operačním systému také další funkce a služby, jako je například zajištění vzájemné komunikace jednotlivých procesů nebo správa procesů a paměti. Správa paměti je neodmyslitelnou součástí operačního systému, protože ten musí zajistit, aby procesy běžící současně nemohly libovolně přepisovat paměť, tedy nemohly v paměti libovolně zabrat prostor přidělený již jinému procesu.

2.1.2.2 Schopnost obsloužit několik uživatelů najednou

Pokud operační systém dovoluje souběžně zpracovávat několik běžících procesů, měl by umožňovat také souběžný přístup více uživatelů v jediný okamžik. Fakt, že je některý operační systém multiuživatelský, bývá ale často mylně interpretován, jako schopnost operačního systému vytvářet uživatelské účty. V zásadě toto vysvětlení není zcela nepravdivé, ovšem vůbec nezahrnuje definici souběžného přístupu. Existence více uživatelských účtů bývá předpokladem multiuživatelského přístupu, ale rozhodně k němu není ekvivalentní. Správná definice by měla spíše operovat s pojmem sezení nebo cizojazyčně *session*. Sezení se dá charakterizovat jako doba práce uživatele se systémem. Operační systém je tedy multiuživatelský, pokud umožňuje více souběžných sezení.

2.1.2.3 Terminál

V předchozích dvou odstavcích jsem vysvětloval dvě podstatné vlastnosti systému Unix. Jednalo se o multiúlohovost a multiuživatelskost. Pokud si představíme kombinaci těchto dvou pojmů, tedy existenci velkého počítače umožňujícího krom souběžného zpracovávání úloh také souběžnou práci více uživatelů, a připustíme, že je fyzicky docela nepohodlné, aby u jediného stroje pracovalo v jeden okamžik více lidí, nutně dojdeme k pojmu terminál. Slovem terminál označujeme zařízení, jež má zajistit interakci uživatele s počítačem, na kterém běží operační systém. Jedná se tedy o hardwarové zařízení složené z klávesnice a zobrazovací jednotky. Terminál nemá řešit žádné výpočetní úlohy, ale pouze zobrazovat uživateli výstupy a počítači s operačním systémem posílat uživatelské vstupy. Tento přístup se ale v současnosti jeví jako zastaralý. Dnes se spíše preferuje zavedení plnohodnotných počítačů namísto terminálů, na nichž ovšem běží programy nazývané terminály. Ty dokáží plně emulovat hardwarový terminál. Výhodou také je, že dokáží běžet na libovolném operačním systému. Takovýto vzdálený přístup obvykle vyžaduje ověřování totožnosti uživatele, což bývá běžně řešeno pomocí uživatelského jména a hesla, popřípadě i jinými metodami, jako dnes již masivně rozšířenými magnetickými kartami, otiskem prstu či v současnosti méně obvyklými, jako zkoumání struktury sítnice oka a podobně.

2.2 Operační systém BSD

Vývojem systému BSD se v Berkeley zabývala skupina Berkeley Computer Systems Research. V roce 1993 tato skupina vyčerpala poslední finance vyčleněné na projekt. To vedlo k výměně hlavních vývojářů operačního systému, ale také rozdělení jeho vývoje do více větví. Otázka financí ovšem nebyla jediným důvodem pro změnu. Značnou roli hrála též poslední verze BSD vyvíjená v Berkeley nezávisle na ostatních Unixových operačních systémech. Jednalo se o verzi 386BSD určenou pro platformu Intel 386. Její uzavřenost a pomalé reakce jejích tvůrců na zasílané opravy měly nemalý podíl na odvrácení vývojářů od původní koncepce a hledání nových východisek. Hned na jaře roku 1993 vznikl projekt FreeBSD, od něhož se na podzim téhož roku oddělila větev NetBSD inspirovaná snahou o co možná největší přenositelnost systému. O tři roky později, v roce 1996, vznikla skupina vývojářů vedená Theo de Raatem, která se zaměřila na bezpečnost a robustnost systému. Vznikla tak další větev pod názvem OpenBSD.

NetBSD a OpenBSD se nyní navzájem ovlivňují. Do NetBSD se z OpenBSD přenášejí bezpečnostní prvky, naopak v NetBSD probíhá rychlý vývoj ovladačů, jež je dále zužitkován v OpenBSD. Vzájemná komunikace tvůrců obou systémů probíhá již od prvopočátku přes síť Internet. Tato metoda komunikace nejen umožnila volné šíření nových verzí systému, ale také přinesla nový trend vývoje softwaru.

2.3 Operační systém Linux

2.3.1 Historie operačního systému Linux

Operační systém Linux má na svědomí fenomén frustrace, který snad odjakživa žene vývojáře k novým nápadům a odhodlání k vyzkoušení nových postupů. Díky frustraci vznikla již řada „programátorských perel“ [4], jako například operační systém Unix, Stallmanův textový editor GNU Emacs nebo Knuthův typografický systém TEX. Tím frustrovaným byl v případě operačního systému Linux tehdy jednadvacetiletý finský student Linus Torvalds, který se nedokázal smířit s nepříjemnými omezeními systému Minix od Andyho Tanembauma pro Intel 386. Proto se pokusil vytvořit vlastní operační systém, jehož vznik oficiálně oznámil v říjnu roku 1993. Tehdy se jednalo o verzi 0.02. Ta uměla pracovat s IDE diskem, obsahovala malý souborový systém a dal se na ni spustit překladač jazyka C. Již od této verze systém neobsahoval žádné zdrojové texty z původního Unixu z Bellových laboratoří ani z BSD. Od verze 0.03 se změnilo číslování systému na 0.10 a začalo na něm pracovat více programátorů. Verze 1.0 byla ve znamení změny číslování jader. Zavedla se koncepce x.y.z, kde x značí generaci jádra, y subgeneraci jádra a z pořadové číslo jádra v rámci subgenerace. Pokud je y značící subgeneraci liché číslo, znamená to, že jádro pochází z nestabilní série. Takové jádro je vhodné jen pro vývojáře operačního systému Linux. Jeho používání může způsobit pád systému, ztrátu dat, nebo dokonce poškození počítače samotného. Pro uživatelský přístup je vhodné používat jen jádra ze sudých subgenerací. První použitelná verze Linuxu měla označení 1.2.z. Od série 2.0.z je Linux přenositelný i na jiné platformy, než je Intel 386. V dalších sériích přibývaly a stále přibývají vždy nové funkce, díky čemuž je Linux právem označován za Unix 90. let. Díky jeho malým nárokům na hardware počítače vyhovuje k použití jak v kapesních počítačích, tak ve výkonných grafických stanicích. Přesto se od původní verze zvětšila velikost komprimovaných zdrojových textů přibližně 450krát.

Jádro systému tedy ovlivňuje jeho celkovou funkčnost i výkon. Existují v zásadě dva názory na instalaci jádra. První z nich preferuje využívání již předpřipravených a optimalizovaných jader (implicitní jádra), ten druhý zase říká, že jádro by se mělo ještě před zavedením manuálně zkonfigurovat pro dosažení optimálního výkonu. Samotné jádro se následně zkompiluje a zavede do paměti.

2.3.2 Kompatibilita operačního systému Linux

Linux sice vychází z koncepce Unixu, ovšem kompatibilita zde existuje jen na úrovni zdrojového kódu. Zkompilované programy nemusí být dokonce přenositelné ani v rámci jednotlivých distribucí operačního systému Linux. To bývá obvykle způsobeno použitím rozdílných verzí knihoven a podobně. Linux splňuje i velkou část normy POSIX.

2.3.3 Distribuce operačního systému Linux

2.3.3.1 Základní programové vybavení

Linux je volně šiřitelný operační systém. Lze ho získat buď zcela zdarma, nebo si zakoupit jednu z mnoha komerčních distribucí. Pojem Linux chápeme jako jeho uživatelsky nepoužitelné jádro doplněné o základní softwarové vybavení. Mezi toto vybavení patří shell, textový editor, překladače a interprety programovacích jazyků a další nástroje. Softwarové vybavení obvykle není závislé na verzi jádra, ale v některých případech tomu může být právě naopak.

2.3.3.2 Nadace na podporu volně šiřitelného softwaru, GNU GPL

O vývoj kvalitního softwaru pro operační systém Linux se stará nadace na podporu volně šiřitelného softwaru Free Software Foundation. Ta byla založena v roce 1985 Richardem Stallmanem s cílem podporovat práva uživatelů počítačů používat, studovat, kopírovat, modifikovat a redistribuovat počítačové programy. Nadace také vyvíjí svůj vlastní operační systém Hurd. Veškeré programové vybavení je vyvíjeno pod záštitou projektu GNU (GNU's Not Unix). Filozofií GNU je používání svobodného softwaru všemi způsoby přinášejícími společenský užitek. Software lze totiž jakkoli kopírovat a modifikovat a právě díky tomu může být společnosti prospěšný. Nadace věří, že by právě takový software měli uživatelé využívat. FSF poskytuje na FSF/UNESCO (adresář svobodného softwaru) přes 4000 balíčků svobodného softwaru. FSF podporuje jeho šíření také poskytováním FTP serverů, které jsou zrcadleny po celám světě. Pro vývojáře mnoha GNU i non-GNU projektů poskytuje Free Software Foundation CVS server Savannah. Asi mezi nejznámější projekty FSF patří unikátně navržený překladač GCC, textový editor GNU Emacs či příkazový interpret GNU Bash.

Operační systém Linux (jeho jádro) doplněný o zmíněné programové vybavení od Free Software Foundation se pak stává plnohodnotným a nadstandardně vybaveným operačním systémem. Takovýto Linux následně označujeme GNU/Linux. Veřejná licence GNU GPL (General Public License), pod níž je pak operační systém Linux distribuován, umožňuje kopírovat, sdílet, modifikovat a dále distribuovat software za podmínky, že ten nebude nikdy od této licence odloučen. GNU GPL ovšem také umožňuje komerční šíření softwaru.

2.3.3.3 Instalace operačního systému Linux

Manuálně instalovat operační systém Linux se rozhodně běžnému uživateli nedoporučuje. Bývá to zpravidla velice složitá a časově náročná činnost určená jen velice vyspělým uživatelům. Právě tento zádrhel vedl vývojáře zejména komerčních distributorů Linuxu k poskytnutí instalačních balíčků,

jež vlastně představují takový „úhledný kabát“ oblečený na tento těžkopádný časově náročný proces. Tento „kabát“ většinou představuje operační systém Linux obohacený o administrační nástroje například pro správu a instalaci softwarových balíků. Tyto nástroje pracují většinou interaktivně s uživatelem a umožňují pak instalace a provádění změn i relativním nováčkům. Přesto všechno není administrace systému Linux nic jednoduchého a po uživateli vyžaduje značné znalosti, které se nedají získat za malou chvíli.

2.3.3.4 Pořízení operačního systému Linux

Jednotlivé distribuce se dají rozdělit na komerční a nekomerční. Komerční distribuce poskytují vývojářům potřebné finanční prostředky pro další rozvoj a přidávání nových balíčků. Mezi nejznámější komerční distributory patří Mandrake, Slackware nebo například RedHat Linux.

Existují ale rovněž neméně oblíbené nekomerční distribuce, jako například Debian GNU/Linux. Na jeho vývoji pracuje kolem 2000 lidí všude na světě. Každý, kdo má zájem o vývoj a správu Linuxu či má možnost jinak pomoci, se může připojit k jeho vývoji. V současnosti je k dispozici Debian GNU/Hurd běžící na jádru Hurd a Debian GNU/NetBSD běžící na jádru NetBSD. Distribuce Debianu se dají rozdělit na stabilní, testovací a nestabilní. Nestabilní distribuce v současnosti obsahuje kolem 10000 balíčků, ovšem mnohé funkce nemusí být k dispozici nebo mohou způsobovat chyby. Řádně fungující části nestabilní distribuce se následně přesunují do distribuce testovací. Ta má zajistit jejich řádné otestování. Posledním krokem je přesun balíčků do části stabilní, která by již neměla obsahovat žádné chyby. V současnosti tato distribuce obsahuje kolem 8500 balíčků. Operační systém Debian GNU/Linux lze podobně jako u komerčních distribucí získat přímo z Internetu nebo instalovat z CD či DVD.

3 Programovací jazyky

Každý programovací jazyk má své výhody a nevýhody, své silné a slabé stránky. Jednou z prvních otázek, kterou si klade softwarový inženýr, když začíná pracovat na návrhu projektu, je právě vhodný výběr programovacích jazyků v dané situaci. Přitom musí brát na vědomí spoustu faktorů, jež ovlivňují tuto volbu. Není to jen otázka výhod programovacího jazyka, jednoduchost a rychlost psaní konstrukcí, ale také schopnost programátorů daného oddělení pracovat s jazykem efektivně a nemuset se učit nový programovací jazyk, pokud by to znamenalo přemrštěné náklady. V úvahu musí softwarový inženýr vzít také požadavky zákazníka, schopnost programovacího jazyka kooperovat na úrovni zdrojových textů s jinými programovacími jazyky nebo budoucí plány na rozšiřování výsledného systému. Právě z toho důvodu by měl mít softwarový inženýr jisté povědomí o výhodách a nevýhodách jednotlivých jazyků a také jejich historii.

Pro tvorbu domácích úkolů pro předmět algoritmy byl nejprve vybrán programovací jazyk Pascal, který ovšem v posledních letech upadá v nelibost u programátorské veřejnosti a zůstává ve stínu jazyka C. Tvůrci domácích úloh pro předmět Algoritmy na tuto změnu zareagovali poměrně rychle a v předchozích letech reimplementovali veškerá zadání právě do jazyka C. Došlo také k nové implementaci systému pro automatické vyhodnocování domácích úloh pomocí jazyků Bash, PHP a Perl.

A právě tyto jazyky a také některé jejich předchůdce bych rád přiblížil v této kapitole.

3.1 Nejvýznačnější programovací jazyky

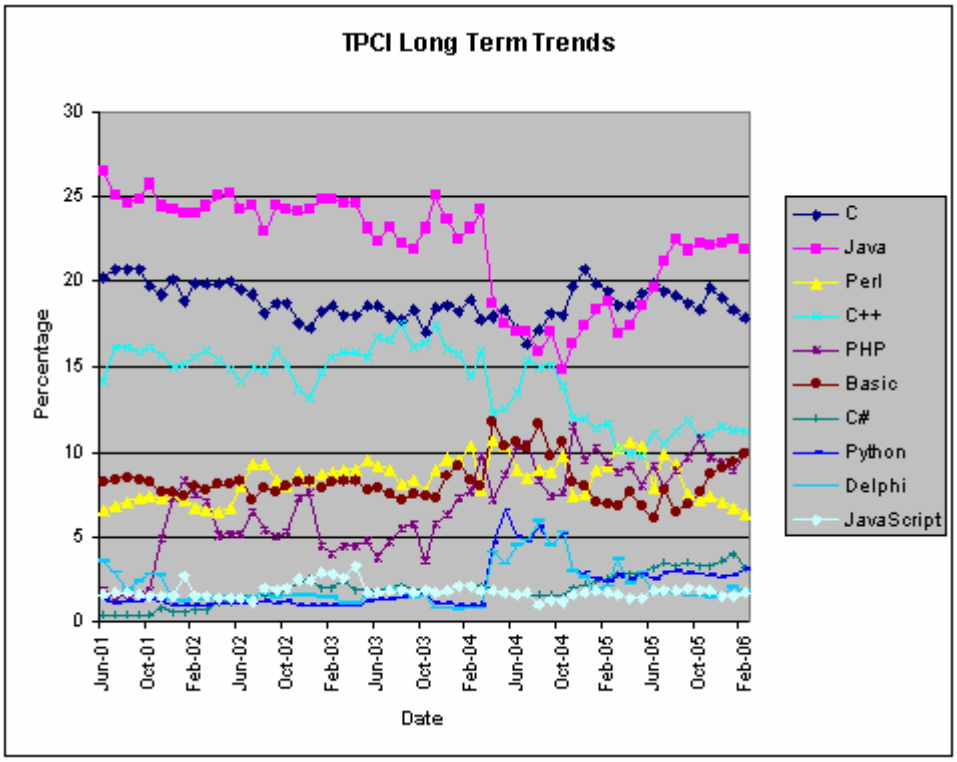
Tabulka obsahuje přehled významných neobjektově orientovaných programovacích jazyků. Sloupec rok značí období, v němž byl programovací jazyk vyvíjen, sloupec předchůdce pak obsahuje buď pomlčku, značící, že jazyk nevychází z žádného jiného jazyka, nebo seznam všech jazyků, jež sloužily jako podklad pro vznik daného programovacího jazyka.

Jazyk	***Rok***	***Autor***	***Předchůdce***
Assembler	1952 - 1988	Kolektiv autorů	-
Fortran	1954 - 1957	J. Backus (IBM)	Assembler
Algol 58	1958	Kolektiv autorů	Fortran
Algol 60	1958 - 1960	Kolektiv	Fortran
Cobol	1959 - 1960	Kolektiv autorů	Assembler
APL	1956 - 1960	K. Iverson (Harvard)	-
Lisp	1956 - 1962	J. McCarthy (MIT)	-
Snobol	1962 - 1966	R. Griswold (Bell Labs)	-
PL / I	1963 - 1964	IBM - kolektiv	Fortran, Algol 60, Cobol
Basic	1964	Kemeney a Kurtz	Algol 60
Simula 67	1967	Q. J. Dahl (Norwegian Computing Center	Fortran
Algol 68	1963 - 1968	Kolektiv autorů	Algol 60

Pascal	1971	N. Wirth (ETM Zurich)	Algol 60
RTL / 2	1971 - 1974	John Barnes	Algol 60, Algol 68
C	1974	D. Ritchie (Bell Labs)	Algol 60, Algol 68
Paralelní Pascal	1975	P. Brinch Hansl	Pascal
CLU	1974 - 1977	B. Liskov (MIT)	Simula 67
Euclid	1977	Kolektiv autorů	Pascal
PLZ	1977	Zilok Inc.	Pascal
Modula	1977	N. Wirth (ETM Zurich)	Pascal
Ada	1979	J. Ichbiah a kolektiv	Pascal, Simula 67, Modula

Obr. 3.1 – Tabulka informací o programovacích jazycích

3.2 Graf četnosti použití



Obr. 3.2 – Graf četnosti použití nejpoužívanějších programovacích jazyků

Graf [7] znázorňuje oblibu desíti nejpoužívanějších programovacích jazyků mezi roky 2001 až 2006. Silný propad Javy v roce 2004 přičítám zejména uvedení Visual Studia 2003 z dílny Microsoftu na trh. Přesto je stále objektově orientovaný jazyk Java nejoblíbenější ze všech programovacích jazyků. C#, pracující na platformě .NET, značně zaostává za Javou, i když se jedná o jejího největšího konkurenta.

3.3 Popis vybraných programovacích jazyků

V této podkapitole stručně popíši programovací jazyky, které považuji za důležité z hlediska jejich rozšíření a popularity nebo také kvůli jejich vlivu na další rozvoj informačních technologií. Veškeré informace jsem získal z Wikipedie [8], protože se mi tam dané téma zdá velice kvalitně rozpracované. Z velkého množství dostupných informací jsem následně provedl výtah.

3.3.1 Assembler

Assembler je programovací jazyk nízké úrovně, což znamená, že se velmi blíží strojovému kódu. Někdy se také správněji nazývá jazyk symbolických instrukcí. Slovo assembler se vžilo jako název programovacího jazyka i přesto, že se původně jednalo o název překladače, který tento jazyk překládá do strojového kódu.

Protože je assembler programovací jazyk nejnižší úrovně, je strojově závislý, a proto musí mít v podstatě každý procesor svůj vlastní odlišný assembler.

Assembler tvoří pouze zástupné symboly, které přímo odpovídají strojovému kódu. Činnost procesoru je totiž řízena pomocí instrukcí strojového kódu, což jsou vlastně posloupnosti čísel s přesným významem. Pokud bychom vyjádřili instrukce binárně, jasně bychom viděli, jak procesor reaguje přímo na změny napětí na jeho vstupních vodičích. Mnemotechnické symboly se mnohem lépe zapamatovávají, než nic neříkající čísla strojového kódu. Další výhodou je používání symbolů pro adresy dat a instrukcí, které se při překladu nahradí skutečnou aktuální adresou.

Překladače také obvykle poskytují tzv. makra, která nahrazují často používané posloupnosti jednotlivých instrukcí. Další součástí assembleru bývají direktivy (pseudoinstrukce), které umožňují ovlivňovat překlad, vyhrazovat místo v paměti pro proměnné nebo manipulovat s čítačem adres.

3.3.2 Fortran

3.3.2.1 Základní charakteristika

Jedná se o starý programovací jazyk určený zejména pro vědecké výpočty a numerické aplikace. Jméno vzniklo ze zkratek FORmula TRANslator. Jazyk obsahuje řadu nevýhod, které vyplývají hlavně z krkolomného způsobu formátování. Je nutno používat příkaz goto a čísla řádků. Některé problémy byly v pozdějších verzích odstraněny, jiné přetrvaly.

3.3.2.2 Historie

Kompilátor pro Fortran byl vyvíjen letech 1954 až 1957 v americké firmě IBM týmem pod vedením Johna W. Backuse. Byl to optimalizovaný kompilátor, protože autoři byli toho názoru, že nikdo nebude používat jazyk, jehož výkonnost není srovnatelná s assemblerem. Jazyk byl široce používán vědci pro psaní numericky náročných programů a zároveň se objevovaly kompilátory, které generují rychlejší kód. Postupně se objevilo několik standardů jazyka, jako například Fortran IV a Fortran 66, Fortran 77, Fortran 90, Fortran 95 a Fortran 2003.

Význam Fortranu v historii programovacích jazyků je obrovský. Pokud se detailněji zaměříme na tabulku programovacích jazyků v podkapitole 3.1, zjistíme, že právě Fortran stál na počátku cesty vedoucí např. ke vzniku jazyka C nebo jazyka Pascal.

3.3.3 Pascal

3.3.3.1 Základní charakteristika

Pascal vznikl jako programovací jazyk k výuce programování. Jeho název byl zvolen na počest francouzského filosofa, matematika a fyzika Blaise Pascala.

První návrh programovacího jazyka Pascal se objevil na začátku 70. let. Profesor Niklaus Wirth z Vysoké školy technické v Curychu přišel tenkrát s nápadem vytvořit jazyk vhodný pro výuku programování, který by byl založen na omezeném počtu srozumitelných konstrukcí, a současně chtěl navrhnout strukturu jazyka tak, aby bylo snadné implementovat Pascal na většině tehdejších počítačů.

První verze Pascalu byla publikována roku 1971, o 3 roky později, v roce 1974, byla uveřejněna opravená definice jazyka. Norma ISO byla pak vydána až v roce 1981. Vedle akademické verze Pascalu od profesora Niklause Wirtha vznikla řada komerčních implementací Pascalu, které se od něj dle normy ISO více či méně odchylovaly, zejména zavedením dalších konstrukcí zjednodušujících praktické programování. Asi největší popularitu získal programovací jazyk Pascal díky své implementaci Turbo Pascal od firmy Borland. Objektové rozšíření Pascalu se pak stalo i základem systému Delphi téže firmy.

3.3.3.2 Historie

První kompilátor Pascalu byl navržen v Curychu pro počítače řady CDC 6000 a do provozu byl uveden v roce 1970.

V Severní Americe vznikl první kompilátor Pascalu na Illinoiské univerzitě pro počítač PDP-11. Projekt byl veden Donaldem B. Gilliesem a generoval nativní strojový kód.

Za účelem rychlého rozšíření jazyka vznikl v Curychu tzv. compiler porting kit, zahrnující kompilátor do „virtuálního“ strojového kódu (mezikódu) a simulátor toho kódu. Z této sady vznikl p-System. Ten měl umožnit vznik kompilátorů do pravých strojových kódů, ovšem vznikaly také implementace, jež ho využily k vytvoření interpretačního systému. Typickým příkladem je pozoruhodná implementace USCD, jež jej využila k vytvoření interpretačního systému USCD p-System.

Z dalších implementací zmíním např. IP Pascal, který nejdříve používal Micropolis DOS, ale rychle se přesunul na CP/M běžící na Z80.

Začátkem 80. let byl UCSD Pascal portován pro počítače Apple II a AppleIII, aby tak poskytl strukturovanou alternativu k interpretům BASICu, dodávaným spolu s počítačem.

Další kompilátor pojmenovaný Blue Label Pascal vznikl v 80. letech. Autorem byl Anders Hejlsberg, který kompilátor napsal pro počítač Nascom-2. Hejlsberg začal později pracovat pro firmu Borland a svůj kompilátor přepsal pro Turbo Pascal na architektuře IBM PC. Nový kompilátor prodával za $49.95, což byla mnohem nižší cena, než za jakou prodával původní Blue Label Pascal.

Laciný kompilátor společnosti Borland měl velký vliv na komunitu okolo Pascalu, která se koncem 80. let zaměřila především na architekturu IBM PC. Turbo Pascal, dostupný pouze na této architektuře, překládal přímo do strojového kódu procesoru Intel 8088, čímž docílil mnohem větší rychlosti, než interpretované návrhy.

Objektově orientované programování se poprvé objevilo u verze Turbo Pascal 5.5 od společnosti Borland. Později se Borland rozhodl, že je potřeba komplikovanějších vlastností, a začal pracovat na programovacím prostředí Delphi. Vývojáři firmy přitom vycházeli z návrhu jazyka Object Pascal společnosti Apple. Jméno Object Pascal se také pro tento nový programovací jazyk po určitou používalo, ovšem následně bylo změněno na Delphi.

3.3.4 Jazyk C

3.3.4.1 Základní charakteristika

Jazyk C, který byl vyvinut Kenem Thompsonem a Dennisem Ritchiem původně pro potřeby operačního systému Unix, zažívá nejen v posledních letech období obrovské obliby. Z grafu v podkapitole 3.2 můžete jednoduše vyvodit, že se momentálně jedná o nejpoužívanější programovací jazyk. C našel využití zejména při psaní systémového softwaru a aplikací.

Jazyk C je nízkoúrovňový, kompilovaný, relativně minimalistický programovací jazyk. C je díky své velké síle vhodný pro implementaci většiny systémových součástí (ovladače a jádro operačního systému). Přesto může nastat situace, kdy bychom potřebovali jazyk ještě nižší úrovně. S tím ovšem jazyk C počítá, protože poskytuje rovněž tzv. inline assembler, tedy metodu zápisu assembleru přímo do zdrojového kódu. Zdrojový kód jazyka C se přitom vyznačuje mnohem lepší čitelností, než zdrojový kód jazyka assembler, je jednodušší ho zapsat, a navíc se dá snadněji přenést na jiné architektury. Právě proto je programovací jazyk C tak oblíbený vývojáři operačních systémů, překladačů, knihoven a interpretrů vysokoúrovňových jazyků.

Ukládání dat je v C řešeno třemi základními způsoby:

- Statickou alokací paměti (při překladu),
- automatickou alokací paměti na zásobníku,
- dynamickou alokací na haldě (heap) pomocí knihovních funkcí.

Jazyk nedisponuje žádnou abstrakcí nad alokací operační paměti. To se projevuje tak, že se s pamětí pracuje přes datový typ zvaný ukazatel, který obsahuje odkaz na místo v operační paměti, ovšem můžeme nad ním provádět také aritmetické operace. Ukazatelé tedy existují nezávisle na proměnných, na něž se odkazují, a je na odpovědnosti programátora, aby nedocházelo k únikům paměti, odkazování na nealokovanou paměť a dalším podobným nešvarům spojeným s používáním ukazatelů.

Jazyky Java a C#, oba odvozené od C, používají méně univerzální způsob odkazování alokovaných proměnných, který snižuje pravděpodobnost chyby v programu. Jazyk C++, původně rozšíření jazyka C, si ovšem ukazatele zachoval.

Mnoho dalších moderních programovacích jazyků přebírá způsob zápisu (syntaxi) z jazyka C. Patří mezi ně například zmíněná Java či Perl a PHP.

3.3.4.2 Historie

V roce 1978 vydali Dennis Ritchie a Brian Kernighan první vydání knihy The C Programming Language. Tato kniha, mezi programátory C známá jako „K&R“, sloužila po mnoho let jako neformální specifikace jazyka. Verze C, kterou takto popsali, bývá označována jako „K&R C“. Druhé vydání knihy popisovalo novější standard ANSI C.

K&R zavedli následující vlastnosti jazyka:

- Datový typ struct,
- datový typ long int,
- datový typ unsigned int,
- operátor =+ byl změněn na +=, a podobně (=+ mátl lexikální analyzátor překladače C).

K&R C je považován za základní normu, kterou musejí obsahovat všechny překladače jazyka C. Ještě mnoho let po uvedení normy ANSI C byl K&R samozřejmým minimálním společným základem, který využívali programátoři v jazyce C. To bylo také způsobeno neúplnou podporou

normy ANSI C u některých překladačů jazyka C. Program napsaný jen přesně podle normy ANSI C by potom nemusel být zcela přenositelný.

V několika následujících letech po uvedení normy K&R C bylo uvedeno a přidáno několik „neoficiálních“ vlastností jazyka. Tyto vlastnosti podporovaly jen některé překladače od AT&T a několika dalších dodavatelů. Změny zahrnují

- datový typ void * a funkce vracející void,
- funkce vracející typ struct nebo union,
- ukládání položek struktury do odděleného jmenného prostoru pro každou strukturu,
- modifikátor const,
- standardní knihovnu zahrnující většinu funkcí implementovaných různými dodavateli,
- výčtový typ enumeration,
- datový typ float.

3.3.4.3 Norma ANSI C a ISO C

Ke konci 70. let začal programovací jazyk C nahrazovat programovací jazyk BASIC, který do té doby dominoval jako přední programovací jazyk pro mikropočítače. Popularita jazyka C se ještě zvýšila v 80. letech, kdy se C začalo používat na architektuře IBM PC. V té době se také projevila snaha ze strany Bellových laboratoří o rozšíření jazyka C o nové objektové prvky. Toto rozšiřování dostal za úkol Bjarne Stroustrup, ale i další programátoři. Jejich snaha vyvrcholila vznikem nového objektově orientovaného jazyka, který byl nazván C++. Že tito vývojáři vytvořili úspěšný projekt, si můžeme snadno ověřit i dnes, kdy je C++ nejrozšířenější programovací jazyk pro aplikace běžící na operačních systémech Microsoft Windows. C zůstává stále populárnější ve světě Unixu.

Standardní specifikace jazyka C ovšem stále mnohým chyběla. Proto v roce 1983 se American National Standards Institute (ANSI) dohodl na sestavení komise X3J11, aby vytvořil standardní specifikaci jazyka C. Standard byl dokončen teprve až roce 1989 a schválen jako ANSI X3.159-1989 „Programming Language C“. Tato verze jazyka je často stále označována jako ANSI C. V roce 1990 byl standard ANSI C převzat institucí International Organization for Standardization (ISO) jako norma „ISO 9899|ISO/IEC 9899:1990“.

Již výše jsem zmiňoval, že jazyk C byl rozšířen o velký počet „neoficiálních vlastností“, jež nebyly vedeny v obecně dodržované normě K&R C. A právě jedním z nejdůležitějších cílů standardizačního procesu ANSI C bylo vytvořit nadmnožinu K&R C zahrnující mnoho těchto „neoficiálních vlastností“. Navíc standardizační komise přidala několik vlastností jako funkční prototypy (vypůjčené z C++) a schopnější preprocesor.

Norma ANSI C je dnes podporována téměř všemi globálně rozšířenými překladači. Většina kódu psaného v současné době v jazyce C je založena na normě ANSI C. Jakýkoli program napsaný pouze ve standardním C je přeložitelný a spustitelný na jakékoli platformě, která odpovídá tomuto standardu. Nicméně mnoho programů se dá přeložit pouze na jedné platformě nebo jedním překladačem. To může být způsobeno:

- použitím nestandardních knihoven, např. pro grafiku,
- faktem, že některé překladače v implicitním módu neodpovídají standardu ANSI C.

3.3.4.4 Norma C99

Na konci 90.let došlo k vydání dokumentu ISO 9899:1999 (obvykle nazývaný C99), který byl následně v březnu 2000 přijat i jako ANSI standard. Stalo se to i přes téměř bezvýhradné soustředění vývojářů po standardizaci jazyka C v roce 1989 na programovací jazyk C++.

Norma C99 představila několik nových vlastností, které se stejně jako v předchozím případě u normy ANSI C již delší dobu objevovaly v mnohých překladačích programovacího jazyka C. Některé z nich uvádím v následujícím přehledu.

- Inline funkce.
- Proměnné mohou být deklarovány kdekoli (V C89 mohly být deklarovány pouze na začátku bloku).
- Nové datové typy, včetně long long int (tj. 64 bitový integer), bool (logický ano/ne typ) nebo typ complex určený na reprezentaci komplexních čísel.
- Pole s nekonstantní velikostí.
- Podpora pro zakomentování jednoho řádku //, tak jako v jazycích C++ nebo BCPL.
- Nové knihovní funkce, hlavně ve formě náhrady za funkce náchylné na buffer overflow, např. snprintf().
- Nové hlavičkové soubory, např. stdint.h

Kompilátory, které se začaly více méně řídit novou normou, umožňují použít více optimalizací, ovšem za cenu občasné nekompatibility s některými zdrojovými texty psanými podle starší normy. To je způsobeno tím, že standard C99 je v některých ohledech přísnější než původní standard C89. Například je zakázáno odkazovat se na stejnou paměť ukazateli jiných typů.

Prozatím neexistuje kompilátor, který by plně implementoval normu C99. Z kompilátorů, které se normě dosti přibližují, se dá jmenovat výborný GNU kompilátor GCC. Slabého rozšíření se norma C99 také dočkala z toho důvodu, že velké firmy jako Microsoft nebo Borland neprojevily velký zájem o implementaci normy C99. To je způsobeno především jazykem C++, který již nyní poskytuje většinu funkcí a vlastností této normy. Často se tak ale právě děje v kombinaci s nekompatibilitou s normou C99 (datový typ complex v C99 versus třída complex v C++).

3.3.5 Java

3.3.5.1 Základní charakteristika

Java je objektově orientovaný programovací jazyk. Vyvinula ho firma Sun Microsystems, která ho představila 23. května 1995.

Java je jedním z nejpoužívanějších a nejpopulárnějších programovacích jazyků na světě. Jako jeho největšího konkurenta spatřuji programovací jazyk C# z dílny Microsoftu, o kterém se podrobněji zmíním v následující podkapitole 3.3.6. C# ovšem ani zdaleka nedosahuje popularity programovacího jazyka Java. Toto dokládá rovněž graf uvedený v podkapitole 3.2. Díky své přenositelnosti je Java používána tam, kde existuje požadavek na zkombinování práce různých systémů. Může se jednat o čipové karty na platformě JavaCard, mobilní telefony a různá zabudovaná zařízení na platformě Java ME, aplikace pro desktopové počítače na platformě Java SE nebo rozsáhlé distribuované systémy, které pracují na řadě spolupracujících počítačů a jsou rozprostřené po celém světě. Zde se využívá platformy Java EE. Souhrnně nazýváme výše zmíněné platformy Java.

3.3.5.2 Důležité vlastnosti

Jazyk Java obsahuje spoustu dobrých a užitečných vlastností. Proto by bylo na místě podívat se podrobněji na některé z nich.

- Jednoduchost. Syntaxe jazyka Java zjednodušuje známou verzí syntaxe jazyka C a C++. To vede také k její celkové oblíbenosti, protože programátoři se nemusí učit od začátku nové neznámé konstrukce. Navíc odpadla většina konstrukcí,

které způsobovaly programátorům problémy, a na druhou stranu přibyla řada užitečných rozšíření.

- Objektová orientace. Jazyk využívá jednoduché dědičnosti. Krom osmi primitivních datových typů jsou všechny ostatní datové typy objektové.
- Distribuovanost. Java je navržena pro podporu aplikací v síti. Java disponuje podporou mnoha různých úrovní síťového spojení, prací se vzdálenými soubory nebo možností vytvářet distribuované klientské aplikace a servery.
- Interpretovanost. Java vytváří mezikód (bajtkód) místo skutečného strojového kódu, jak tomu je např. u jazyků C a C++. Mezikód interpretovaný pomocí virtuálního stroje Javy, Java Vitrual Machine (JVM), se vyznačuje nezávislostí na architektuře počítače nebo zařízení. Program pak může pracovat na libovolném počítači nebo zařízení, který má k dispozici tento interpret. Interpret ovšem výrazně zpomalí běh programu. Proto není v novějších verzích mezikód Javy přímo interpretován, ale je před prvním svým provedením dynamicky zkompilován do strojového kódu daného počítače (tzv. just in time compilation - JIT). Tato vlastnost zásadním způsobem zrychlila provádění programů v Javě, ale výrazně zpomalila start programů. Moderním trendem se v Javě stávají technologie zvané HotSpot compiler, které mezikód zpočátku interpretují a na základě statistik získaných z této interpretace později provedou překlad často používaných částí do strojového kódu včetně dalších dynamických optimalizací, jako je např. inlining krátkých metod apod.
- Robustnost. Java obsahuje některá omezení, jež mají za důsledek znemožnění použití některých programátorských konstrukcí, které bývají častou příčinou chyb (např. správa paměti, příkaz goto, používání ukazatelů). Ovšem právě tato omezení jsou důležitá pro psaní vysoce spolehlivého softwaru. Java používá tzv. silnou typovou kontrolu, což znamená, že veškeré používané proměnné musí mít definovaný svůj datový typ. Správa paměti je realizována pomocí automatického Garbage collectoru, který automaticky vyhledává již nepoužívané části paměti a uvolňuje je pro další použití. Jedná se o další z vlastností, jež výrazným způsobem zvyšují efektivitu programátorské práce a robustnost, ale podílejí se na celkovém zpomalení běhu samotného programu. Problém je v současnosti řešen využitím co nejefektivnějších algoritmů pro správu paměti. Paměť se rozdělí na více částí, a v každé se používá jiný algoritmus pro garbage collection. Objekty jsou pak mezi těmito částmi přesunovány podle délky svého života.
- Bezpečnost. Programovací jazyk Java obsahuje spoustu obraných mechanizmů proti napadení aplikací běžících v síťovém prostředí škodlivým kódem. Jedná se o jakousi zabudovanou ochranu proti útočníkům na sítích.
- Nezávislost na architektuře. Díky tomu, že jsou aplikace psány pro virtuální stroj Javy a ne pro konkrétní operační systém, jsou tyto aplikace teoreticky zcela přenositelné mezi jednotlivými architekturami. Jedinou zásadní podmínkou pro jejich správný běh je přítomnost odpovídajícího virtuálního stroje na dané platformě. Jednotlivé platformy se liší v neposlední řadě také svým vzhledem, kterému se aplikace též sama přizpůsobí.
- Přenositelnost. Vedle zmíněné nezávislosti na architektuře je jazyk nezávislý i co se týká vlastností základních datových typů. V jazyce Java je například explicitně určena vlastnost a velikost každého z primitivních datových typů. Přenositelností ale chápeme pouze přenášení v rámci jedné platformy Javy (např. J2SE). Při přenášení

mezi platformami Javy (ME, SE, EE) musíme brát na vědomí, že platforma určená pro jednodušší zařízení většinou nepodporuje všechny funkce dostupné na platformě pro složitější zařízení. Kromě toho může daná platforma Javy definovat některé vlastní třídy, které doplňují nějakou speciální funkčnost nebo nahrazují třídy vyšší platformy, které jsou pro nižší platformu příliš komplikované.

- Výkonnost. Java se dá i přes své drobné potíže s rychlostí spouštěných programů označit za velice výkonný jazyk. Virtuální stroj má zabudovány mechanismy, jež se starají o překlad jen aktuálně využívané části zdrojového kódu, a tak zvyšují celkovou rychlost a efektivitu.
- Víceúlohovost. Jazyk Java podporuje zpracování vícevláknových aplikací.
- Dynamika. Knihovny Javy mohou být modifikovány a rozšiřovány také za běhu samotné aplikace. To dovoluje měnit prostředí aplikace bez přerušení jejího běhu. Knihovny může takto měnit nejen samotná aplikace, ale též některý externí zdroj.
- Elegance. Tato vlastnost zahrnuje spoustu jiných dalších vlastností, jako jsou například čitelnost, skvělé ošetření výjimek apod. Eleganci jazyka Java ale nejlépe dokládá jeho masová oblíbenost, neotřesená ani nástupem jazyka C# z dílny Microsoftu.

3.3.5.3 Nevýhody

I přes nespornou propracovanost a vysokou efektivitu má práce s programovacím jazykem Java své nevýhody. Především se jedná o virtuální stroj, který sice dokáže výborně zajistit přenositelnost, ovšem také radikálním způsobem zpomaluje spouštění programů. Tato nevýhoda je patrná zejména při srovnání jazyka Java s jinými jazyky, které, na rozdíl od ní, provádějí tzv. statickou kompilaci (např. C++). V prostředí Javy se musí program nejprve přeložit a potom teprve spustit. Je však možnost využít mechanismů JIT a HotSpot, kdy se často prováděné nebo neefektivní části kódu přeloží do strojového kódu a program se zrychlí. Na zrychlení se také podílí nové přístupy ke správě paměti.

Ruku v ruce s pomalým spouštěním programů v Javě jde také větší paměťová náročnost při běhu způsobená nutností mít v paměti celé běhové prostředí.

3.3.6 Jazyk C#

3.3.6.1 Základní charakteristika

C# je vysokoúrovňový objektově orientovaný programovací jazyk z dílny firmy Microsoft. Od samotného počátku je provázán s platformou .NET, pro kterou byl vyvíjen. Následně byl schválen standardizačními komisemi ECMA a ISO. Microsoft při návrhu programovacího jazyka C# vycházel z koncepce jazyků C++ a Java, a tedy se jedná o dalšího, i když nepřímého, potomka jazyka C. Jazyk C# nalezl široké uplatnění snad ve všech oblastech programování, ovšem jeho nasazení se prozatím pohybuje jen v nízkých číslech (viz. graf v podkapitole 3.2). C# se využívá hlavně k tvorbě databázových programů, webových aplikací, webových služeb, formulářových aplikací ve Windows apod. Podstatným nástrojem pro práci v jazyce C# je velmi globální a propracované řešení Microsoft Visual Studio, momentálně ve verzi 2005. Pokud se bavíme o jazyce C#, je podle mě na místě říci několik málo slov také o Visual Studiu. To umožňuje programátorovi jednoduše a intuitivně provádět velké množství i složitých postupů a vede vývojáře k nezvykle vysoké efektivitě práce. Visual Studio pracuje nejen s programovacím jazykem C#, ale také s ostatními jazyky založenými na společné platformě .NET. Dále podporuje psaní kódu např. v jazycích C, C++, ale také mnoha jiných. Visual

Studio 2005 obsahuje také Microsoft SQL Server Express pro jednoduchou práci s databázemi. Z dalších jeho funkcí může vývojář ocenit automatické generování nákresů ERD v UML nebo možnou instalaci rozličných doplňků, jako např. výborného systému Serena Change Man Dimension, neocenitelného při větších projektech. Ten umožní vysoce kvalitní přehlednost verzí kódů, revizí, správu procesů vývoje pomocí tzv. „Change dokumentů“ apod. Troufnu si říci, že i když je programovací jazyk Java asi „nejblíže mému srdci“, zatím se nemá šanci vyrovnat obrovské efektivitě jazyka C# v kombinaci s Visual Studiem. Samozřejmě, že v některých ohledech nelze C# s Javou srovnávat, zejména pokud se jedná o specializované přístroje, jako jsou mobilní telefony apod. Zde vede Java na plné čáře a předpokládám, že se situace v brzké době nezmění. Problémem rozšíření C# je bezesporu také fakt, že pochází právě z dílny firmy Microsoft. To představuje asi největší překážku i přes to, že již existují první verze virtuálního stroje .NET i pro Uniové operační systémy.

3.3.6.2 Vlastnosti jazyka

Definice podle standardu ECMA pro C# vypadá takto:

- C# je jednoduchý, moderní, obecně účelový a objektově orientovaný programovací jazyk.
- Jazyk a jeho implementace poskytuje podporu pro principy softwarového inženýrství jako jsou: Hlídání hranic polí, detekce použití neinicializovaných proměnných a automatický garbage collector. Důležitá je zejména jeho robustnost, trvanlivost a programátorská produktivita.
- Jazyk C# je vhodný pro vývoj softwarových komponent distribuovaných v různých prostředích.
- Přenositelnost zdrojového kódu je velmi důležitá, obzvláště pro ty programátory, kteří využívají programovací jazyky C a C++.
- Důležitá je též mezinárodní podpora.

3.3.7 PHP

3.3.7.1 Základní charakteristika

PHP (rekurzivní zkratka PHP: Hypertext Preprocessor, „PHP: Hypertextový preprocesor“, původně Personal Home Page) je skriptovací programovací jazyk, určený především pro programování dynamických internetových stránek. PHP skript lze umístit přímo do struktury jazyka HTML, XHTML či WML, což je velmi výhodné pro tvorbu webových aplikací, ovšem mnohem přehlednější je obě složky oddělit pro lepší orientaci programátora ve zdrojovém kódu. PHP lze ovšem také použít k tvorbě konzolových a desktopových aplikací.

PHP skripty jsou prováděny na straně serveru. K uživateli je přenášen až výsledek jejich činnosti. Syntaxe jazyka vychází hned z několika programovacích jazyků. Nalezneme v ní prvky z jazyků Perl, C, Pascal nebo Java. PHP je nezávislý na platformě, a tedy skripty PHP fungují bez úprav na mnoha různých operačních systémech. Skriptovací jazyk PHP dokáže díky velkému množství knihovních funkcí efektivně zpracovávat text a grafiku, pracovat se soubory a přistupovat k většině databázových serverů jako například k MySQL, ODBC, Oracle, PostgreSQL, MSSQL. Jazyk PHP podporuje spoustu internetových protokolů. Jako příklad mohu uvézt HTTP, SMTP, SNMP, FTP, IMAP, POP3, LDAP, a mnoho dalších.

Velká obliba PHP plyne zejména z faktu, že vychází z několika masově rozšířených a velmi oblíbených programovacích jazyků. Navíc byla koncepce syntaxe zpracována jednoduše a nabízí vývojářům rozsáhlé možnosti svého využití.

Programátoři v PHP často mluví o tzv. „svaté trojici“. Jedná se o kombinaci PHP s databázovým serverem MySQL nebo PostgreSQL a webovým serverem Apache. Toto řešení je zcela zdarma i přes to, že nabízí rozsáhlé možnosti. Další vžitou zkratkou je LAMP – tedy spojení Linuxu, Apache, MySQL a PHP nebo Perlu.

Nejnovější verze PHP, tedy verze 5, přinesla výrazné zlepšení objektově orientovaného přístupu k programování. Velká část této koncepce se velice podobá Javě.

3.3.7.2 Historie

Již v kapitole o operačním systému Linux jsem v souvislosti s jeho vznikem mluvil o fenoménu frustrace. Zmiňoval jsem tehdy, že opravdu značné množství velice kvalitních a propracovaných softwarových řešení vznikla právě v jejím důsledku. A PHP není v tomto ohledu žádnou výjimkou. Od roku 1994 je PHP jedním z nejpoužívanějších způsobů tvorby dynamicky generovaných WWW stránek. Rasmus Lerdorf jej vytvořil pro svou osobní potřebu přepsáním z Perlu do jazyka C. Sada skriptů byla vydána ještě v témže roce pod názvem Personal Home Page Tools, zkráceně PHP. Stejný autor ještě v roce 1994 spojil PHP s rovněž svým programem Form Interpreter. Jako výsledek vzniklo PHP/FI 2.0.

Dále na PHP pracovali Zeev Suraski a Andi Gutmans, kteří v roce 1997 přepsali parser jazyka a položili tak základ pro PHP verze 3. Současně byl název změněn na dnešní podobu PHP hypertext procesor. PHP3 se objevilo poprvé v roce 1998. Od předchozích verzí značně zrychlilo a také se rozšířilo o řadu nových funkcí. Významným krokem také bylo jeho zprovoznění pod operačním systémem Microsoft Windows.

Rok 2000 je pro PHP ve znamení nové verze 4, a o čtyři roky později pak vychází verze 5 s vylepšeným objektovým přístupem, podobným jazyce Java. Zatím poslední verzí je PHP 5.2.1 z roku 2007.

3.3.8 Perl

Jako poslední programovací jazyk mého rychlého přehledu jsem vybral interpretovaný skriptovací jazyk Perl, který mě zaujal zejména přístupem svého autora k sobě samému i k jazyce Perl. Autor Larry Wall totiž považuje sebe za umělce a svůj výtvor, jazyk Perl, za umělecké dílo. Sám pak tvrdí, že dobrý programátor má být líný, netrpělivý a arogantní. Na Wikipedii jsem dohledal příklad skriptu v jazyce Perl, který právě podporuje a demonstruje tvrzení Larryho Walla. Jedná se o skript, který vypíše poměrně pomalu slova: „*Just another Perl / Unix hacker*“. Za tím účelem vytvoří 32 paralelních procesů, které složitě synchronizuje pomocí rour. Zde přikládám onen skript:

```
@P=split//,".URRUU\c8R";@d=split//,"\nrekcah xinU / lreP rehtona
tsuJ";sub p{@p{"r$p","u$p"}=(P,P);pipe"r$p","u$p";++$p;($q*=2)+=$f=!
fork;map{$P=$P[$f^ord($p{$_})&6];$p{$_}=/^$P/ix?$P:close$_}keys%p}p;
p;p;p;p;map{$p{$_}=~/^[P.]/&&close$_}%p;waituntil$?;map{/^r/&&<$_>}%
p;$_=$d[$q];sleep rand(2)if/\S/;print
```

Obr. 3.3 – Ukázkový skript psaný v jazyce Perl

3.3.8.1 Základní charakteristika

Jak již bylo řečeno, Perl je interpretovaný programovací jazyk vytvořený Larry Wallem v roce 1987. S rozvojem internetu se Perl stal velmi populárním nástrojem pro tvorbu CGI skriptů.

Perl zahájil svou éru jako skriptovací jazyk, který sloužil jako náhrada jazyka AWK a interpretru sh. Největšího rozšíření dosáhl ve verzi 4 z roku 1991. Verze 5 pak přinesla zejména výkonné datové struktury a objektový přístup. Perl získává v poslední době stále vyšší oblibu v rozličných odvětvích informatiky. Jako typický příklad může sloužit jeho rozšíření v bioinformatice. Uznávaný časopis PC Magazín zařadil Perl v roce 1998 mezi finalisty soutěže Technical Excellence Award v kategorii Development Tool.

Díky tomu, že se Larry Wall při konstrukci jazyka Perl řídil heslem: „Dá se to udělat více způsoby“ („There's more than one way to do it“), jazyk umožňuje psát krátké programy jednoduše a rychle. Přitom nic nebrání v psaní i rozsáhlých aplikací. Jeden ze způsobů je přitom obvykle velmi stručný, takže Perl získal nezaslouženou pověst jazyka, ve kterém se tvoří nesrozumitelný a neudržovatelný kód. Tato kritika ale není oprávněná, Perl je vhodný k řešení malých i velkých problémů. Schopnosti a nástroje, které se používají u velkých projektů, lze použít i v krátkých skriptech.

3.3.8.2 Historie

Jazyk Perl nedisponuje příliš dlouhou a rozmanitou historií. Zajímavým rokem ale byl rok 2000, ve kterém se konala schůze vývojářů Perlu. Na ní se rozhodlo o projektu vývoje virtuálního stroje jména Parrot pro Perl 5 a 6, Python a Rubi a započala tvorba nové verze jazyka Perl 6. Betaverze Perlu 6 by se mohla objevit již v roce 2008 a slibuje rozsáhlé změny jak v syntaxi, tak ve výkonu.

3.3.8.3 Důležité vlastnosti

Jako u každého interpretovaného jazyka představuje hlavní výhodu určitě rychlý vývoj bez nutnosti kompilace a linkování. Efektivita Perlu pak také dovoluje psát programy s menším počtem řádků kódu, které se zabezpečují mnohem lépe, než je tomu u kompilovaných jazyků. Také funkční výbava Perlu je poměrně slušná. K dispozici je přes 4000 volně dostupných modulů třetích stran skrz Comprehensive Perl Archive Network CPAN. Pojmenování, kategorizace, dokumentace, testování a instalace modulů jsou standardizovány. Moduly zpřístupňují prakticky veškerá dostupná rozhraní a knihovny. Např. rozhraní Perlu pro databáze (DBI) podporuje databáze Oracle, Sybase, PostgreSQL, MySQL, SQLite a modul CGI.pm umožňuje lehce pracovat s HTML formuláři. Pro tak objemné množství funkcí je nutná existence technické dokumentace a literatury. Perl v tomto ohledu jistě netrpí nedostatky. K dispozici má programátor také řadu konferencí „newsových“ skupin apod.

Pokud bychom se zajímali o administraci kódu, Perl podporuje dynamickou práci s pamětí a pokročilé datové typy, jakým je například asociativní pole, neboli hash. Perl využívá také práce s ukazateli na statické, dynamické i anonymní datové struktury. Ty podporují procedurální i objektově orientované programování.

Práce s textem a značkovacími jazyky je v Perlu také jednoduchá. Perl využívá regulárních výrazů, podporuje znakovou sadu Unicode.

Pro optimalizace zde najdeme také několik užitečných funkcí. Perl umožňuje tvorbu dynamického kódu a dynamické volání procedur. Také zde existuje možnost inline využití jazyka C či C++.

Perl podporuje automatické zavádění. Pokud funkce neexistuje pak Perl umožňuje spuštění definovaného kódu.

Jazyk Perl je vhodný pro práci s webem a se zakódovanými webovými daty, mezi něž patří transakce u elektronického obchodování. Při použití Perlu jako součásti webových serverů může dojít k nárůstu výkonu až o 2000%. Na serveru Apache se dá nasadit se stejnými výhodami, jako PHP.

Interpret Perlu může být vložen (embedded) do ostatních systémů, #include <Perl.h>.

Perl je svobodný software, licencovaný pod Artistic License nebo GNU General Public License.

Stejně jako každý programovací jazyk, není ani Perl ušetřen drobných problémů, jež mohou nastat v souvislosti s jeho nasazením. Nedisciplinovaný programátor může snadno vytvářet nesrozumitelný kód, v některých aplikacích se může projevit neefektivnost interpretru ve srovnání s kompilovanými jazyky, zejména spotřeba paměti, adresu odkazů není možno uložit do asociativního pole (hash), u kruhových odkazů je nutné vytvořit destruktor ručně atd.. Výhody skriptovacího jazyka Perl se ale strmě tyčí nad jeho několika málo nevýhodami a tedy jde skutečně o velice efektivní a univerzální programovací prostředek.

4 Abstraktní datové typy

V této kapitole se zaměřím na objasnění problematiky spojené s abstraktními datovými typy a následně podrobněji proberu tři vybrané abstraktní datové typy, jež jsem použil při tvorbě domácích úloh.

4.1 Vymezení pojmu abstraktního datového typu

Již v úvodu mé bakalářské práce se zamýšlím nad tématem správného a hlubokého porozumění problematiky algoritmizace. Tvrdím, že opravdu dobrý programátor prostě musí dostatečně porozumět tématu tvorby algoritmů nad datovými strukturami, tedy práci s abstraktními datovými typy, jejich návrhu a implementaci. Nyní bych rád upřesnil termín abstraktní datový typ a zamyslel se nad podměty, jež v minulosti vedly k jeho vzniku.

Nejprve uvedu definici abstraktního datového typu převzatou z [1]:

Abstraktní datový typ (ADT) je definován množinou hodnot, jichž mohou nabývat prvky tohoto typu a množinou operací, definovaných nad tímto typem. Abstraktní datový typ representuje třídu prvků se shodnými vlastnostmi. Jednomu konkrétnímu prvku takového typu říkáme někdy “proměnná daného typu”, “prvek daného typu”,“instance daného typu”, “výskyt daného typu”, nebo také "abstraktní datová struktura daného typu" - ADS.

V informatice se velice často skloňují slova efektivita a výkonnost, která ve většině případů přímo souvisí s pojmem abstrakce. Vysvětlení je relativně jednoduché. Počítač je sice stroj, který pracuje na základě posloupností elektrických impulzů různé napěťové úrovně, ale jeho tvůrce, člověk, je z masa a kostí. A i přes fakt, že člověk rozumí pochodům probíhajícím uvnitř počítače, bylo by pro něj velmi obtížné, ne-li zcela nemožné, vyvíjet kvalitní a součastně i složitý software jen pomocí strojového kódu nebo instrukcí assembleru, tvořících jen nepatrnou míru abstrakce právě nad strojovým kódem.

Lidé si po desetiletí tento fakt dobře uvědomují, a proto se tvůrci programovacích jazyků pokouší vytvářet takové koncepce, které by právě díky určitému stupni abstrakce dokázaly programátorovi nabídnout lepší ztotožnění s psaným kódem tak, aby pro programátora nepředstavoval jen shluky čísel, ale také se přiblížil lidskému myšlení.

První známky takového přístupu můžeme zaregistrovat již u programovacího jazyka Algol 60, který je prapředkem takových programovacích jazyků, jak jsou například Basic, Pascal nebo C.

Nejprve tedy byla datová struktura méně používaným prostředkem programátorů. Svůj vliv měly časté obavy z nízké efektivity práce s takovou strukturou, neodpovídající množství operací, které mohly být využívány, a vysoká spotřeba paměti. Výzkum výpočetní složitosti algoritmů ovšem dokázal, že opak může být pravdou. Zjistilo se, že existuje vazba mezi složitostí algoritmu a použitím vhodné datové struktury. Je tedy možné implementovat efektivnější a v důsledku rychlejší algoritmus i při použití složitější datové struktury. Pro takovou realizaci je ale nutné provést ještě jeden krok. Datová struktura se musí odpoutat od některých svých vlastností, které by mohly ztěžovat její implementaci. Jedná se zejména o její reprezentaci v paměti a informace popisující podrobnou technickou strukturu operací, jež jsou nad ní prováděny. Místo toho se soustředění musí zaměřit na použití operací nad datovými strukturami jako celkem a na reprezentaci vnějšího chování datové struktury z hlediska toho, co sama reprezentuje. Tímto způsobem se dojde k abstraktní datové struktuře a abstraktnímu datovému typu, jako její třídě.

Abstraktní datový typ tedy vznikne vybráním pro nás nejdůležitějších vlastností, zaměřením se na poslání datové struktury a odpoutáním se od její technické stránky. To má za důsledek mnohem lepší pochopení zdrojového kódu a následně také diametrálně velký pokrok ve zvýšení efektivity psaní programů, ale též ve výkonu takové aplikace. Díky standardizaci abstraktních datových typů pak dosáhneme universálnějšího, a tedy také snadnějšího psaní programů.

4.2 Příklady abstraktních datových typů

Pro tuto kapitolu jsem vybral 3 abstraktní datové typy, které jsem měl za úkol naimplementovat v domácích úlohách: Jsou to:

- Dvousměrně vázaný kruhový lineární seznam,
- zásobník,
- orientovaný graf.

4.2.1 Lineární seznamy

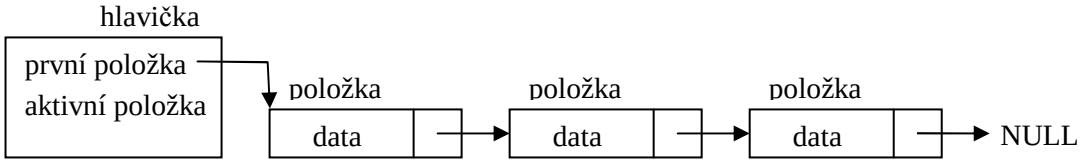
Lineární seznam se řadí mezi abstraktní datové typy (ADT). Je nejčastěji využíván pro tvorbu databází, ale obecně se jedná o jeden z nejpoužívanějších prostředků snad ve všech programovacích disciplínách. Může být totiž využit jako základ i jiných abstraktních datových typů, jako je například zásobník nebo fronta. Znalost jeho struktury a využití patří mezi elementární znalosti na poli informačních technologií.

Existují v zásadě 3 typy lineárních seznamů:

- Jednosměrně vázaný lineární seznam,
- dvousměrně vázaný lineární seznam,
- kruhový lineární seznam, který může být rovněž jedno i dvousměrně vázaný.

4.2.1.1 Jednosměrně vázaný lineární seznam

Jednosměrně vázaný lineární seznam se sestává z hlavičky, 0 až n položek a funkcí implementujících operace nad danou abstraktní datovou strukturou.



Obr. 4.1 – Schématický nákres jednosměrně vázaného lineárního seznamu se třemi položkami

Hlavička

Hlavička obsahuje ukazatel na první položku lineárního seznamu, který je datového typu položka.
Pokud seznam neobsahuje žádné položky, měl by se nacházet v tzv. „inicializovaném“ stavu.
To znamená, že ukazatele by měly obsahavat hodnotu NULL a ne jakoukoli nedefinovanou hodnotu.

Položka

Položka je datová struktura, která obsahuje ukazatel na následující položku seznamu, která je datového typu položka, a užitečná data jakéhokoli datového typu. Pokud se daná položka v seznamu vyskytuje jako poslední, ukazatel na další položku by měl obsahovat hodnotu NULL. Jednotlivé položky jsou alokovány v operační paměti dynamicky, a tedy v žádném případě nelze

počítat s tím, že by se nacházely právě hned vedle sebe. Počet položek seznamu je omezen pouze velikostí operační paměti daného počítače.

Operace nad datovou strukturou

Množina operací lze libovolně modifikovat podle potřeby, ovšem můžeme se podívat na některé základní a nejčastěji využívané operace:

- Inicializace. Uvádí neinicializovaný seznam do inicializovaného stavu před prvním použitím. Nejedná se o operaci, která by odstranila položky seznamu.
- Vyprázdnění seznamu. Operace korektně uvolní paměť všech položek seznamu a uvede ho do inicializovaného stavu.
- Vložení položky na začátek lineárního seznamu.
- Vymazání první položky. Operace korektně uvolní paměť první položky seznamu a odstraní ji ze zřetězení. Pokud je mazaná položka v seznamu poslední, operace seznam inicializuje.
- Ověření prázdnosti seznamu. Operace vrací logickou hodnotu „pravda“, pokud je seznam prázdný, jinak vrací „nepravdu“.
- Zjištění užitečných dat první položky, pokud seznam není prázdný.

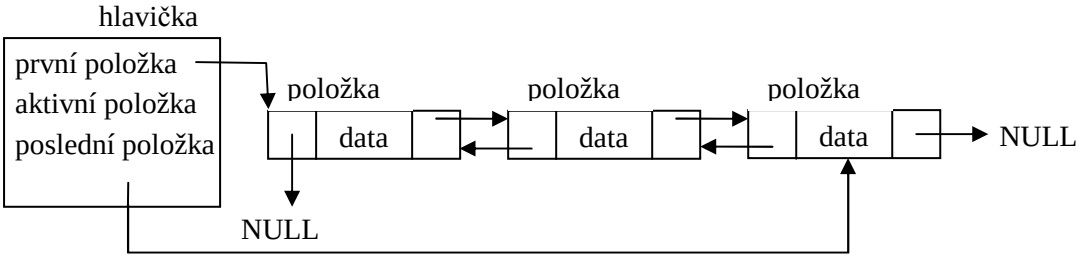
Pokud zavedeme do struktury hlavičky seznamu ještě ukazatel na aktivní prvek (což je v případě lineárních seznamů téměř samozřejmostí), množina základních operací se může dále rozšířit. Operace fungují analogicky k výše uvedeným.

- Nastavení aktivity na první položku.
- Zjištění užitečných dat aktivní položky.
- Odstranění položky za aktivní položkou.
- Posun aktivity na následující položku.
- Přidání nové položky za aktivní položku. .
- Ověření aktivity seznamu.
- Změna užitečných dat aktivní položky.

Pokud implementujeme funkčnost operací nad datovou strukturou, vždy musíme být pozorní ve smyslu krajních stavů, do kterých se může datová struktura dostat. Musíme důsledně uvážit všechny možnosti a stavy, které by mohly při zanedbání jejich ošetření vést k chybám a nekonzistencím datové struktury.

4.2.1.2 Dvousměrně vázaný lineární seznam

Pokud mám popsat strukturu a vlastnosti dvousměrně vázaného lineárního seznamu, mohu velice snadno vyjít z předchozí podkapitoly (4.2.1.1). Dvousměrně vázaný lineární seznam má totiž stejné vlastnosti, jako jednosměrný, rozšířené díky druhému směru provázanosti položek. Situaci asi nejlépe vystihne jednoduchý nákres:



Obr. 4.2 – Schématický nákres dvousměrně vázaného lineárního seznamu se třemi položkami

Z obrázku je asi více něž patrné, že rozdíl oproti jednosměrnému lineárnímu seznamu tkví v tom, že každá položka obsahuje navíc ukazatel datového typu položka na předchozí položku. U první položky, analogicky k poslední položce, obsahuje ukazatel na předchozí položku hodnotu NULL. Dále je také logické rozšířit hlavičku seznamu o ukazatel na poslední položku seznamu.

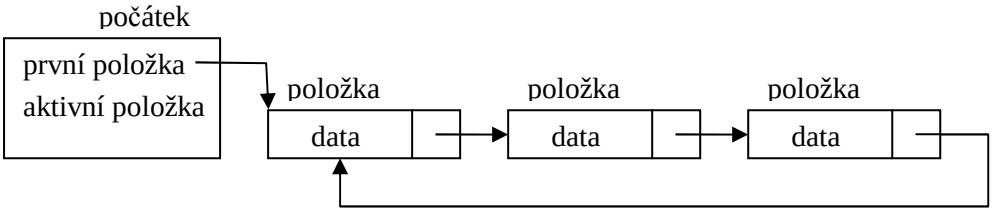
Nejdůležitější vlastností dvousměrně vázaného lineárního seznamu oproti jednosměrnému je tedy možnost procházet položky oběma směry. Tato vlastnost umožňuje zvětšit množinu základních operací nad tímto abstraktním datovým typem o operace:inverzní k směrově závislým operacím. Níže následuje přehled některých z nich:

- Vložení nové položky na konec seznamu.
- Nastavení aktivity na poslední položku seznamu.
- Získání užitečných dat poslední položky seznamu.
- Odstranění poslední položky seznamu.
- Odstranění položky před aktivním prvkem.
- Vložení nové položky před aktivní prvek.
- Posun aktivity na předchozí položku.

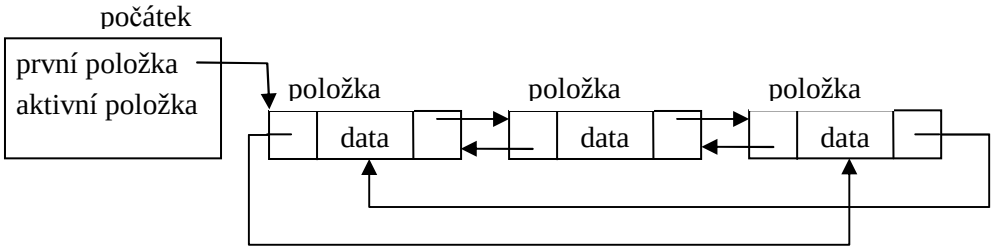
Dvousměrně vázaný lineární seznam tedy nabízí nespočet různých možností za cenu poměrně zanedbatelného zvýšení nároků na operační paměť oproti seznamu jednosměrně vázanému. Jedná se o velikost ukazatele na položku vynásobenou počtem položek plus velikost ještě jednoho ukazatele na poslední položku seznamu.

4.2.1.3 Kruhový lineární seznam

Kruhový lineární seznam může být opět implementován jako jednosměrný či dvousměrný (viz. podkapitoly 4.2.1.1 a 4.2.1.2). Rozdíl oproti předchozím typům lineárních seznamů spočívá v tom, že kruhový lineární seznam nemá počátek, ani konec. Abychom ale mohli se seznamem efektivně zacházet, je nutné vybrat určitý prvek a ten označit. Ono označení je opět datovou strukturou podobající se hlavičce lineárního jednosměrného seznamu. Obsahuje opět ukazatel na aktivní položku a ukazatel na položku, kterou chápeme jako výchozí pro operace. Situaci nejlépe vystihují následující nákresy:



Obr. 4.3 – Nákres jednosměrně vázaného kruhového lineárního seznamu se třemi položkami



Obr. 4.4 – Nákres dvousměrně vázaného lineárního kruhového seznamu se třemi položkami

Síla tohoto seznamu tedy vychází právě z jeho kruhovosti, tedy možnosti procházet seznam cyklicky bez jakýchkoli dalších mechanizmů, které by kruhovost simulovali. Oproti dvousměrně vázanému lineárnímu seznamu zabírá dvousměrně vázaný lineární seznam dokonce o velikost jednoho ukazatele na položku méně paměti a nabízí pohodlnější přístup programátorovi.

Množina základních operací se v zásadě neliší od předchozích typů seznamů. Rozdíly se ovšem samozřejmě objevují v jejich implementaci. Protože je implementace domácí úlohy využívající kruhový dvousměrně vázaný lineární seznam součástí zadání této bakalářské práce, naleznete podrobný popis celé této datové struktury v příloze na CD v adresáři *domaci_ulohy/c207* v souboru *c207.h* a podrobný výčet a popis všech operací v souboru *c207.c*.

4.2.2 Zásobník

„Zásobník je v oblasti informatiky důležitým pojmem, zvláště jedná-li se o systémový software.“[3] Nejčastěji bývá implementován pomocí statického pole nebo lineárním seznamem, v našem případě dvousměrně vázaným lineárním seznamem. Já se ovšem nyní budu zabývat popisem zásobníku jako abstraktního datového typu obecně a ne v závislosti na jeho konkrétní implementaci.

Zásobník je také znám pod anglickou zkratkou „LIFO“, což znamená „Last in, first out“. Tato zkratka vystihuje chování zásobníku k položkám v něm uskladněných. Položka, která je jako poslední do zásobníku vložena, je následně jako první ze zásobníku odstraněna. Implementace operací nad zásobníkem by obecně neměla dovolovat manipulaci či čtení jiného prvku, než toho, který se nachází na vrcholu zásobníku.

Množina operací nad zásobníkem je tedy v zásadě přesně definována:

- Inicializace zásobníku před jeho prvním použitím.
- Vložení prvku na vrchol zásobníku.
- Odstranění prvku z vrcholu zásobníku.
- Čtení užitečných dat prvku z vrcholu zásobníku.
- Test na prázdnost zásobníku.

Množina operací bývá často doplňována o operaci, která současně vrací hodnotu prvku na vrcholu zásobníku a tento prvek následně odstraňuje. Pokud implementujeme zásobník pomocí statického pole, nabízí se ještě naimplementovat také operaci testu na plnost zásobníku.

Jednotlivé operace mohou v případě určitých stavů zásobníku způsobit vyvolání chyby. Většinou se jedná o operaci čtení či odstranění prvku ze zásobníku, pokud je zásobník prázdný, nebo přidání prvku do plného zásobníku v případě jeho implementace statickým polem. Ovšem i přes tyto zažité standardy je nakonec celá implementace v rukou programátora. Podrobný popis této datové struktury implementované pomocí dvousměrně vázaného lineárního seznamu naleznete v příloze na CD v adresáři *domaci_ulohy/c208* v souboru *c208.h* a podrobný výčet a popis všech operací v souboru *c208.c*.

4.2.3 Orientovaný graf

Graf obecně lze označit za jednu z nejuniverzálnějších datových struktur. To dokládá také fakt, že se za teorií grafů ukrývá poměrně objemný matematický aparát a jeho k zkoumání je možné vyčlenit celý samostatný oddíl matematiky. V této technické zprávě bych se raději zabýval grafem jako datovou strukturou a podrobnější matematický výklad nechal na příslušných publikacích a učebnicích matematiky. „Tvrzení, že bez této datové struktury by se mnoho algoritmických problémů nedalo vůbec vyřešit, zdaleka není přehnané“.[3] V učebnicích algoritmiky pak zabírá

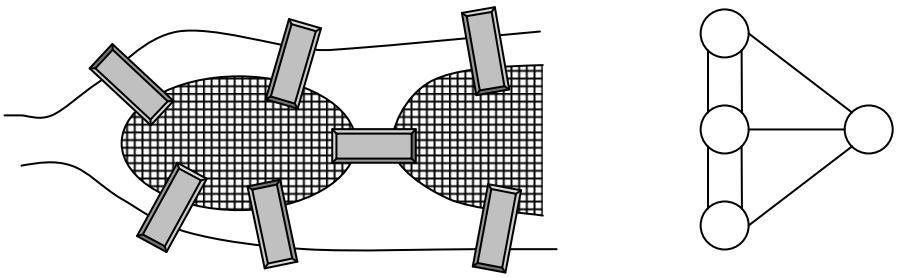
teorie grafů jedno z výsadních postavení. Z publikací mohu například doporučit [9]. Nejčastější využití nachází datová struktura grafu při řešení problémů umělé inteligence.

4.2.3.1 Definice orientovaného grafu

Orientovaný graf je dvojice (U, H), kde U je libovolná množina uzlů a H binární relace nad množinou U, které říkáme množina hran. Přitom platí, že existuje vždy isomorfismus mezi libovolnou binární relací a orientovaným grafem. Takovému grafu pak říkáme graf relace.

4.2.3.2 Historie grafů

Datová struktura grafu a její prvopočátky mají docela zajímavou historii, o které by bylo škoda se alespoň krátce nezmínit. Grafy se poprvé objevily v roce 1736 díky práci německého matematika L. Eulera. Občané města Königsberg tehdy řešili problém s okružní trasou kolem jejich náměstí. Ta totiž zahrnovala 7 mostů a občané se chtěli procházet po celém městě, ovšem přejít přitom každý most jen jedenkrát. Euler pro tento problém vymyslel grafovou strukturu, díky níž byl následně schopen občanům poradit. Paradoxně tak navrhnul složitou datovou strukturu, která při použití u většiny algoritmických úloh řeší zadání už svou podstatou. „Programátor, který dobře zná možnosti grafů, prakticky zdvojnásobuje své dovednosti účinného modelování řešených problémů“.[3]

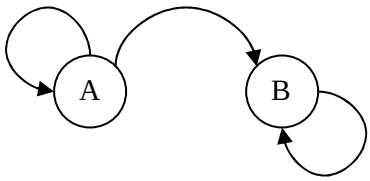


Obr. 4.5 – Nákres náměstí v Königsbergu a jeho grafová reprezentace

4.2.3.3 Implementace abstraktní datové struktury grafu

Struktura grafu lze naimplementovat mnoha způsoby jako jsou pole, seznamy a stromy. V tomto textu se zaměřím zejména na implementaci grafu pomocí seznamů, ale nejprve bych rád přiblížil graf pomocí reprezentace polem.

Reprezentace grafu ve statickém dvourozměrném poli se mi zdá jako nejvhodnější pro pochopení zákonitostí, jež problematiku použití grafů popisují. Takovéto pole tvoří tabulku sousednosti jednotlivých uzlů grafu. Pokud zavedeme konvenci, že řádky takového grafu budou představovat počáteční uzly hran grafu a sloupce uzly koncové, můžeme snadno pomocí vyplnění takovéto tabulky vyjádřit celou jeho strukturu. Vyplněné buňky pak značí hranu mezi dvěma uzly. Pokud bude pole číselného typu, dá se také využít rovnou pro ohodnocení jednotlivých hran. Na obrázku 4.6 je vyjádřen pomocí tabulky sousednosti graf obsahující dva uzly A, B a tři hrany směřující z uzlu A zpět do uzlu A, z uzlu A do uzlu B a z uzlu B zpět do uzlu A.

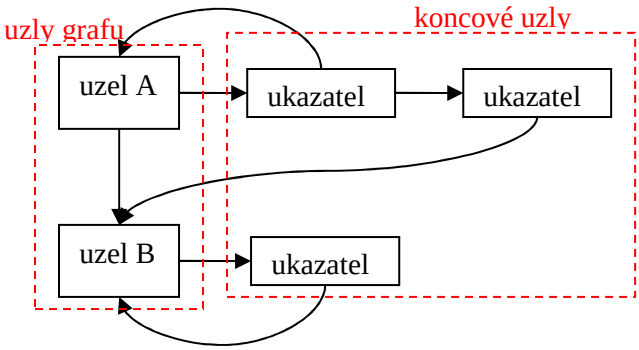


	A	**B**
A	1	1
B		1

Obr. 4.6 – Orientovaný graf a jeho tabulka sousednosti.

Výhodou takovéto reprezentace je právě její jednoduchost a možnost použít ji téměř v libovolném programovacím jazyce. Nevýhodou je neměnná velikost, což je ve většině případů naprosto neakceptovatelné.

Dále se budeme zabývat již jen dynamickou reprezentací grafu pomocí slovníku uzlů. Ten je realizován pomocí nejlépe dvousměrně vázaných lineárních seznamů. První seznam obsahuje všechny uzly. Každý uzel pak obsahuje ukazatel na další lineární seznam, obsahující ukazatele na případné koncové uzly. Hrana je tedy definována počátečním uzlem z prvního lineárního seznamu uzlů a ukazatelem na koncový uzel v dalším lineárním seznamu, který vychází z každého uzlu v seznamu. Tuto zamotanou situaci asi nejlépe vysvětlí obrázek níže. Ten využívá grafu z obrázku 4.6.



Obr. 4.7 – Slovník uzlů orientovaného grafu.

4.2.3.4 Operace nad orientovaným grafem

Množina operací tohoto abstraktního datového typu implementovaného dvousměrně vázaným lineárním seznamem je skutečně rozsáhlá. Nicméně alespoň pro vytvoření a modifikaci grafu je potřeba některých základních operací. Jedná se zde zejména o operace odvozené z množiny operací nad lineárními seznamy, které jsou vhodně modifikovány, popřípadě kombinovány.

- Inicializace dosud neinicializovaného grafu.
- Odstranění všech uzlů a hran grafu.
- Vložení nového uzlu do grafu.
- Odstranění uzlu z grafu.
- Vytvoření nové hrany mezi dvěma uzly.
- Odstranění hrany.
- Nalezení uzlu v grafu.
- Nalezení hrany v grafu.

Kromě těchto základních operací můžeme jejich množinu rozšířit ještě o některé pomocné operace, jako například o test na prázdnost grafu a jiné.

V domácí úloze jsem naimplementoval ještě tři operace, které se v souvislosti s grafy poměrně často využívají. Jsou to součet grafů, kompozice grafů a Floyd-Warshallův algoritmus. Podrobný popis datové struktury grafu implementované pomocí dvousměrně vázaných lineárních seznamů a operací nad ní naleznete v příloze na CD v adresáři *domaci_ulohy/c601* v souboru *c601.h* a podrobný výčet a popis všech operací v souboru *c601.c*. Přesto, že jsem poměrně podrobně popsal ve zdrojových kódech veškerou implementaci, považuji za vhodné rozvézt více toto téma i zde v technické zprávě. Zejména chci přidat matematický podklad, který ve zdrojových textech chybí.

Součet grafů

Jedná se o poměrně jednoduchou operaci nad grafy. Mějme dva grafy G_1 a G_2 se stejnými množinami uzlů a jakýmikoli množinami hran. $G_1 = (X, \Gamma_1)$ a $G_2 = (X, \Gamma_2)$. Výsledkem součtu grafů je následně graf G_3, který obsahuje množinu uzlů stejnou, jako u sčítaných grafů a množinu hran, jež je součtem množin hran obou grafů. $G_3 = (X, \Gamma_1 \cup \Gamma_2)$. Jinak řečeno, výsledný graf G_3 obsahuje všechny hrany grafů G_1 a G_2.

Kompozice grafů

Kompozice se opět provádí nad dvěma grafy G_1 A G_2 se stejnými množinami uzlů a jakýmikoli množinami hran. Výsledkem kompozice grafů je následně graf G_3, který obsahuje množinu uzlů stejnou, jako u výchozích grafů a množinu hran (x, y) takových, že (x, z) patří do G_1 a (z, y) do G_2. Matematický zápis vypadá takto:

$$G_3 = G_1 \circ G_2 = (X, \Gamma_3 = \{(x, y) \mid \exists z \in X : (x, z) \in \Gamma_1 \wedge (z, y) \in \Gamma_2\})$$

Roy-Warshallův algoritmus

Hned od počátku návrhu domácí úlohy práce s orientovaným grafem mi vrtala hlavou myšlenka, že bych mohl jemně zabrousit do umělé inteligence. Ovšem počítal jsem s tím, že tento předmět si studenti okusí jen o pouhý semestr později a nerad bych je příliš zatěžoval v už tak nabytém třetím semestru. Pak jsem ale objevil v knihách Roy-Warshallův algoritmus. Jedná se o velice silný, ale na implementaci jednoduchý způsob, jak nalézt všechny uzly přístupné z určitého uzlu grafu bez použití umocňování grafu. Algoritmus je tedy jinými slovy určen k výpočtu přechodného uzávěru grafu.

Matematický zápis

Mějme graf $G = (X, \Gamma)$. Algoritmus vychází z operace $\Theta_k = \Gamma \cup (y, z) \mid (x, k) \in \Gamma \wedge (k, y) \in \Gamma$. Zápis znamená, že pro daný vrchol *k* zahrneme do množiny hran *Γ* rovněž hrany spojující předchůdce a následovníky tohoto vrcholu. Lze ukázat, že přechodný uzávěr grafu je možno určit postupnou dekompozicí operace Θ v případě grafu o vrcholech 1, …, n. $\Gamma^+ = \Theta_n(\Theta_{n-1...}(\Theta_1)...)$

Díky své značné jednoduchosti a vysokému výkonu je Roy-Warshallův algoritmus právem považován za jednu z programátorských perel. Dočkal se samozřejmě také svých modifikací. Pokud například zavedeme slovník navrhovaných cest, můžeme algoritmus poupravit tak, aby našel správnou cestu grafem. Jednou z dalších významných modifikací je Floyd-Warshallův algoritmus, který slouží k nalezení nejkratší cesty grafem.

5 Implementace domácích úloh

V této kapitole popíši, jak jsem postupoval při samotné implementaci domácích úloh. Uvedu, s jakými problémy jsem se při implementaci setkal, jaká dilemata jsem musel řešit, a objasním řešení, která jsem při implementaci zvolil. Pokud to bude nutné, uvedu také příklady částí zdrojových kódů, ale nebudu nějak hlouběji zabíhat do samotné implementace jednotlivých funkcí, protože jsem jednak abstraktní datové typy podrobněji popsal v kapitole 3 a také jsem velmi podrobně okomentoval zdrojové kódy, které naleznete v příloze této bakalářské práce na přiloženém CD.

V zásadě se lze na problematiku domácích úloh pro předmět algoritmy dívat hned z několika pohledů:

- Pedagogický přístup.
- Zažité konvence (struktura kódu, počet souborů, komentáře).
- Technická stránka věci (nutné postupy pro správnou funkci).

V následujících odstavcích proberu postupně tyto aspekty.

5.1 Pedagogické hledisko

V 6. semestru jsem absolvoval předmět pedagogická psychologie a tedy si dokáži docela dobře uvědomit, jaká úskalí a dilemata čekají na pedagoga při přípravě práce pro studenty. Ovšem teprve nyní jsem okusil tento úděl jak se říká „na čisto“. Tvůrce domácích úloh musí mít na vědomí hned několik pohledů na zadání domácí úlohy. Může se jednat o rozsah, složitost, ale také přitažlivost pro studenta, dostupnost materiálů a v neposlední řadě také význam úlohy a její přínos pro studenta.

5.1.1 Rozsah

Pedagog navíc bývá, nebo alespoň by měl být, přesvědčen, že právě jeho předmět student potřebuje přímo esenciálně pro svůj další rozvoj. Přesto musí brát v úvahu, že student navštěvuje také jiné předměty, a vyučující těchto lekcí mají také potřebu prosazovat to své. Pedagog proto musí rozumně zvážit, zda nezatěžuje studenta příliš. Efektivita a kvalita výuky sice s narůstajícími požadavky průměrně stoupá, jenomže existuje jistá hranice, přes kterou by se nemělo přejít. Potom totiž student musí na úkor jedné činnosti zanedbávat ostatní, neméně důležité.

Předmět algoritmy hodnotím v tomto ohledu jako přiměřený. Mé hodnocení považuji za docela akceptovatelné, protože jsem byl před rokem sám na místě studentů tohoto kurzu. Student dostává zadané dvě sady úloh. Každá sada obsahuje tři samostatné úlohy, které jsou vhodně seskupeny v rámci tématiky, ale též obtížnosti. Letos se ovšem bude nasazovat nový postup, který by měl studentovi přiblížit vzájemnou provázanost jednotlivých datových typů a způsobit u něj jistý „myšlenkový posun“. Jedná se o využití implementace jedné domácí úlohy v úloze jiné. Student musí vstřebat zkombinování více domácích úloh, ale také současně pochopit jejich význam, a ne je jen slepě realizovat podle zadání.

Samotný rozsah jednotlivých úloh se diametrálně liší podle tématu. Je asi zřejmé, že pokud srovnám množství operací například u dvousměrně vázaného lineárního seznamu a u zásobníku, rozdíl je značný.

5.1.2 Složitost

Zde platí to známé „přiměřeně“, tedy přiměřeně probírané látce, znalostem a schopnostem studenta. Cílem úloh je především prakticky procvičit znalosti, které student získává během přednášek. I když se u mnoha z úloh přímo nabízí, aby zadání obsahovalo o něco více. Jako typický příklad může sloužit úloha *c601*, kterou naleznete v příloze na CD v adresáři *domaci_ulohy*. Abstraktní datový typ graf, využívaný zejména v umělé inteligenci, by vybízel k implementaci nejednoho z mnoha zajímavých algoritmických problémů, ale student by byl zbytečně zaměstnán vymýšlením složitého algoritmu, než aby se soustředil na vyučovanou datovou strukturu. Mimo jiné, také z tohoto důvodu jsou úlohy koncipovány na podrobné procvičení operací datového typu, a ne na jeho zasazení do většího projektu. O to se následně postarají jiné předměty vyučované na fakultě.

5.1.3 Přitažlivost pro studenta

Dalším velice důležitým aspektem je zájem studenta o probíranou látku. Studenti si často neuvědomují důležitost učiva a projektů a snaží se příklad hlavně rychle vypracovat, než pochopit. Člověk, který vymýšlí domácí úlohu, by s tímto měl jistě počítat a zvýšit atraktivitu a pokud možno i zábavnost projektu. To ovšem u domácích úloh v předmětu algoritmy představuje zásadní potíž. Úlohy jsou koncipovány spíše technicky a mnoho příležitostí k zábavě při vypracovávání nenabízí.

Pokud je tak obtížné atraktivitu zvýšit, bylo by dobré pokusit se ji alespoň nesnižovat. Oblíbenost takové úlohy zhoršují zejména nepřesnosti a chyby v zadání, složité a nelogické pojmenování používaných proměnných a pocit studenta, že „to vlastně nikdy nebude potřebovat“. Proto je nutné dbát na výše jmenované fenomény a co nejvíce se jich vyvarovat.

5.1.4 Dostupnost materiálů

Materiály k domácím úlohám pro předmět algoritmy lze obecně získávat ze třech zdrojů. Jsou to podklady z přednášek a studijních opor, komentáře ve zdrojových kódech a externí zdroje. Zatímco na první dva lze bezezbytku spoléhat, třetí je nejistý, a tedy nelze počítat s jeho existenci, natož kvalitou. Přesto se domnívám, že by student Vysokého učení technického měl disponovat schopností vyhledávat si samostatně informace. Tato schopnost pro něj bude později představovat hlavní informační kanál. Právě proto jsem zadal v úloze práce s orientovaným grafem k vypracování operace součtu, kompozice a Roy-Warshallova algoritmu, což by dle mého názoru nemělo způsobovat studentovi, který dostatečně pochopí problematiku, sebemenší potíže.

5.1.5 Přínos pro studenta

O přínosu domácích úkolů z předmětu algoritmy jsem vedl pojednání již v úvodu této bakalářské práce. Nyní se k tomuto tématu ještě jednou vrátím.

Student vypracovává domácí úlohy hned z několika důvodů:

- Získání bodového ohodnocení.
- Získání znalostí ke zkoušce semestrální i státní.
- Rozšíření znalostí pro použití v jiných předmětech.
- Porozumění principům tvorby algoritmů.
- Získání výchozích znalostí, dovedností a návyků pro budoucí práci a rozvoj.

Především poslední dva body by měl student přijmout jako nezvratnou skutečnost, která se bude následně odrážet v jeho profesionální kariéře. V informatice prakticky neexistuje obor, v němž by se neskloňoval pojem algoritmus. Umění jeho tvorby potom jde ruku v ruce s úspěšností

programátora. Návyky, které začínající programátor pochytí, ho doprovázejí dlouhou dobu, někdy se dokonce vůbec nezmění. Proto je nutné tyto návyky pochytit správně a trvale.

5.2 Zažité konvence

Každý jednotný soubor programů by měl mít také jednotný formát. A právě o formátu domácích úloh do předmětu algoritmy bude několik následujících odstavců. Upozorňuji, že se nejedná o záležitosti funkčnosti úloh, ale pouze jejich rozvržení. Význam této kapitoly je dle mého názoru značný, protože nejednotná koncepce i funkčních programů způsobuje nepřehlednost a v důsledku zvýšenou neefektivitu, což také odrazuje od práce jak studenty, tak tvůrce zadání. V této kapitole zkoumám hlavně konvence:

- Názvů proměnných a funkcí,
- rozvržení funkcí a jejich volání,
- komentářů a jejich jazyka,
- názvů souborů,
- výpisů na standardní výstup.

5.2.1 Názvy proměnných a funkcí

Pro proměnné abstraktních datových typů a funkce byly zavedeny přesné konvence. Ty by měly napomoct lepší orientaci ve zdrojovém kódu jak jeho tvůrci, tak hlavně studentům. Pod výpisem pravidel naleznete ukázkový příklad.

- Prefix obsahuje písmeno malé t následované anglickou zkratkou velkými písmeny.
- Zkratka představuje přesný název datového typu.
- Základ názvu datového typu začíná vždy velkým písmenem a vystihuje co nejlépe její podstatu.
- Základ názvu funkce začíná vždy velkým písmenem a vystihuje co nejlépe její podstatu.
- Každé další slovo v základu názvu následuje bezprostředně bez mezery a začíná rovněž velkým písmenem.
- Pokud se jedná o datový typ ukazatel na abstraktní datový typ, pak název odpovídá názvu ADT, za nímž následuje bezprostředně bez mezery přípona Ptr. Ta začíná rovněž velkým písmenem.

```
typedef struct tDLCElem {/* položka seznamu */
  int data;              /* užitečná data položky */
  struct tDLCElem * lptr;/* ukazatel na předchozí položku */
  struct tDLCElem * rptr;/* ukazatel na následující položku */
} * tDLCElemPtr;

typedef struct tDLCList{/* dvojitě vázaný kruhový lineární
seznam */
  tDLCElemPtr First; /* ukazatel na první položku seznamu */
  tDLCElemPtr Active;/* ukazatel na aktivní položku seznamu */
} * tDLCListPtr;
```

```
/* prototypy funkcí */
void Error();
void DLCInitList (tDLCListPtr L);
void DLCDisposeList(tDLCListPtr L);
void DLCInsertFirst (tDLCListPtr L, int d);
void DLCInsertLast (tDLCListPtr L, int d);
void DLCFirst (tDLCListPtr L);
void DLCLast (tDLCListPtr L);
void DLCCopyFirst (tDLCListPtr L, int * val);
void DLCCopyLast (tDLCListPtr L, int * val);
void DLCDeleteFirst(tDLCListPtr L);
void DLCDeleteLast(tDLCListPtr L);
void DLCPostDelete(tDLCListPtr L);
void DLCPreDelete(tDLCListPtr L);
void DLCPostInsert(tDLCListPtr L, int d);
void DLCPreInsert(tDLCListPtr L, int d);
int  DLCCopy(tDLCListPtr L);
void DLCActualize (tDLCListPtr L, int d);
void DLCSucc(tDLCListPtr L);
void DLCPred(tDLCListPtr L);
int DLCActive (tDLCListPtr L);
```

Obr. 5.1 – Ukázka použitého názvosloví v domácích úlohách

5.2.2 Rozvržení funkcí a jejich volání

- Funkce, které má vypracovat student, se nachází v souboru cXXX.c.
- Pomocné funkce pro studenta se nachází v souboru cXXX.c.
- Definice datových typů se nachází v hlavičkovém souboru cXXX.h.
- Testovací funkce včetně funkce pro tisk, funkce main a dalších pomocných testovacích funkcí se nachází v souborech testů cXXX-test.c a cXXX-advanced-test.c. Tyto funkce nemá student k dispozici a nesmí s nimi manipulovat.
- Studentovy funkce nejsou v testovacích zdrojových kódech volány přímo, ale prostřednictvím funkcí, jež mají stejný název, jen obsahují navíc prefix use nebo use_. Tyto funkce totiž zajišťují ošetření vyvolaných chybových stavů a nebo případné nenaimplementování řešení.

5.2.3 Komentáře

- Komentáře se nově zpracovávají dvojjazyčně, v češtině a angličtině. Obojí komentáře se musí nacházet v kódu pro studenta, jinde stačí anglické.
- Zdrojové texty se komentují dostatečně a přehledně.
- Každý zdrojový text musí mít hlavičku se základními informacemi (soubor, název, autor).
- Zdrojový text, kam má student psát svá řešení, obsahuje celkový přehled a základní rady a reference k vypracování úlohy.
- Každá funkce, jež má být implementovaná studentem, obsahuje ihned za svou hlavičkou podrobný popis její funkčnosti.

- V hlavičkovém souboru se nachází podrobný popis datových typů použitých v domácí úloze.

5.2.4 Názvy souborů

- Úlohy spadají do kategorií, které jsou označeny trojciferným číslem.
- Každý název souboru začíná prefixem vyjadřujícím jazyk implementace (např. c pro jazyk C) následovaný číslem úlohy.
- Názvy testovacích souborů mají příponu –test či –advanced-test.
- Každý soubor končí příponou podle jeho typu.

5.2.5 Výpisy na standardní výstup

- Veškerý výstup je realizován v anglickém jazyce.
- Chyby jsou uvozeny řetězcem „ERROR: “.
- Hlavičky jednotlivých bodů testovacích úloh jsou uvozeny řetězcem „[TESTčíslotestu]“ následovaným stručným popisem daného bodu.
- Ostatní výpisy by měly být zobrazovány jednoduše a co nejpřehledněji.

5.3 Technická stránka

Tuto podkapitolu rozčlením na dvě části. První část bude pojednávat o obecných konvencích, které jsou nutné pro správné začlenění nových domácích úloh do existujícího systému. V druhé části se zaměřím na úlohy *c208* a *c601*. Tyto dvě úlohy je přímo nutné vysvětlit a popsat, protože se jedná o zcela nový typ úloh. Úloha *c208*, tedy zásobník implementovaný dvojitě vázaným lineárním seznamem, představuje nový trend mezi domácími úlohami pro předmět algoritmy, úloha *c601*, práce s orientovaným grafem, pak otevírá novou větev začínající číslem 6, a většinu jejich funkcí jsem naimplementoval podle vlastního uvážení, a ne podle nějakého vžitého schématu. Právě proto se sluší, abych podrobně objasnil důvody mého řešení.

5.3.1 Obecné konvence

5.3.1.1 Pojmenování souborů a jejich obsah

Každá domácí úloha je složena z pěti souborů. XXX označuje číslo úlohy:

- cXXX.c – Soubor se zadáním. Student modifikuje pouze těla funkcí v tomto souboru.
- cXXX.h – Hlavičkový soubor. Obsahuje definici datových typů a jejich popis.
- cXXX-test.c – Soubor základních testů. Obsahuje funkci main. Stará se o spouštění testovacích úloh, které má student k dispozici společně se zadáním.
- cXXX-advanced-test.c – Soubor pokročilých testů. Obsahuje funkci main. Stará se o spouštění testovacích úloh, které nemá student k dispozici společně se zadáním.
- Makefile – soubor pro program GNU Make

5.3.1.2 Speciální značky

Systém zahrnuje program interpretu. Ten slouží na základě volby uživatele k vygenerování potřebných souborů domácí úlohy. Aby interpret rozpoznal, která část kódu se vztahuje k čemu, jsou smluveny speciální značky, které musí autor domácích úloh vhodně zakomponovat do zdrojového textu.

- /**/ - Text ohraničený těmito značkami je vygenerován beze změn.
- /*v*/ - Text ohraničený těmito značkami je považován za vzorové řešení, a tedy samozřejmě není poskytnut studentům v zadání.
- solved=0; - Nachází se v řešených funkcích. Značí, že funkce ještě nebyla studentem řešena. Pokud student funkci vyřeší, řádek zakomentuje či smaže.

5.3.2 Nové typy domácích úloh

5.3.2.1 c208 – Zásobník

Jak již bylo v tomto textu několikrát zmíněno, jedná se o implementaci abstraktního datového typu zásobník pomocí dvojitě vázaného lineárního seznamu. Zvláštnost této úlohy spočívá ve faktu, že se poprvé využilo implementace jiné domácí úlohy napsané v jazyce C (v době, kdy byly domácí úlohy napsány v jazyce Pascal, existovaly podobná zadání). Student tedy nemá za úkol vymýšlet složité algoritmické konstrukce pro práci s dvojitě vázaným lineárním seznamem, ale jen využije již implementované operace. Význam této úlohy je značný, protože nutí studenta k jistému „mentálnímu posunu“ a hlubšímu pochopení využití abstraktního datového typu.

Při implementaci jsem se snažil realizovat zásobník na co možná nejvyšším stupni abstrakce, tak, aby se jeho implementace co nejvíce přibližovala charakteristice abstraktního datového typu, jak jsem ji uvedl v kapitole 4.1.

Datová struktura pak derivuje datovou strukturu seznamu, datový typ prvku je také abstrahován a operace pracují s novými datovými typy. Pro studenta je implementace operací nad lineárním seznamem zapouzdřena a neměl by na ni brát jakékoli ohledy.

```
typedef tDLList tStack;     /* datatype fo the stack */
typedef int tData;          /* datatype of value of stack */

/* Headers of functions for work over stack. */
void stackError ( tData );
void stackInit ( tStack * );
int stackEmpty ( tStack *  );
tData stackTop ( tStack *  );
void stackPop ( tStack *  );
void stackPush ( tStack * , tData );
tData stackTopPop ( tStack * );
```

Obr. 5.2 – ADT zásobník implementovaný dvojitě vázaným lineárním seznamem

5.3.2.2 c601 – Orientovaný graf

Úloha práce s orientovaným grafem je zcela novou úlohou. Problematika grafů se zatím vyskytovala mezi domácími úlohami pro předmět Algoritmy jen v jejich velmi specializované podobě jako práce s binárními stromy. Proto zde chci nyní objasnit mé pohnutky, které mě vedly k realizaci v té podobě, jakou naleznete na přiloženém CD v adresáři *domaci_ukoly/c601*.

Zvolená podoba abstraktního datového typu je popsána již v podkapitole 4.2.3 tohoto technického textu a také podrobně okomentována v příslušných zdrojových textech, které se nachází ve zmíněném adresáři na CD.

Teď tedy přistoupím k vysvětlení jednotlivých vybraných základních operací nad tímto datovým typem, u kterých by mohly vzniknout jisté nejasnosti v implementaci.

*tDGVertexPtr DGVertexExist (tDGGraphPtr G, char * name)*

Funkce zjišťuje, jestli existuje uzel, jehož jméno je předáno v parametrech. Pokud existuje, funkce vrátí jeho adresu, jinak vrátí NULL. Tuto funkci jsem implementoval jako předpřipravenou pro studenty. Učinil jsem tak z toho důvodu, že její implementace je poměrně jasná (jedná se o prohledání lineárního seznamu) a použití časté. Podle mého názoru nemá smysl zatěžovat studenta těmito jednoduchými operacemi, když má za úkol procvičit si práci s tak složitou datovou strukturou, jakou je orientovaný graf.

*tDGEdgePtr DGEdgeExist (tDGVertexPtr First, char * name)*

Funkce zjišťuje, jestli existuje hrana, která začíná v uzlu First a směřuje do uzlu, jehož jméno je předáno v parametrech v proměnné name. Pokud hrana existuje, funkce vrátí její adresu, jinak vrátí NULL. Funkce je opět naimplementována dopředu ze stejných důvodů, jako předešlá.

Sporné mohou ovšem připadat parametry funkce. Ty jsou ale takto zvoleny z důvodu efektivity. Pokud by se totiž měla vyhledávat hrana touto funkcí, jejíž parametry by byly oba ukazatele na uzel grafu, musely by se projít 3 lineární seznamy v případě, že známe jen název uzlu, což je nejčastější případ. Dva by se musely procházet ještě před použitím této funkce pro nalezení adresy uzlů a třetí při procházení seznamu uzlů. V mé verzi této funkce se prochází pouze dva seznamy. První pro nalezení adresy počátečního uzlu a druhý při hledání existence uzlu koncového v seznamu hran daného počátečního uzlu.

*void DGVertexDelete (tDGGraphPtr G, char * name)*

Funkce odstraní uzel, jehož jméno je uvedeno v parametru name. Pokud uzel daného jména v grafu G neexistuje, je volána chybová funkce *DGError(DELERR)*. Realizace funkce se očekává od studenta. Opět je zde parametr sporným bodem. Zde mě zvláště netěší použití sémantického označení mazaného uzlu, ovšem z hlediska následného testování studentovi implementace se jedná o jediné rozumné řešení. Jinak bych totiž musel pokaždé mazaný uzel nejprve vyhledávat, což by učinilo testování značně krkolomným a nepřehledným.

*void DGEdgeInsert (tDGGraphPtr G, char * fromName, char * toName)*

Funkce vytvoří novou hranu která bude vycházet z uzlu pojmenovaného fromName a bude směřovat do uzlu pojmenovaného toName. Pokud tato hrana již existuje, či neexistuje některý z uzlů hrany, volá se funkce *DGError(LINKERR)*. Opět může nastat debata ohledně typu parametrů a mé vysvětlení je stejné, jako u předchozí funkce. Ten samý problém rovněž se stejným odůvodněním vzniká též u funkce *DGEdgeDelete (tDGGraphPtr G, char * fromName, char * toName)*, která má naopak za úkol vybranou hranu odstranit.

Řazení

Poslední problém nastává při modifikaci grafu. V definici grafu totiž nikde není řečeno, že by měly být jednotlivé uzly jakkoli seřazeny, a už vůbec ne podle svého názvu. Proto nemůžu studenty nutit k implementaci, jež by zahrnovala také určité řazení uzlů. Nicméně pro testovací účely se jeví naprosto nevhodné, aby byly uzly seřazeny jakýmkoli libovolným způsobem. Usoudil jsem, že toto není problém studenta, ale tvůrce úlohy. Z toho důvodu jsem v obou testovacích souborech (c601-

test.c a c601-advanced-test.c) využil algoritmů rychlého řazení quicksort. Funkce, které tyto algoritmy realizují se jmenují *nodeSort* pro řazení uzlů a *linkSort* pro řazení hran. Jejich vliv na domácí úlohu lze velice lehce odstranit jejich zakomentováním ve funkci pro tisk obsahu struktury grafu *printGraph*.

6 Systém pro automatické vyhodnocování domácích úloh

6.1 Stručný popis

Systém pro automatické vyhodnocování domácích úloh se sestává z několika skriptů napsaných v jazycích PHP, Bash a Perl a interpretru implementovaném prostřednictvím jazyka C. Veškeré části jsou odladěny pro provoz na operačním systému FreeBSD na školním serveru eva. Pro vypracování posledního úkolu zadání mé bakalářské práce bylo nezbytné, abych nastudoval funkci tohoto systému, přesněji zejména skriptů generujících výstupní soubory. V následujících odstavcích této podkapitoly zběžně popíši strukturu systému včetně funkcí jednotlivých částí. Zdrojové soubory systému naleznete v příloze na CD v adresáři *system*.

6.1.1 Interpreter

Na základě volby provede vytvoření adresářů *vzorove* a *zadani*, kam následně generuje vybrané domácí úlohy. Úlohy jsou vygenerovány v závislosti použití speciálních značek v jejich zdrojových textech. Přehled značek naleznete v této práci v podkapitole 5.3.1.2 Speciální značky.

6.1.2 Porovnání výstupů – skript checker.php

Skript slouží ke srovnání dvou souborů, které jsou mu předány v parametrech. První soubor obsahuje vzorový výstup domácí úlohy, druhý pak výstup úlohy studenta. Výstupem skriptu je výpis úspěšnosti jednotlivých testovacích bodů.

```
[TEST01] OK
[TEST02] OK
[TEST03] OK
[TEST04] failed
[TEST05] OK
[TEST06] OK
[TEST07] OK
```

Obr 6.1 – Příklad výstupu skriptu checker.php

6.1.3 Spouštění jednotlivých úloh – skript saferun.pl

Skript je určen pro bezpečné spouštění studentských řešení. Spuštění testu studenta se díky tomuto skriptu realizuje jako samostatný proces, aby v případě závažné chyby, kterou například představuje zacyklení programu, nedošlo k přerušení běhu celého systému.

Skript vždy v intervalu 0.1s kontroluje stav procesu. Pokud délka jeho života přesáhne nastavenou časovou hranici, proces je ukončen, a na výstup je tisknuta zpráva o chybném průběhu. V případě úspěchu se skript chová „tiše“.

6.1.4 Hlavní kontrolní skript – checkall.sh

Jedná se o „srdce“ celého systému pro kontrolu domácích úloh. Skript plní řadu funkcí, na které se nyní podrobněji zaměříme.

6.1.4.1 Zjištění příkladů a testů ke kontrole

Na začátku skriptu se nachází 2 řetězce. První z nich obsahuje názvy domácích úloh, které budou kontrolovány, druhý pak testy spouštěné nad domácími úkoly. Jednotlivé položky jsou vždy odděleny mezerami.

6.1.4.2 Kompilace vzorových řešení a vygenerování vzorových výstupů

V cyklu se pro každý projekt provede sestavení vzorových řešení pomocí příkazu *make*. Zkompilovaný soubor se spustí a vygeneruje vzorový výstup, jež je následně srovnáván se studentovým řešením pomocí skriptu checker.sh.

6.1.4.3 Kompilace a sestavení odevzdaných řešení

Skript nyní prochází jednotlivé složky studentů označené jejich loginem a obsahující odevzdaný soubor s řešením. K tomuto souboru je prostřednictvím skriptu dokopírován soubor testů, Makefile a hlavičkový soubor. Následně se přistoupí k sestavení úlohy pomocí příkazu make. Veškeré chybové stavy se zaznamenávají do logovacího souboru v témže adresáři. Skript bude dále pokračovat, jen pokud sestavení studentova řešení proběhlo úspěšně, jinak přidělí studentovi 0b za všechny testy.

6.1.4.4 Spuštění testů

Skript přistoupí k samotnému spouštění jednotlivých testů pomocí výše zmíněného skriptu *saferun.pl* (kapitola 6.1.3). Výstup skriptu je přesměrován do souboru s příponou *.output*.

6.1.4.5 Vyhodnocení výsledků

Vyhodnocení probíhá pomocí spuštění výše zmíněného skriptu *checker.php* (kapitola 6.1.2). Výsledek se uchová opět ve studentově adresáři. Při kontrole je možné definovat více vzorových řešení v případě nejednoznačnosti.

6.1.4.6 Výpis výsledků na výstup

Výsledky jsou tisknuty na standardní výstup ve formátu, který vyhovuje použití programu *grep* pro snadné zpracování. Výstup obsahuje informace jako login studenta, název příkladu a shrnutí výsledků s přiděleným počtem bodů.

6.1.5 Přidělování bodů – skript points.sh

Skript má za úkol podle výpisu výsledků přidělit studentovi v závislosti na stupnici hodnocení počet bodů. Tuto stupnici je nutné přímo ve skriptu definovat. Skript je oddělen od hlavního skriptu, protože cvičící se může rozhodnout změnit bodová ohodnocení jednotlivých testů až po provedení hlavního skriptu. Při oddělení obou částí tak není nutné provádět vše znovu. Výstupem skriptu je tedy stručný souhrn dosažených výsledků a bodů pro každého studenta.

6.1.6 Odesílání e-mailů – skripty preparemails.sh a sendmails.sh

Skript *preparemails.sh* má za úkol připravit podobu e-mailů rozesílaných jednotlivým studentům. Skript *sendmails.sh* následně vytvořené e-maily rozesílá. Skripty jsou oddělené, aby mohl cvičící zasáhnout např. do množiny příjemců nebo jinak zprávy modifikovat.

6.1.7 Odhalování plagiátů – skript findduplicates.php

Skript se stará o odhalování plagiátů řešení domácích úloh. Studenti totiž mají často tendenci domácí úkol nezpracovávat samostatně a uchylují se k jednodušší variantě, tedy zkopírování jeho podstatných částí od jiných řešitelů. Musí se ovšem počítat s faktem, že studenti si jsou vědomi rizik, která představuje jejich odhalení. Proto také systém pro kontrolu domácích úloh kalkuluje s rozličnými způsoby, jaké studenti mohou při plagiátorství využít. Jedná se zejména o záměnu komentářů, názvů proměnných, prohazování řádků zdrojového textu či jiného formátování. Systém na výše zmíněná rizika pamatuje právě prostřednictvím skriptu *findduplicates.php*.

Plagiát se může odhalit také pomocí srovnání výstupů domácích úloh. Zde platí staré známé přísloví: „Stejné chyby, stejné známky."

Postup při odhalování plagiátů:

- Odstranění komentářů pomocí deterministického automatu.
- Změna všech konstantních řetězců na prázdné.
- Odebrání netisknutelných znaků.
- Seřazení řádků zdrojového textu podle abecedy.
- Provedení kontrolního součtu pomocí MD5 algoritmu.

Algoritmus MD5 pro kontrolní součty je využit z důvodu obrovské režie v případě porovnávání celých zdrojových textů všech studentů. Výpočetní nároky v situaci stále se zvyšujícího počtu studentů by byly zcela neúnosné. Řešení pomocí kontrolního součtu je jednoduché a brilantní.

6.2 Modifikace systému

6.2.1 Zadání problému a návrhy jeho řešení

Mým posledním úkolem ze zadání bakalářské práce bylo vyřešit problém vzniku velkých výstupních souborů (až v jednotkách gigabytů) při testování studentských řešení. Nabízelo se hned několik možností, jak situaci ošetřit.

Velké soubory na výstupu vznikají kvůli chybám studentů při práci s ukazateli a následnému zacyklení testovacího programu. Proto zřejmě nejefektivnější a reakčně nejrychlejší řešení spočívá ve vložení jakési pojistky, tedy horní hranice vypisovaných prvků, přímo do každého testovacího zdrojového souboru. Výhody tohoto řešení:

- Chyba je eliminována již v samotném programu domácí úlohy.
- Ke vzniku velkého výstupu vůbec nedojde.
- Student má již doma při vypracovávání možnost detekovat takovou chybu a vyvarovat se jí.

Ovšem tento návrh má také své nedostatky:

- Ošetření není univerzální, musí se zahrnout do každé úlohy.

- Horní hranice počtu vypisovaných prvků se musí při každé modifikaci testů kontrolovat.
- K chybě nemusí docházet jen ve funkci pro tisk obsahu datových struktur a pojistka je v takovém případě neúčinná.

S přihlédnutím k výše uvedeným nedostatkům mi nezbylo nic jiného, než hledat řešení problému o úroveň výše, tedy přímo ve skriptech samotného systému pro automatické vyhodnocování domácích úloh.

Představa byla poměrně jasná. Velké potíže při opravování totiž nepředstavuje vznik jednoho takového objemného souboru, ale možnost vzniku více takových při hromadném zpracování studentských řešení. Pro představu uvedu, že pokud v ročníku studuje 500 lidí a například jen u 40% z nich by došlo k výše uvedené komplikaci, mohlo by se rázem volné místo na disku počítače, kde se úlohy opravují, zmenšit až o 100GB. Proto stačí, když okamžitě po vzniku takového souboru zareaguje mechanismus, který velikost vzniklého výstupu „ořízne“. Jeho úplné smazání by přitom nepředstavovalo správnou cestu, protože by tak student přišel i o body, které eventuálně mohl dostat za nezacyklenou část úlohy. V kombinaci s předchozí uvedenou pojistkou přímo ve zdrojových textech domácích úloh by pak mohla být velká většina těchto situací zcela eliminována.

Celý postup by pak byl následující:

- Detekce velkého souboru. Zvolil jsem hranici 0.5MB.
- Ořezání do nového souboru s příponou .tmp na přijatelnou velikost. Zvolil jsem řádky maximální délky 200 znaků a jejich počet omezil na maximálně 2000. Mé parametry jsou na první pohled předimenzované, ovšem představují určitou rezervu, kterou je možné akceptovat jako vyhovující.
- Přepsání původního souboru obsahem dočasného.
- Odstranění dočasného souboru.

6.2.2 Použití změn

Jakmile jsem měl vypracovanou koncepci, zbývalo jen vybrat vhodný soubor, do něhož bych celý mechanismus začlenil. Podle poznatků získaných při studiu systému jsem nakonec zvolil skript *checkall.sh* implementovaný v programovacím jazyka Bash. Konstrukci jsem vložil do místa, které je určeno ke zpracování výsledků testu jediného studenta, jak ukazuje následující obrázek.

```
for TEST in $TESTS; do
          echo -e -n "\t[$PROJECT]Running '$TEST' ... "
          #compilation successfull lets check results
          ./saferun.pl 15 $DIR/${PROJECT}-$TEST > $DIR/${PROJECT}-
          $TEST.output

          #we store size of the output file to varialble SIZE
          SIZE=$(ls  -l  $DIR/${PROJECT}-$TEST.output  |  sed  's/\(
          +\)/ /g' | sed 's/  / /g' | cut -d ' ' -f 5)

          #we find out if output is not bigger then it should be
          #if it is bigger, we cut the file on 2000 lines with 200
          #chars or smaller
```

```
#if number of lines or number of chars in a line is
#smaller it won't be increase
if [ $SIZE –gt 500000 ]; then
     cat $DIR/${PROJECT}-$TEST.output | cut –c 1-
     200 | head -2000 > $DIR/${PROJECT}-$TEST.output.tmp;
     cat $DIR/${PROJECT}-$TEST.output.tmp >
     $DIR/${PROJECT}-$TEST.output;
     rm $DIR/${PROJECT}-$TEST.output.tmp;
fi

echo "OK"
echo -e -n "\t[$PROJECT]Comparing results ... "
#pro pripad kdy bylo nejasne zadani a exituje vice
#moznych reseni prozkoumame vsechny a zapocitame nejlepsi
#vysledek
#budeme porovnavat se vsemi prislusnymi *.output* soubory
NUMOK=0;
NUMFAILED=0;
```

Obr. 6.2 – Výpis řádků 62 až 84 ze skriptu checkall.sh

Pokud tedy bude zachycen soubor větší jak 0.5MB, jeho velikost se zkrátí tak, aby obsahoval maximálně 2000 řádků po maximálně 200 znacích.

7 Závěr

Výsledkem mého snažení jsou tři nové domácí úlohy. Jedná se o práci s dvousměrně vázaným kruhovým lineárním seznamem, zásobníkem implementovaným pomocí dvousměrně vázaného lineárního seznamu a zejména o práci s orientovaným grafem. U první z nich se jednalo spíše o rozšíření sbírky úloh zaměřených na lineární seznamy, ovšem zbylé dvě úlohy přinesly nové pojetí zadávání domácích úloh a nový abstraktní datový typ, díky němuž vznikla větev zadání s novým číslováním.

Od úlohy práce se zásobníkem si slibuji zejména zlepšení povědomí studentů o praktickém využití datových struktur a abstraktním datovém typu obecně. Od úlohy práce s orientovaným grafem zase očekávám lepší proniknutí studenta do problematiky umělé inteligence vyučované v následujícím semestru. Při tvorbě úloh byly použity některé nové prvky, jako kompletní komentáře v anglickém jazyce, provázání několika zadání do jednoho celku nebo kontrola zacyklení při výpisu obsahu datových struktur. V úloze práce s orientovaným grafem byla navíc vytvořena nová implementace abstraktního datového typu graf včetně názvosloví jednotlivých datových typů, jejíž koncepce se doufám v budoucnu bude dodržovat v celé sbírce úloh s kódem 600.

V druhé části práce jsem se zabýval problematikou systému pro automatické vyhodnocování domácích úloh. Mým úkolem bylo zvýšení robustnosti systému. Toho jsem dosáhl úpravami systémových skriptů v kombinaci s využitím pojistného mechanismu přímo v testech pro domácí úlohy. Při řešení problému jsem pamatoval také na zájmy studenta. Studentovo závadné řešení tedy není zcela odstraňováno, ale pouze modifikováno tak, aby bylo únosné pro systém a současně student neztrácel zasloužené bodové ohodnocení.

Pro další úspěšný vývoj systému na zadávání a kontrolu domácích úloh navrhuji zavedení metodiky psaní zdrojových textů pro tvůrce domácích úloh. Jasnou metodikou by se dalo předejít nejasnostem a nepřehlednostem v zadáních a minimalizovalo by se tak riziko refaktorizace zdrojových textů, což by vedlo také ke snížení rizika vzniku chyb. Z tohoto důvodu jsem se pokusil v této technické zprávě shrnout skupinu, dle mého názoru, nejdůležitějších pravidel, která by měla být při tvorbě dodržována.

Dále navrhuji postupné úpravy systému tak, aby byl bezezbytku využitelný také v jiných předmětech na Vysokém učení technickém, které mají úlohy podobného charakteru. Rozsah a koncepce systému si zaslouží širší využití.

Literatura

[1] Honzík, J. M., Hruška, T., Máčel, M.: Vybrané kapitoly z programovacích technik, Vysoké učení technické v Brně, Brno, 1991. ISBN 80-214-0345-4.

[2] Herout, P.: Učebnice jazyka C, 3. upravené vydání, Kopp, České Budějovice, 1998. ISBN 80-85828-21-9.

[3] Wróblewski, P.: Algoritmy, 3. upravené vydání, Computer press, Brno, 2004. ISBN 80-251-0343-9.

[4] Vychodil, V.: Linux, 1. vydání, Computer press, Brno, 2003. ISBN 80-7226-333-1.

[5] Kadlec, V.: Agilní programování, 1. vydání, Computer press, Brno, 2004. ISBN 80-251-0342-0.

[6] Schildt, H.: Nauč se sám C, 3. upravené vydání, SoftPress s.r.o., Praha, 2001. ISBN 80-86497-16-X.

[7] Java.net, TIOBE: Java 6x populárnější než .NET., TPCI Long Term Trends, 9. března 2006. Dokument dostupný na URL: http://www.neo.cz/~tomas/java.net/2006/03/tiobe-java-6x-populrnj-ne-net.html, (18. dubna 2007).

[8] Wikipedia.cz, Dokument dostupný na URL: http://cs.wikipedia.org/ (18. dubna 2007).

[9] Cormen T.H., Leiserson Ch. E., Rivest R.L., Introduction to Algorithms, 5. vydání, MIT Press, 2001.

A. Obsah přiloženého CD

- soubor *readme* – textový soubor obsahující informace o struktuře CD
- soubor *xrezac09_dp.pdf* – technická zpráva ve formátu PDF
- adresář *domaci_ulohy* – adresář obsahující implementace tří domácích úloh podle zadání
- adresář *system* – adresář obsahující upravený systém pro automatické vyhodnocování domácích úloh