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 JAN TRÁVNIČEK
AUTHOR

BRNO 2008

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 IN THE COURSE ALGORITHMS

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE — JAN TRÁVNIČEK
AUTHOR

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

BRNO 2008

Abstrakt

Práce pojednává o tvorbě domácích úloh pro předmět Algoritmy. Nejvýznamnější část tvoří tři nově implementované domácí úkoly. Součástí práce je i modifikování části systému pro automatické zadávání a vyhodnocování domácích úloh.

Klíčová slova

Předmět Algoritmy, domácí úloha, algoritmus, abstraktní datový typ, seznam, zásobník, graf

Abstract

This bachelor's thesis describes creation of homework in the course Algorithms. Three newly implemented homeworks form the most important part. A modification of a system for automatic homework evaluation is a part of this thesis too.

Keywords

Subject Algorithms, homework, algorithm, abstract data type, list, stack, graph

Citace

Jan Trávniček: Příprava domácích úloh pro předmět algoritmy, bakalářská práce, Brno, FIT VUT v Brně, 2008

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.

……
Jan Trávniček
10. května 2008

Poděkování

Děkuji svému vedoucímu Ing. Bohuslavu Křenovi Ph.D. za pomoc s touto prací.

Obsah

1 Úvod

Hlavní náplní této bakalářské práce je vytvoření tří nových domácích úloh pro předmět Algoritmy. Tento předmět je vyučován na Fakultě informačních technologií Vysokého učení technického v Brně jako povinný. Každý student se proto během svého studia s tímto předmětem setká a měl by být schopen získané znalosti aktivně používat. Ke kontrole těchto znalostí slouží právě domácí úlohy. Velká část práce se tedy věnuje domácím úlohám, které jsem měl za úkol implementovat, a to nejen z pohledu implementačního, ale i z pohledu přínosu pro studenta a dalších hledisek.

Ke vzorovému vypracování domácích úkolů je ovšem potřeba znát význam několika pojmů, které se k předmětu Algoritmy vážou. Několik kapitol se proto věnuje pojmům jako algoritmus, datová struktura nebo abstraktní datový typ.

Vzhledem k počtu studentů se již dnes samozřejmě nemohou domácí úkoly opravovat po jednom, a proto byl vytvořen systém, který se o zadávání a opravování těchto úkolů stará. Mým úkolem bylo upravit jednu z jeho částí, program pro vytvoření zadání nebo vzorového řešení. Požadavkem bylo program doplnit o možnost výběru mezi českým nebo anglickým zadáním, případně vzorovým řešením. Řešením tohoto problému se zabývá příslušná kapitola.

V závěru jsou shrnuty výsledky mé práce a možná rozšíření do budoucna.

2 Algoritmy

Vytváření a používání algoritmů je nedílná součást práce každého programátora, stejně tak i předmětu Algoritmy. Proto se v této kapitole pojmem „algoritmus“ budu zabývat. Nastíním také některé důležité vlastnosti algoritmů a jakým způsobem se zapisují.

2.1 Pojem „algoritmus“

Dle mého názoru je nejlepší pro pochopení pojmu „algoritmus“ uvést některou z jeho definic. Jako příklad poslouží definice z [3].

„Algoritmus je konečná posloupnost/uspořádání postupů aplikovaných na konečný počet dat, jež dovoluje řešit přibližně stejné třídy problémů.“

Zmíněná definice vyjadřuje, co si máme pod pojmem „algoritmus“ představovat, ale pouze z obecnějšího hlediska. Této definici vyhoví např. i recept na vaření, příp. jiné obecnější postupy. Z pohledu programátora a informatiky je daleko důležitější počítačový algoritmus, který je chápan jako teoretický postup při řešení konkrétní programové situace. Na rozdíl od obecného algoritmu by měl mít i jisté rozšiřující vlastnosti. V této práci, pokud nebude uvedno jinak, se pojmem „algoritmus“ míní počítačový algoritmus.

2.2 Vlastnosti algoritmů

Každý algoritmus by měl splňovat následující čtyři základní vlastnosti. Informace jsem čerpal z [4,6,7].

2.2.1 Konečnost

Každý algoritmus musí skončit po libovolně velkém, ale konečném počtu kroků. Každý jeho krok musí být opět konečný. Pokud by neskončil, nedostali bychom výsledek a algoritmus by pozbýval smyslu.

Existují i postupy, které tuto podmínku nesplňují. Patří mezi ně např. procesy, které průběžně reagují na podněty z okolí.

To, že algoritmus skončí v konečném čase není v praxi postačující. Od algoritmu je také vyžadováno, aby skončil v „rozumném čase“, tzn. abychom mohli smysluplně využít jeho výsledek.

2.2.2 Obecnost

Algoritmus by neměl řešit pouze jeden konkrétní případ, např. součet čísel 2 a 3. Měl by obecně vyřešit co nejvíce obdobných případů, např. součet dvou libovolných celých čísel.

2.2.3 Jednoznačnost

Jednoznačnost znamená, že každý krok algoritmu musí být přesně a jasně definován, v každé situaci musí být zřejmé, co a jakým způsobem se má provádět. Jakýkoliv stav algoritmu je určen výsledkem stavu, který mu bezprostředně předchází. Protože běžný jazyk nám tuto jednoznačnost neposkytuje, byly pro zápis algoritmů navrženy programovací jazyky.

2.2.4 Efektivnost

Za efektivní algorimus je obecně považován takový, jehož každá jednotlivá operace je natolik jednoduchá, aby mohla být v principu provedena pouze s použitím tužky a papíru.

S efektivností algoritmů se často spojuje jeho náročnost, složitost. Rozlišujeme časovou a postorovou složitost.

2.2.4.1 Časová složitost

Časová složitost algoritmů udává, jakým způsobem se bude měnit chování programu (počet operací) v závislosti na změně (počtu) vstupních dat. Nejčastějším hodnotícím kritériem pro algoritmy je asymptotická časová složitost. Vyjadřuje se porovnáním algoritmu s jistou funkcí pro N (počet položek) blížícímu se nekonečnu. Porovnání má podobu tří různých složitostí:

- O – Omikron – Vyjadřuje horní hranici časového chování algoritmu. Zápis $f(n) = O(g(n))$, označuje, že funkce $f(n)$ roste maximálně tak rychle jako funkce $g(n)$. Funkce $g(n)$ je horní hranicí množiny takových funkcí určené zápisem $O(g(n))$.
- Ω – Omega – Vyjadřuje dolní hranici časového chování algoritmu. Zápis $f(n) = \Omega(g(n))$, označuje, že funkce $f(n)$ roste minimálně tak rychle jako funkce $g(n)$. Funkce $g(n)$ je dolní hranicí množiny takových funkcí určené zápisem $\Omega(g(n))$.
- Θ – Theta – Vyjadřuje třídu chování shodnou s danou funkcí. Zápis $f(n) = \Theta(g(n))$, označuje, že funkce $f(n)$ roste tak rychle jako funkce $g(n)$. Funkce $g(n)$ je horní a současně dolní hranicí množiny funkcí určených zápisem $\Theta(g(n))$.

Pro praktické aplikace se nejčastěji používá složitost Omikron, která vyjadřuje tzv. nejhorší případ.

Při určování složitosti zanedbáváme multiplikativní a aditivní konstanty. Lineární časová složitost pak např. udává, že doba trvání algoritmu se zvýší přibližně tolikrát, kolikrát se zvýší počet vstupních dat. U kvadratické složitosti se při dvojnásobném zvýšení počtu dat doba trvání zvýší čtyřikrát. V následující tabulce jsou uvedeny některé základní typy složitostí.

O(1)	konstantní
O(log N)	logaritmická
O(N)	lineární
O(N log N)	lineárnělogaritmická
$O(N^2)$	kvadratická
$O(N^3)$	kubická
$O(N^X)$	polynomiální
$O(X^N)$	exponenciální
O(N!)	faktoriálová

Obr. 2.1 - Tabulka algoritmických složitostí

2.2.4.2 Prostorová složitost

U prostorové složitosti se používá stejný postup určování jako u časové složitosti. Pouze se uvažuje namísto operační náročnosti paměťová náročnost. Např. lineární paměťová složitost udává, že využití paměti se zvýší přibližně tolikrát, kolikrát se zvýší počet vstupů.

Oba typy složitostí se velmi často ovlivňují, snížením jedné většinou dojde ke zvýšení druhé. Např. pro zvýšení rychlosti, tj. snížení počtu kroků algoritmu, musíme ukládat více mezivýsledků, a tím dojde ke zvětšení paměťové náročnosti.

2.3 Zápis algoritmů

Při zápisu algoritmu je nutné si uvědomit, že se nejedná o program jako takový, ale o postup při řešení určité konkrétní situace. Proto není použití konkrétního programovacího jazyka nezbytné, v určitých případech ani vhodné. Jednotlivé možnosti zápisu algoritmů nyní podrobněji popíši.

2.3.1 Přirozený jazyk

Jednou z možností popisu algoritmů je použití běžného jazyka. Jedná se o sekvenci kroků, kde každý krok je popsán jednou nebo více větami v běžném jazyce. Nespornou výhodou tohoto postupu je srozumitelnost pro každého, bez nutnosti znát jakýkoliv konkrétní programovací jazyk. Proto může být pochopení algoritmu rychlejší. Zásadní nevýhodou je naproti tomu možná víceznačnost běžných jazyků a z toho vyplývající formální neúplnost.

Pro pochopení algoritmu je použití běžného jazyka vhodné, ale jeho převedení do programové podoby je poté naopak složitější. V předmětu Algoritmy se přirozený jazyk pro popis algoritmů nepoužívá, ale jiné předměty, např. Základy umělé inteligence, tento postup používají. Proto zde jako příklad algoritmu zapsaného přirozeným jazykem uvedu algoritmus z [5].

1. *Sestroj frontu OPEN (bude obsahovat všechny uzly určené k expanzi) a umísti do ní počáteční uzel.*
2. *Je-li fronta OPEN prázdná, pak úloha nemá řešení a ukonči proto prohledávání jako neúspěšné. Jinak pokračuj.*
3. *Vyber z čela fronty OPEN první uzel.*
4. *Je-li vybraný uzel uzlem cílovým, ukonči prohledávání jako úspěšné a vrať cestu od kořenového uzlu k uzlu cílovému (vrací se posloupnost stavů nebo operátorů). Jinak pokračuj.*
5. *Vybraný uzel expanduj, všechny jeho bezprostřední následníky umísti do fronty OPEN a vrať se na bod 2.*

Obr. 2.2 - Algoritmus zapsaný přirozeným jazykem - metoda slepého prohledávání do šířky (BFS)

2.3.2 Programovací jazyk

Pro zápis algorimu lze použít přímo nějaký konkrétní programovací jazyk. Nemusíme použít celý program, stačí jen část kterou potřebujeme. Je ovšem potřeba zvolit vhodný programovací jazyk, který se v oblasti použití daného algoritmu využívá. Při tomto způsobu zápisu je nutné, aby čtenář znal velmi dobře daný jazyk, proto je výběr jazyka důležitý. Zároveň tento způsob neumožňuje zjednodušit algoritmus o pro něj nepotřebné informace, např. operace pro vstup a výstup.

Výhodou tohoto přístupu je okamžitá možnost použití bez nutnosti větších úprav. Naopak pochopení algoritmu zapsaného touto formou může být obtížnější. Jako příklad může posloužit libovolný program nebo jeho funkce implementujíci nějaký algoritmus.

V oblasti informatiky můžeme narazit na velké množství takových jazyků, které se dělí do mnoha typů a skupin, mají své výhody a nevýhody a různé oblasti použití. V této práci ale nebudu rozebírat velké množství programovacích jazyků. Zaměřím se pouze na jazyky, které mají spojitost s předmětem Algoritmy a tématem této práce. Jedná se o jazyk Pascal a jazyk C. Informace jsem čerpal z [8] a [9].

2.3.2.1 Pascal

Jazyk Pascal navrhl na počátku 70. let profesor Niklaus Wirth z Vysoké školy technické v Curichu. Hlavní myšlenkou tohoto návrhu bylo vytvořit jednoduchý a srozumitelný jazyk vhodný pro výuku programování. Název byl zvolen na počest francouzského filosofa, matematika a fyzika Blaise Pascala.

Pascal byl velmi dlouho využíván jako základní jazyk pro výuku algoritmů. Nejrozšířenější byla implementace Turbo Pascal od firmy Borland. Používal se i v předmětu Algoritmy a i nyní je zde využíván jako základ pro popis algoritmů. Na konci 90. let, ale začal být postupně nahrazován jazykem C, který je v praxi daleko více používán. V dnešní době je již jedinou používanou variantou jazyka Pascal jeho objektová nadstavba Object Pascal využívána v systému Delphi.

Narozdíl od ostatní jazyků, např. jazyku C, Pascal vyniká přehledností a čitelností. Jako příklad nám poslouží program pro výpis slov „Hello world!“.

```
program hello;

  begin
    writeln('Hello world!');
  end.
```

Obr. 2.3 - Program v jazyce Pascal

2.3.2.2 Jazyk C

Jazyk C je nízkoúrovňový, relativně minimalistický programový jazyk. Byl původně vyvinut Kenem Thompsonem a Dennisem Ritchiem pro potřeby operačního systému Unix. Dnes je ale velmi rozšířený i pro tvorbu běžných aplikací. Všechny implementované domácí úlohy i úprava systému pro zadávání domácích úloh jsou vytvořeny právě v tomto jazyce. Jedná se o v dnešní době velmi rozšířený a oblíbený programovací jazyk.

Jazyk C je vhodný pro většinu systémových aplikací jako jsou ovladače nebo úprava jádra operačního systému. Pokud se přesto potřebujeme dostat na nižší úroveň, můžeme využít tzv. inline assembler, neboli zápis assembleru přímo do kódu programu v jazyce C.

Pro alokaci paměti jazyk obsahuje jen minimální abstrakci. Pro práci s pamětí je definován typ ukazatel, který obsahuje odkaz na paměťový prostor pro daný typ proměnné. Vzhledem k tomu, že typ ukazatel existuje nezávisle na proměnných, na které se odkazuje, musíme dávat pozor, abychom se neodkazovali na nealokovanou paměť. Spravné pochopení a zvládnutí techniky práce s ukazateli je v Jazyce C velmi důležité a může zabránit nepříjemným paměťovým únikům a přístupům do paměťového prostoru, který nám nepatří.

Syntaxe jazyka C je velmi volná a dovoluje psát i velmi nečitelné programy, proto je nutné při zapisu programu správně odsazovat a dostatečně komentovat. Oproti jazyku Pascal používá místo některých klíčových slov speciální znaky, např. složené závorky pro blok příkazů. Jako příklad nám poslouží stejný program jako u jazyka Pascal, který vypíše „Hello world!“.

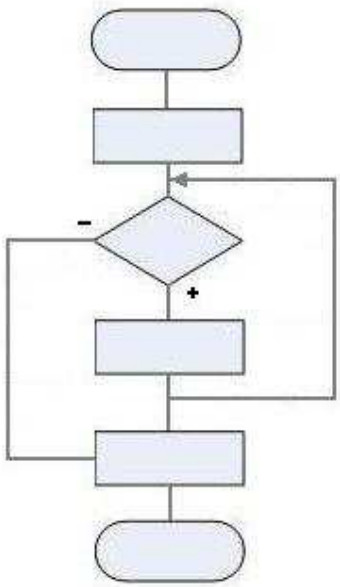
```
#include <stdio.h>

main()
{
  printf("Hello world!");
}
```

Obr. 2.4 - Program v jazyce C

2.3.3 Vývojový diagram

Další možností zápisu algoritmu je použít vývojové diagramy. Jedná se o grafickou formu znázornění algoritmů. Informace jsem čerpal z [10] a [11].

Vývojové diagramy jsou jedním z nejčastěji používaných prostředků pro popis algoritmů. Skládají se z grafických značek, do kterých jsou vepisovány upřesňující údaje. Tvary a velikosti značek jsou dány normami. Značky jsou spojeny čarami a znázorňují tak posloupnost jednotlivých kroků algoritmu. Vývojový diagram čteme ve směru shora dolů. Pokud potřebujeme pořadí čtení značek upravit, např. u cyklů (Obr. 2.5), můžeme čáry mezi značkami doplnit o šipky, které znázorňují směr zpracování.



Obr. 2.5 - Vývojový diagram pro cyklus se vstupní podmínkou

Jedná se o přehledný, ale poměrně pracný a rozsáhlý způsob zápisu algoritmů. Složitější diagramy se zpravidla nevejdou na jednu stránku.

2.3.4 Strukturogram

Pro úspornější znázornění algoritmů můžeme použít strukturogram. Je tvořen tabulkou, kde do jednotlivých řádků zapisujeme pořadí kroků v pořadí, v jakém budou prováděny. Výhodou tohoto způsobu zápisu je přehlednost i pro složitější algoritmy a jednoznačný a poměrně snadný přepis do formálního jazyka. Strukturogram se používá při návrhu algoritmu shora dolů.

Informace jsem čerpal z [11].

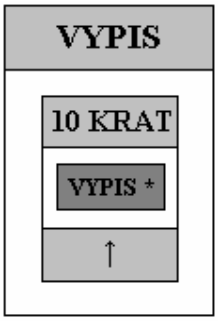
<table>
<tr><td colspan="5">Řešení kvadratické rovnice</td></tr>
<tr><td colspan="5">Čti a, b, c</td></tr>
<tr><td colspan="5">Spočti diskriminant D=b*b -4*a*c</td></tr>
<tr><td colspan="5">Je D>0?</td></tr>
<tr><td rowspan="4"></td><td>Ano</td><td colspan="3">Ne</td></tr>
<tr><td>x1=(-b+sqrt(D))/(2*a)</td><td colspan="3">Je D=0 ?</td></tr>
<tr><td>x2=(-b-sqrt(D))/(2*a)</td><td rowspan="3"></td><td>Ano</td><td>Ne</td></tr>
<tr><td rowspan="2">Tisk 2 kořeny:
x1, x2</td><td>x1=(-b+sqrt(D))/(2*a)</td><td rowspan="2">Tisk:
Nemá v R řešení.</td></tr>
<tr><td>Tisk 1 kořen:
x1</td></tr>
</table>

Obr. 2.6 – Strukturogram pro řešení kvadratické rovnice

2.3.5 Kopenogram

Kopenogram byl vyvinut pro výukový jazyk Karel. Název je odvozen od jmen jeho autorů: Kofránek, Pecinovský a Novák. Jedná se o další grafickou formu zápisu algoritmů. Informace jsem čerpal z [12].

Algoritmus v kopenogramu je rozdělen na dvě části. Název algoritmu je od jeho těla oddělen dvojitou čarou a je podbarven žlutě, tělo algoritmu by mělo být podbarveno červeně. Tělo algoritmu je tvořeno blokem příkazů. Příkazy mohou být různých typů, např. při podmínce se příkaz rozdělí na dva sloupce pro splněnou i nesplněnou podmínku. U cyklů má horní i dolní část pro opakování zelené podbarvení. U kopenogramu lze velmi jednoduše zapsat rekurzi, když pouze namísto běžného příkazu napíšeme název celého algoritmu.



Obr. 2.7 - Kopenogram pro výpis deseti znaků *.

3 Abstraktní datové struktury

U datových struktur byla velmi silná vazba na technické vlastnosti počítače. Používaly se datové struktury, které byly přímo reprezentovatelné v paměti počítače a pro něž existovaly odpovídající operace přímo v repertoáru základních strojových instrukcí. Jiné typy datových struktur se příliš nepoužívaly, především z obavy malé rychlosti a velké spotřeby paměti.

Naopak u řídících struktur byla situace opačná. Již v jazyce Algol 60 se objevil složený příkaz, podmíněný příkaz a příkaz cyklu. Tyto příkazy umožňily výraznější odpoutání algoritmizace od úrovně strojových instrukcí a přiblížily ji úrovni lidského myšlení.

Obrat v posuzování datových struktur přinesly nové poznatky o souvislosti datových struktur s efektivností algoritmů. Výzkumy ukázaly, že použití zdánlivě složitých datových struktur může vést ke vzniku daleko účinějších algoritmů.

Informace jsou čerpány z [1] a [4].

3.1 Abstraktní datový typ

Pokud u datové struktury potlačíme její vnitřní chování neboli to, jakým způsobem jsou data zobrazena a jakým způsobem se nad nimi operace provádějí a naopak zdůrazníme její vnější chování, tzn. co data reprezentují a co s nimi operace provádějí, získáme abstraktní datovou strukturu (ADS). Abstraktní datové struktury se shodnými vlastnostmi vytvářejí tzv. abstraktní datové typy (ADT).

Abstraktní datový typ tedy získáme zvýrazněním nejdůležitějších vlastností datového typu a pominutím ostatních, především technických a implementačních. Každý ADT má definované své rozhraní neboli množinu operací pro práci s ním. Jakýkoliv přístup nebo manipulace s daty ADS se děje výhradně přes toto rozhraní. Implementace takovýchto operací bývá uzavřená do samostatných programových celků a přímo nesouvisí s používáním ADS. Pokud změníme implementaci některé operace ADT, např. pro zvýšení rychlosti, tak program, který s tímto ADT pracuje, zůstává nezměněn.

Všechny tyto vlastnosti výrazně zjednodušují práci s abstraktními datovými typy. Obrovskou výhodou je i vysoká míra standardizace operací nad jednotlivými druhy ADT, která urychluje návrh a tvorbu programu.

3.2 Příklady běžných abstraktních datových typů

Implementace a používání abstraktních datových typů je jednou z nejdůležitějších oblastí v předmětu Algoritmy, především u domácích úkolů. Proto v této kapitole popíši nejvýznamnější z nich.

3.2.1 Seznam

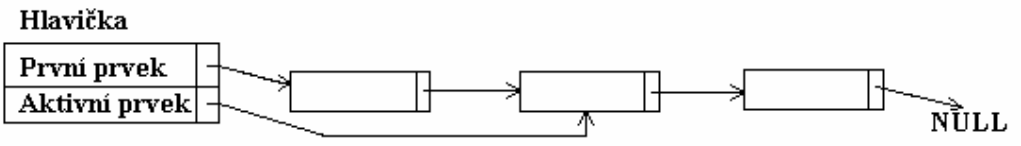
Seznam je jednou z nejobecnějších datových struktur. Jedná se o lineární, homogenní, dynamickou datovou strukturu. Lineárnost seznamu znamená, že každému prvku lze přiřadit právě jednoho předchůdce a jednoho následníka s výjimkou prvního a posledního prvku.

Prvkem seznamu může být libovolný datový typ, např. také seznam. Seznam může být, v závislosti na velikosti paměťového prostoru, libovolně rozsáhlý, příp. i prázdný. Přístup k prvnímu (okrajovému) prvku seznamu je přímý. K ostatním prvkům je přístup sekvenční ve směru průchodu.

Rozlišujeme tři druhy seznamů, které zde podrobněji rozeberu.

3.2.1.1 Jednosměrný seznam

Jendnosměrně vázáný lineární seznam se skládá, stejně jako ostatní abstraktní datové typy, z dat a operací, které s těmito daty manipulují.



Obr. 3.1 – Příklad jednosměrného seznamu

Struktura dat

Základní položkou jednosměrného seznamu je tzv. hlavička. Obsahuje ukazatel na první položku seznamu. Pokud je seznam prázdný obsahuje hodnotu NULL. Součástí hlavičky je i ukazatel na aktivní prvek seznamu. Co je to aktivita seznamu si vysvětlíme později. Pokud přistupujeme k položkám seznamu, přistupujem k nim práve přes tuto hlavičku.

Jak již bylo uvedeno, prvkem seznamu může být jakýkoliv jiný datový typ. Kromě těchto dat obsahuje každá položka navíc ukazatel na svého nejbližšího následníka. Tento způsob nám umožní sekvenčně procházet seznamem od první položky až po poslední. Poslední položka obsahuje tento ukazatel také, ale jeho hodnota je NULL.

Operace nad ADT

Nad datovým typem jednosměrný seznam mohou být definovány různé typy operací. Jako příklad zde použiji množinu operací, která se používá v předmětu Algoritmy. Nejprve je ale nutné si vysvětlit pojem aktivita seznamu, který se v těchto operacích používá.

K seznamu lze přistupovat přes první prvek nebo přes aktivní prvek. V seznamu může být aktivní prvek vždy právě jeden (seznam je aktivní) nebo žádný (seznam je neaktivní). Aktivita prvku v seznamu se musí nastavit, nově vytvořený seznam není aktivní.

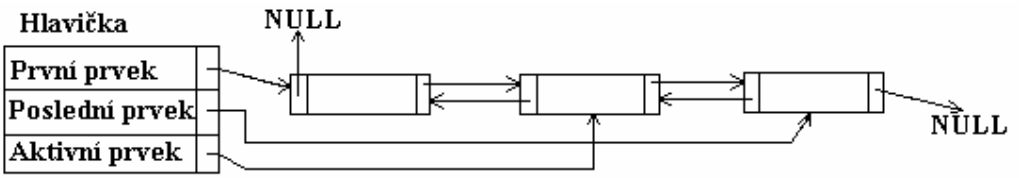
Pro práci s jednosměrným seznamem jsou v předmětu Algoritmy definovány následující operace:

- Inicializace seznamu. Vytvoření nového prázdného seznamu.
- Vložení prvku na začátek seznamu. Jedná se o jedinou možnost jak vložit prvek do prázdného seznamu.
- Získání hodnoty prvního prvku. Při prázdném seznamu způsobí chybu a ukončí program
- Nastavení prvního prvku aktivním. Nad prázdným seznamem nemá účinek.
- Smazání prvního prvku. Smaže první prvek seznamu a pokud byl aktivní, aktivita se ztrácí. Nad prázdným seznamem nemá účinek.
- Posun aktivity na další prvek. Pokud je aktivní prvek poslední, aktivita se ztrácí. Nad neaktivním seznamem nemá účinek
- Test aktivity seznamu. Vrací „true“ nebo „false“.
- Vložení prvku za aktivní prvek. Nad prázdným seznamem nemá účinek.
- Smazání prvku za aktivním prvkem. Pokud je aktivní prvek poslední nebo seznam neaktivní, operace nemá účinek.
- Získání hodnoty aktivního prvku. Pokud je seznam neaktivní, způsobí chybu a ukončí program.
- Přepsání hodnoty aktuálního prvku. Nad neaktivním seznamem nemá účinek.

Jak můžeme vidět, množina operací pro tento abstraktní datový typ je poměrně veliká a to jsou uvedeny pouze základní operace. Možnosti využití seznamů jsou značné, z pohledu algoritmů se používají jako základ pro jiné ADT, např. zásobník.

3.2.1.2 Dvojsměrný seznam

Struktura i operace nad ADT dvojsměrně vázáný lineární seznam vychází z jeho jednosměrné varianty. Nejdůležitějším rozšířením je provázání seznamu v obou směrech, tedy možností procházet seznam v opačném pořadí od konce po počátek.



Obr. 3.2 – Příklad dvojsměrného seznamu

Z obrázku je patrné, že každá položka dvojsměrného seznamu obsahuje, kromě ukazatele na bezprostředně následující prvek, i ukazatel na prvek, který mu bezprostředně předchází. U prvního

prvku seznamu je ukazatel na předchozí prvek roven hodnotě NULL. Stejnou hodnotu u posledního prvku nabývá ukazatel na následující prvek. Hlavička seznamu je navíc nově doplněna o ukazatel na poslední prvek seznamu.

Množina operací vychází také z jednosměrného seznamu. Využívá všechny jeho operace a s pomocí nových ukazatelů na předchozí a poslední prvek je rozšiřuje o další operace. Jedná se o následující operace:

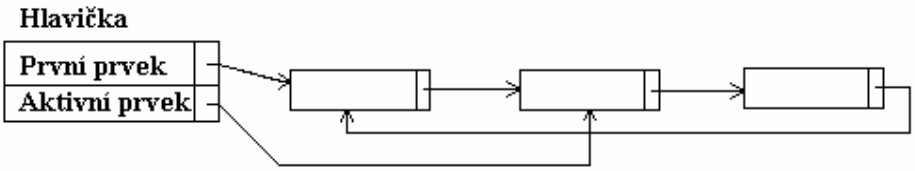
- Vložení prvku na konec seznamu.
- Smazání posledního prvku seznamu.
- Nastavení aktivity na poslední prvek.
- Přečtení hodnoty posledního prvku.
- Vložení prvku před aktivní prvek.
- Smazání prvku, který je před aktivním prvkem.
- Posun aktivity na předcházející prvek.

Dvojsměrný seznam nám tedy umožňuje, jako hlavní rozšíření oproti jednosměrnému, procházení seznamu v obou směrech a tím i širší možnost použití.

3.2.1.3 Kruhový seznam

Kruhový seznam je speciálním případem běžného lineárního seznamu. Může být implementován jako jednosměrný i dvousměrný. Nejvýznamnější změnou je, že kruhový seznam nemá začátek ani konec. Může být např. vytvořen z klasického seznamu provázáním prvního a posledního prvku.

Pro přístup ke kruhovému seznamu je nutné definovat prvek, který je chápán jako vstupní bod neboli „první“ prvek seznamu. Ukazatel na tento prvek je spolu s ukazatelem na aktivní prvek součástí hlavičky seznamu. Operace nad kruhovým seznamem jsou odvozeny od operací s běžným seznamem.



Obr. 3.3 – Příklad kruhového seznamu

3.2.2 Zásobník

Abstraktní datový typ zásobník je stejně jako seznam lineární, homogenní, dynamická datová struktura. Implementována může být pomocí pole nebo lineárního seznamu. Jedna z domácích úloh se věnuje právě implementaci zásobníku pomocí jednosměrného seznamu (kapitola 4.3.1).

Funkčnost zásobníku nejlépe chrarkterizuje anglická zkratka LIFO (Last in, first out), která říká, že poslední vložený prvek je ze zásobníku vyjmut jako první. Díky této vlastnosti zásobník invertuje pořadí vložených prvků. Operace definované nad zásobníkem pracují výhradně s prvním prvkem zásobníku, proto je nutné přístupu k ostatním prvkům zamezit. Nad ADT zásobník jsou definovány tyto operace:

- Vytvoření prázdného zásobníku.
- Vložení prvku na vrchol zásobníku
- Zrušení prvku na vrcholu zásobníku
- Přečtení prvku z vrcholu zásobníku. Při prázdném zásobníku způsobí chybu.
- Test prázdnosti zásobníku.

V praxi se často operace pro přečtení a zrušení prvku na vrcholu zásobníku spojují do jedinné operace. Z hlediska teorie je ale vhodné zachovat zásadu, jedna operace znamená jednu činnost.

Zásobník patří mezi nejvýznamnější ADT a má široké aplikační použití. Např. inverze lineárního pořadí, přidělování paměti, předávání parametrů nebo zpracování výrazů.

3.2.3 Fronta

Fronty nacházejí uplatnění především v oblasti „hromadné obsluhy“. Její název a funkčnost je odvozena od stejného pojmu v běžném životě. Na jednom konci fronty (konec fronty) se prvky vkládají a na druhém konci (začátek fronty) se odstraňují. Jedná se o datový typ velmi podobný zásobníku, ale na rozdíl od něj pracuje s oběma konci lineární struktury. Funkčnost fronty opět nejlépe vystihuje anglická zkratka FIFO (First in, first out), která říká, že první vložený prvek je z fronty vyjmut nejdříve.

Operace definované nad ADT fronta se dají velmi lehce odvodit od operací nad ADT zásobník. Jedná se o tyto operace:

- Vytvoření prázdné fronty.
- Vložení prvku na konec fronty.
- Zrušení prvku na začátku fronty.
- Přečtení prvku na začátku fronty.
- Test prázdnosti fronty.

Stejně jako u zásobníku se v praxi funkčnost operace přečtení a zrušení prvku spojuje do jedinné operace.

3.2.4 Vyhledávací tabulka

Funkčnost abstraktního datového typu vyhledávací tabulka je odvozena od objektu, kterému se říká kartotéka. Každá položka tabulky obsahuje složku, která se nazývá klíč. Tento klíč jednoznačně identifikuje každou položku tabulky. Z toho vyplývá, že hodnoty klíčů v tabulce musí nabývat

jednoznačných, navzájem různých hodnot. Nejzákladnější operací vyhledávací tabulky je vyhledání prvku se zadaným klíčem.

ADT vyhledávací tabulka není součástí mnou vypracovaných domácích úloh. K jejich pochopení není potřebná, proto zde neuvádím její podrobnější specifikaci a je zde uvedena spíše pro úplnost.

3.2.5 Pole

Pole je homogenní lineární datová struktura. Obsahuje prvky, libovolného datového typu, a každému z nich je navíc přiřazena speciální hodnota, tzv. index. Podle hodnot těchto indexů jsou prvky pole lineárně vzestupně uspořádány. Zadáním hodnoty indexu také přistupujeme k danému prvku pole.

Pokud má pole pevně definovaný počet prvků, jedná se o tzv. statické pole. Naopak, pokud se jeho velikost při běhu programu může měnit, nazýváme takové pole dynamickým.

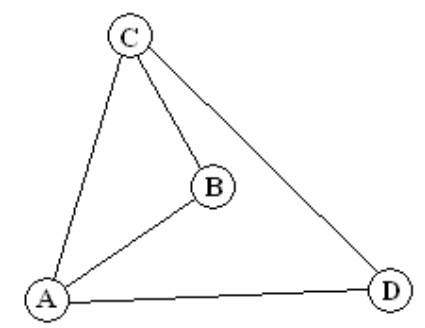
Stejně jako vyhledávací tabulka je zde i pole uvedeno spíše pro úplnost. Podrobnější popis těchto datových typů naleznete v použité literatuře.

3.2.6 Graf

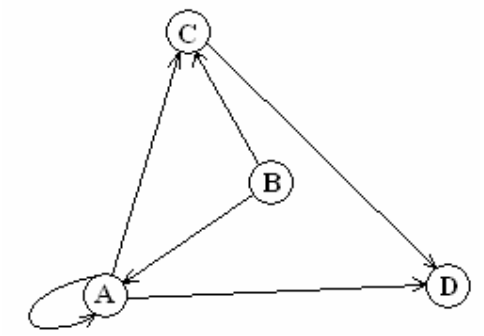
Graf je velmi obecný, nelineární abstraktní datový typ, který má široké možnosti uplatnění. Jeho použití si studenti vyzkouší např. v předmětu Základy umělé inteligence. Jeden z algoritmů tohoto předmětu jsem použil v domácí úloze pro práci s grafem (kapitola 4.3.3).

Graf je definován jako trojice G=(N,E,I), kde N je množina uzlů, jimž lze přiřadit hodnotu, E je množina hran, kterým lze přiřadit hodnotu a I je množina spojení, která jednoznačně určuje spojení dvojic uzlů daného grafu.

Pokud jsou hrany grafu orientované, pak takový graf nazýváme orientovaný graf. V opačném případě se jedná o neorientovaný graf.



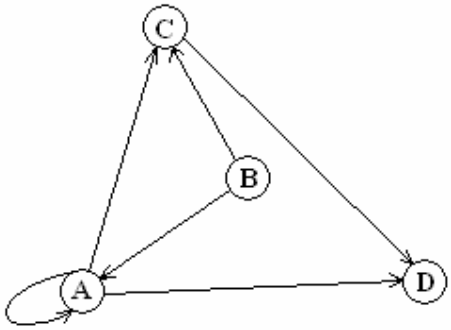
Obr. 3.4 – Příklad neorientovaného grafu



Obr. 3.5 – Příklad orientovaného grafu

Důležitým pojmem v teorii grafů, který je použit v již zmiňované domácí úloze, je cesta. Jedná se o posloupnost hran, po kterých se dostaneme z počátečního uzlu do koncového, aniž bychom šli po některé hraně dvakrát. Pomocí cesty poté definujeme další důležitý pojem, kterým je cyklus. Cyklusem nazýváme neprázdnou cestu, která končí a začíná v témže uzlu. Graf, který cyklus obsahuje nazýváme cyklický graf. Graf, který naopak cyklus neobsahuje nazýváme acyklický.

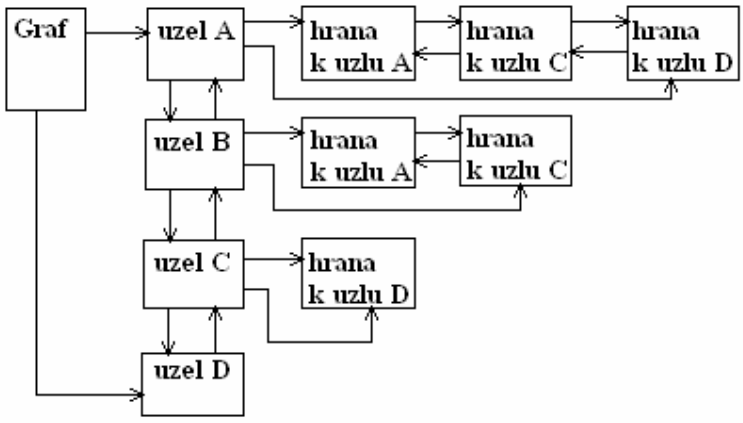
Implementovat graf můžeme více způsoby. Jako první si ukážeme implementaci pomocí matice koincidence, neboli spojení. Jedná se v podstatě o tabulku, ve které sloupce představují počáteční uzly a řádky uzly koncové. Pro každý uzel poté tabulku vyplníme hodnotou 1, pokud je s příslušným uzlem propojen hranou. V případě orientovaného grafu rozlišujeme, jestli v daném uzlu hrana začíná (hodnota 1) nebo zda v daném uzlu končí (hodnota -1). Matice koincidence bývá implementována pomocí dvojrozměrného statického pole. Nevýhodou tohoto způsobu je neměnná velikost grafu.



	A	B	C	D
A	1	-1	1	1
B	1	-	1	-
C	-1	-1	-	1
D	-1	-	-1	-

Obr. 3.6 - Matice koincidence pro orientovaný graf

Jako druhou možnost implementace si uvedeme dynamickou reprezentaci pomocí seznamů. V tomoto případě se omezíme na orientovaný graf, který je používán v domácí úloze s grafovým algoritmem (kapitola 4.3.3). Graf je v tomto případě reprezentován jako seznam uzlů, kde každý uzel osahuje opět seznam, který obsahuje hrany, které v tomto uzlu mají počátek. Pro lepší pochopení této grafové reprezentace je uveden obr. 3.7.



Obr. 3.7 - Dynamická reprezentace orientovaného grafu z obr. 3.5 pomocí seznamů

Nad ADT graf je definována následující základní množina operací:

- Vytvoření prázdného grafu.
- Vložení izolovaného uzlu do grafu.
- Odstranění izolovaného uzlu z grafu.
- Vytvoření spojení mezi dvěma uzly neboli vytvoření hrany.
- Odstranění hrany z grafu.
- Test na prázdnost grafu.
- Test na existenci dané hrany či uzlu v grafu.
- Test na existenci spojení mezi dvěma danými uzly.

Nad grafem existuje obrovská spousta různých dalších rozšiřujících algoritmů. Jedním z nich je např. hledání nejkratší cesty mezi dvěma uzly v grafu. Tento algoritmus je součástí již zmiňované domácí úlohy pro práci s grafem a jeho princip je popsán v příslušné kapitole.

3.2.7 Strom

Jako poslední abstraktní datový typ zde uvedu speciální formu grafu, kterou je ADT strom. Jedná se o orientovaný, acyklický, souvislý graf se speciálním uzlem, který se nazývá kořen. Pro kořen platí, že z každého uzlu stromu vede jen jedna cesta do kořene. Z každého uzlu také vede pouze jedna hrana směrem ke kořenu a libovolný počet směrem od kořene. Uzel, který bezprostředně předchází danému uzlu (směrem ke kořeni), nazýváme „otcovský“, naopak uzly, které bezprostředně následují nazýváme „synovské“.

Nejvyznámnějším představitelem stromu je binární strom. Má všechny základní vlastnosti obecného stromu, ale je omezen počtem možných synovských uzlů a to na maximálně dva uzly. Nad binárním stromem existuje velké množství algoritmů, které mohou mít jak rekurzivní tak nerekurzivní charakter.

4 Domácí úlohy

V této kapitole se zaměřím na nejduležitější část své bakalářské práce, na domácí úlohy pro předmět Algoritmy. V první podkapitole se budu zabývat smyslem domácích úloh a jejich významem pro studenta. Ve druhé podkapitole popíši strukuru těchto úloh, a to především z obecnějšího hlediska. Zmíním ale i duléžité implementační podrobnosti. Poslední část je věnována domácím úlohám, které jsem měl za úkol implementovat.

4.1 Význam domácích úloh

Domácí úlohy tvoří velmi významnou část mé práce i předmětu Algoritmy. Proto zde zmíním, dle mého názoru, proč jsou pro studium předmětu nezbytné. Dále uvedu několik zásad, které jsou pro tvorbu domácích úloh důležité a které jsem se snažil dodržovat.

4.1.1 Studijní hledisko

Studium předmětu Algoritmy, stejně jako většiny ostatních předmětů, se skládá z teoretické a praktické části. Znalosti z teorie může student získat bez problému na přednáškách, ale praktické znalosti získá pouze jejich používáním v konkrétních situacích.

K tomuto účelu jsou právě vhodné domácí úkoly. Při jejich plnění student aktivně používá své znalosti k řešení konkrétních situací. Naučí se používat programovací jazyk k mírně složitějším problémům. Student také není, narozdíl od počítačových laboratoří, tolik limitován časem. A navíc, struktura domácích úkolů se snaží studenta naučit dobrým programátorským návykům a správnému psaní programu.

Z těchto důvodů jsou domácí úlohy důležitou součástí předmětu Algoritmy.

4.1.2 Zásady pro tvorbu domácích úloh

4.1.2.1 Správnost

Správnost je nejdůležitější věc při vytváření domácích úloh. Úlohy musí dodržovat standardní pravidla pro daný programovací jazyk, v tomto případě jazyk C, jinak by student mohl získat špatné programovací návyky. V úloze by měl být použit jednotný typ, který je v daném jazyce standartní pro pojmenování proměnných, funkcí a datových struktur. Důležité je také správně a jednoznačně formulovat zadání a dostatečně komentovat. Častou chybou je používání zvyklostí jiného programovacího jazyka, např. jazyka Pascal.

Samozřejmostí je, že program nesmí obsahovat žádné chyby.

4.1.2.2 Složitost

Cílem předmětu je studenta naučit jistým znalostem, proto by úloha neměla být příliš jednoduchá. Student by měl nad problémem strávit určitý čas, aby danou problematiku dostatečně pochopil. Na druhou stranu by úloha měla procvičovat pouze látku předmětu, určitý algoritmus nebo datový typ a operace nad ním. Student by se neměl zabývat strukturou celého programu, ale pouze danou problematikou.

4.1.2.3 Rozsah

Domácí úloha by neměla být příliš rozsáhlá, aby student měl na každý problém dostatek času. Naopak by neměla být příliš krátká. Rozsah by měl být takový, aby si student procvičil všechny důležité části daného problému. Na úloze by měl student strávit tolik času, kolik se pro předmět Algoritmy předpokládá.

4.2 Struktura domácích úloh

Všechny domácí úlohy v předmětu Algoritmy mají pevně danou strukturu, která se musí dodržovat. Jednotlivé její částí, každá je reprezentována samostatným souborem, nyní podrobněji popíši. Jedná se o následující soubory:

- `cXXX.h` – hlavičkový soubor
- `cXXX.c` – zdrojový soubor
- `cXXX-test.c` – soubor se základními testy
- `cXXX-advanced-test.c` – soubor s pokročilými testy
- `makefile`

kde XXX je číslo domácí ulohy.

4.2.1 Hlavičkový soubor

Každý domácí úkol pro předmět algoritmy spočívá v implementaci funkcí nad daným abstraktním datovým typem. Struktura tohoto abstraktního datového typu, stejně jako prototypy funkcí, které s datovým typem pracují, jsou uvedeny právě v hlavičkových souborech.

Pro každý domácí úkol je vytvořen právě jeden hlavičkový soubor, který tak definuje jeho rozhraní. Jsou v něm uvedeny prototypy všech funkcí, které mají studenti za úkol implementovat. Většinou je zde uvedena i struktura použitého datového typu, případně definice konstant a proměnných. Rozdíl nastává, pokud se jedná o domácí úkol, kde se implementuje nějaký složitější algoritmus s využitím operací z jiného domácího úkolu. V tomto případě je k domácímu úkolu připojen ještě hlavičkový soubor této úlohy, kde je definice námi potřebných struktur a funkcí uvedena. Náš domácí úkol tedy poté obsahuje dva hlavičkové soubory.

Pro správné pochopení domácí úlohy je nezbytné, aby si student před samotnou implementací tyto hlavičkové soubory prošel a dobře pochopil.

4.2.2 Zdrojový soubor

Jedná se o hlavní část domácí úlohy, která obsahuje implementaci daných funkcí. Pokud se jedná o zadání, která mají studenti řešit, jsou těla funkcí prázdná. Obsahují pouze řádek s identifikací, že funkce nebyla řešena. Identifikace je provedena pomocí klíčového slova `solved` a v případě řešení studenti tento řádek smažou, případně zakomentují.

Z pohledu vzorového vypracování domácí úlohy jsou dané funkce samozřejmě implementované. Tělo hotové funkce je poté uzavřeno mezi speciální značky `/*v*/`. Pomocí tohoto speciálního znaku je poté možné z dané úlohy vytvořit jak zadání tak vzorové řešení. K tomuto účelu složí samostatný program zvaný interpret (kapitola 5.2).

4.2.3 Základní testy

Ke každému domácímu úkolu jsou přiloženy základní testy. Jedná se o soubor testů, pomocí nichž si student může vyzkoušet základní funkčnost a správnost implementace svého řešení. Jedná se však pouze o základní testy a pokud chceme otestovat plnou správnost řešení, je potřeba si testy rozšířit. Nejsou zde zastoupeny zejména testy na mezní hodnoty.

Za splnění základních testů student získává první část bodů za domácí úlohu.

4.2.4 Pokročilé testy

Jedná se o soubor testů, které již student nemá k dispozici. Tyto testy se snaží o co největší možnou míru kontroly daného řešení, především o kontroly mezních hodnot a stavů nebo nepovolených vstupů. Za jejich splnění student získává druhou část bodů.

4.2.5 Makefile

Makefile v domácích úloháh neplní pouze úlohu překladu zdrojových souborů, ale s pomocí programu interpret (kapitola 5.2) také vytváří zadání a vzorové řešení. Pro lepší pochopení zde uvádím výčet jeho funkcí ve formátu „příkaz - funkčnost“.

- `make` – Vytvoří spustitelný soubor pro soubor se základními testy.
- `make tests` – Vytvoří spustitelný soubor pro základní i pokročilé testy.
- `make zadani` – Vytvoří zadaní pro studenty obsahující všechny potřebné soubory a makefile pro překlad základních testů.
- `make reseni` – Obdobně jako zadání vytvoří stejným způsobem vzorové řešení.
- `make clean` – Smaže všechny soubory, které vytvořil.

Makefile pro nově implementované úlohy je díky nové verzi interpretu (kapitola 5.3) mírně pozměněn. Neobsahuje příkazy pro vytvoření obecného zadání a řešení, ale nahrazuje je následujícími:

- `make zadaniCZ` – Vytvoří zadani v českém jazyce.
- `make zadaniEN` – Vytvoří zadání v anglickém jazyce.
- `make reseniCZ` – Vytvoří vzorové řešení v českém jazyce
- `make reseniEN` – Vytvoří vzorové řešení v anglickém jazyce.

Student se zadáním domácí úlohy samozřejmě nedostává celý makefile, ale pouze jeho část pro překlad základních testů.

4.3 Implementace domácích úloh

V této kapitole popíši domácí úlohy, které jsem měl za úkol implementovat.

4.3.1 c209 - Zásobník

Zadáním této domácí úlohy je implementovat základní operace nad ADT zásobník pomocí jednosměrně vázaného lineárního seznamu. Jedná se o implementačně poměrně jednoduchou úlohu, která ale nutí studenta využít operace, které již implementoval dříve. Tento typ úlohy by měl studentovi ukázat, že při psaní programu nemusí vytvářet všechny jeho části od počátku, ale může využít již dříve implementovaných částí.

Samotný datový typ zásobník je v této úloze pouze předefinovaný typ jednosměrný seznam. Student tedy vlastně pracuje se seznamem. Pro operace se zásobníkem by proto student měl používat pouze operace z úlohy c201. Na tuto skutečnost je upozorňován na několika místech ve zdrojových souborech.

Úloha obsahuje implementaci následujících operací:

- SInit – Inicializace zásobníku před jeho prvním použitím.
- SPush – Vložení prvku na vrchol zásobníku.
- SPop – Zrušení prvku na vrcholu zásobníku.
- STop – Přečtení prvku z vrcholu zásobníku.
- SEmpty – Test prázdnosti zásobníku.
- SDispose – Smazání zásobníku.

Při zadání této úlohy student jako první úkol dostane úlohu c201, tedy implementaci základních operací nad ADT jednosměrně vázaný lineární seznam. Úloha c209 pak slouží jako navazující a student při ní využívá svých operací z předešlé úlohy. Tento způsob zadání motivuje studenta k větší správnosti psaní domácích úloh, protože případné chyby z úlohy c201 se poté projeví i při této úloze.

4.3.2 c210 – Operace nad ADT seznam

Jedná se o středně náročnou úlohu, ve které studenti implementují vybrané pokročilejší operace nad ADT dvojsměrně vázaný lineární seznam. Studenti by při implementaci těchto pokročilejších algoritmů měli využívat základní operace z úlohy c206 (dvojsměrně vázaný lineární seznam).

Při zadání této úlohy se využívá stejný postup jako při předchozí úloze c209. Student tedy dostane jako první úlohu c206 a úloha c210 pak slouží jako navazující.

Implementovány jsou následující operace:

- DLLength – Zjistí délku seznamu.
- DLEquLists – Zjistí zda jsou dané seznamy shodné.
- DLCopyList – Vytvoří kopii seznamu.
- DLSearch – Vyhledá prvek se zadanou hodnotou v seznamu. Pokud je prvek nalezen, vrací pozici jeho prvního výskytu.
- DLReverse – Obrátí pořadí prvků v seznamu.

4.3.3 c602 – Grafový algoritmus

Nejzajímavější, ale zároveň nejtěžší, je poslední domácí úloha implementující algoritmus vyhledání nejkratší cesty v orientovaném grafu. Jako algoritmus pro vyhledání nejkratší cesty jsem použil metodu slepého prohledávání do šířky (BFS) z předmětu Základy umělé inteligence. Tento algoritmus vyhledává cestu v grafu a skončí, jakmile první z nich nalezne, případně skončí neúspěchem. Pro pochopení algoritmu zde uvádím jeho slovní popis z [5].

1. *Sestroj dva prázdné seznamy, frontu OPEN (bude obsahovat všechny uzly určené k expanzi) a CLOSED (bude obsahovat seznam expandovaných uzlů). Do fronty OPEN umísti počáteční uzel.*
2. *Je-li fronta OPEN prázdná, pak úloha nemá řešení a ukonči proto prohledávání jako neúspěšné. Jinak pokračuj.*
3. *Vyber z čela fronty OPEN první uzel a umísti tento uzel do seznamu CLOSED.*
4. *Je-li vybraný uzel uzlem cílovým, ukonči prohledávání jako úspěšné a vrať cestu od kořenového uzlu k uzlu cílovému (vrací se posloupnost stavů). Jinak pokračuj.*
5. *Vybraný uzel expanduj, všechny jeho bezprostřední následníky, kteří nejsou ani ve frontě OPEN, ani v seznamu CLOSED, umísti do fronty OPEN a vrať se na bod 2.*

Obr. 4.1 - Princip metody BFS

Z popisu algoritmu je patrné, že pro úspěšné zvládnutí této úlohy bude student nucen použít tři typy abstraktních datových struktur. Jedná se o orientovaný graf, frontu a seznam. Úloha tedy otestuje velký okruh znalostí z předmětu Algoritmy.

Jako základ pro strukturu prohledávaného grafu je použita úloha c601. Fronta pro uzly určené k expanzi a seznam, již expandovaných uzlů, jsou implementovány pomocí dojsměrně vázaného

lineárního seznamu. Informace do těchto seznamů jsou ukládány jako dvě hodnoty, aktuální uzel a předcházející uzel. Tento formát umožňuje následné zpětné získání nalezené cesty. Navíc je zde ještě zastoupen jeden seznam, který je použit pro uložení nalezené cesty.

Studenti mají algoritmus slepého prohledávání do šířky pouze doporučený, mohou tedy implementovat kterýkoliv jiný. V úloze se hodnotí pouze výsledky, nikoliv zvolený algoritmus. Kontrola nalezení cesty v grafu, kde se vyskytuje více než jedna nejkratší cesta, mohlo by tedy dojít k více správným řešením, je obsažena pouze v základních testech. Student si proto může otestovat, zda jeho algoritmus pracuje stejným způsobem jako algoritmus slepého prohledávání so šířky, tzn. dává stejné výsledky, jaké se v testech očekávají.

Podrobnější informace o implemntaci naleznete přímo ve zdrojových souborech této úlohy.

5 Systém pro zadávání a hodnocení domácích úloh

Tato kapitola se zabývá poslední částí mé bakalářské práce, systémem pro zadávání a vyhodnocování domácích úloh. V první části stručně popíši jednotlivé části tohoto systému. Ve druhé se podrobně zaměřím, na z pohledu zadání nejdůležitější část systému, program interpret a v poslední části představím implementovaná rozšíření tohoto programu.

5.1 Popis systému

Systém pro zadávání a hodnocení domácích úloh se skládá z několika skriptů napsaných v jazycích PHP, Bash a Perl. Součástí systému je i speciální program interpret, který slouží pro generování zadání a vzorového řešení. Nyní popíši jednotlivé části tohoto sytému vyjma programu interpret, jehož podrobnější popis naleznete v kapitole 5.2.

5.1.1 Skript saferun.pl

Pomocí tohoto skriptu jsou spouštěny vypracované domácí úkoly. Spouštění každého úkolu je řešeno jako samostatný proces a díky tomuto ošetření případný pád jednoho testu neovlivní úspěšné dokončení ostatních.

5.1.2 Skript checker.php

Skript slouží pro porovnání vzorových výstupů domácí úlohy a úloh vypracovaných studenty. Výsledkem skriptu je výpis úspěšnosti dané domácí úlohy v jednotlivých testech.

5.1.3 Skript points.sh

Skript podle výsledku předchozího skriptu checker.php a zadané hodnotící stupnice přidělí jednotlivým studentům bodové ohodnocení. Tuto stupnici je nutné pro spuštění skriptu definovat.

5.1.4 Skript preparemails.sh

Tento skript připravuje podobu e-mailů, které obsahují hodnocení dané domácí úlohy.

5.1.5 Skript sendmails.sh

E-maily vytvořené skriptem preparemails.sh jsou tímto skriptem rozeslány studentům.

5.1.6 Skript findduplicates.php

Jak již název skriptu napovídá, jeho funkcí je odhalování plagiátorství v domácích úlohách. Nastíním zde postup, který skript používá. V první řadě skript odstraní všechny komentáře ze zdrojových souborů. Již pomocí tohoto způsobu je možné několik plagiátů odhalit. Následně skript nahradí všechny konstatní řetězce za prázdné a odstraní netisknutelné znaky, neboli zruší formátování zdrojového textu. Poté seřadí řádky podle abecedy a vytvoří kontrolní součet. Kontrolní součty domácích úloh jsou poté porovnávány.

5.1.7 Skript checkall.sh

Jedná se o hlavní skript systému. Skript si nejprve zjistí, které domácí úlohy budou kontrolovány a přiřadí jim příslušné testy. Následně se zkompiluje vzorové řešení a vygenerují se vzorové výsledky. Zkompilují se také domácí úkoly studentů a spustí se pomocí skriptu saferun.pl. Nakonec se provede vyhodnocení výsledků pomocí skriptu checker.php.

5.2 Interpret

Jedním z úkolů této práce je modifikování programu interpret, proto ho zde podrobně popíši.

Jedná se o program v jazyce C, který se stará o generování zadání, případně vzorového řešení. Pracuje na principu konečného automatu, který vyhledává ve vstupu speciální značky a pomocí nich upravuje výstup. Těmito značkami jsou formátovany všechny soubory domácích úkolů. První výskyt dané značky znamená začátek formátovaného výstupu a druhý výskyt stejné značky jeho konec. Tyto značky se samozřejmě ve výstupu programu nevyskytují. Jedná se o tyto značky:

- `/**/` – Text, který není uzavřen mezi těmito značkami nebude programem interpret zpracováván. Na první pohled se zdá tahle značka přebytečná, má ale svůj význam při generování programu makefile pro zadání domácího úkolu. Při správném naformátování programu makefile generuje jenom jeho potřebnou část.
- `/*v*/` – Označuje části, které budou předány na výstup pouze v případě vzorového řešení.

Kromě těchto dvou značek obsahuje každá funkce ve zdrojovém souboru navíc řádek uvozený značkou `//` následovaná indikací řešení dané funkce. V případě zadání se tento řádek vypíše, v případě vzorového řešení nikoliv.

Funkce interpretu se zadává pomocí volitelného parametru `-v` . V případě zadání se tento parametr neudává, naopak pokud potřebujeme vzorové řešení musíme tento parametr zadat.

5.3 Nová verze interpretu

Zadáním této práce je také modifikování programu interpret takovým způsobem, aby bylo možné vytvářet zadání a vzorová řešení v anglickém i českém jazyce. Při implementaci těchto rozšíření jsem se snažil o plné zachování funkčnosti předchozí verze interpretu (kapitola 5.2), proto veškeré informace uvedené v předchozí kapitole platí i pro novou verzi interpretu.

Pro rozlišení anglického a českého textu jsem zvolil dvě nové značky:

- `/*cz*/` – pro český text
- `/*en*/` – pro anglický text

Veškerý text včetně komentářů je poté v nově implementovaných úlohách uváděn dvojjazyčně a uzavřen do těchto značek. Toto rozšíření ale neplatí pro testové soubory , které jsou uváděny pouze s anglickým komentářem. Tento text nové formátovací značky nepoužívá, aby došlo k jeho vygenerování v české i anglické variantě.

Nová verze interpretu se spouští se dvěma volitelnými parametry, parametr pro výběr jazyka a parametr `-v` pro výběr zadání či řešení. Interpret je možné spustit těmito způsoby:

- `interpret -cz` – Vytiskne české zadání.
- `interpret -cz -v` – Vytiskne české vzorové řešení.
- `interpret -en` – Vytiskne anglické zadání.
- `interpret -en -v` – Vytiskne anglické vzorové řešení.

Na nově implementované úlohy můžeme použít i interpret pouze s volitelným parametrem `-v`, výsledný kód je ale poté zcela bez komentářů. Při použití této varianty na starší úlohy funguje stejně jako starší verze interpretu.

6 Závěr

Celá tato práce se zabývá problematikou domácích úloh pro předmět Algoritmy. Výsledkem mé práce je vytvoření tří nových úkolů. Jedná se o implementaci zásobníku pomocí jednosměrně vázaného lineárního seznamu, implementaci pokročilých operací nad dvousměrným seznamem a hledání nejkratší cesty v orientovaném grafu.

První dvě úlohy jsou nadstavbou nad již dříve vytvořenými domácími úkoly pro práci s jednosměrně a dvojsměrně vázaným lineárním seznamem. Umožňují tedy vytvořit sérii úloh, kde student implementuje v první části základní operace nad těmito strukturami, ve druhé pak s využitím svých základních operací implementuje ty pokročilejší. Poslední úloha je zajímavá tím, že pro implementaci grafového algoritmu student musí použít kromě grafu další dvě abstarktní datové struktury, seznam a frontu.

Druhou částí mé práce bylo doplnit automat pro tvorbu zadání a vzorového řešení o možnost anglické nebo české jazykové varianty. Při implementaci rozšíření jsem se snažil o zachování plné zpětné kompatibility se staršími verzemi domácích úkolů. Součástí tohoto rozšíření byla i úprava souboru makefile pro nové domácí úlohy.

Z pohledu budoucího vývoje systému domácích úloh navrhuji především úpravu dříve implementovaných domácích úkolů do nového dvojjazyčného formátu. Mnou vytvořené úkoly sice tuto úpravu obsahují, využívají ale starší úkoly, které tuto úpravu nemají. Nový automat proto nedokáže vytvořit plně dvojjazyčné zadání nebo řešení.

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, 1996. ISBN 80-85828-21-9.

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

[4] Honzík, J. M.: Algoritmy, Studijní opora, 2006.

[5] Zbořil, F.: Základy umělé inteligence, Studijní opora, 2006.

[6] Wikipedia.cz, heslo Algoritmus, 26.04.2008. Dokument dostupný na URL http://cs.wikipedia.org/wiki/Algoritmus (květen 2008).

[7] Wikipedia.cz, heslo Asymptotická složitost, 09.05.2008. Dokument dostupný na URL http://cs.wikipedia.org/wiki/Asymptotick%C3%A1_slo%C5%BEitost (květen 2008).

[8] Wikipedia.cz, heslo Pascal (programovací jazyk), 04.04.2008. Dokument dostupný na URL http://cs.wikipedia.org/wiki/Pascal_%28programovac%C3%AD_jazyk%29 (květen 2008).

[9] Wikipedia.cz, heslo C (programovací jazyk), 10.04.2008. Dokument dostupný na URL http://cs.wikipedia.org/wiki/Jazyk_C (květen 2008).

[10] Programujte.com, Vývojové diagramy, 11.10.2005. Dokument dostupný na URL http://programujte.com/index.php?akce=clanek&cl=1970010171-vyvojove-diagramy-2-dil (květen 2008).

[11] Bayer, T.: Přednášky z předmětu Programování 1, Praha, 02.12.2007. Dokument dostupný na URL http://www.natur.cuni.cz/~bayertom/Prog1/programovani2.pdf (květen 2008).

[12] PC-Karel, 2004. Dokument dostupný na URL http://www.sweb.cz/pckarel/pckarel.html (květen 2008).

Seznam příloh

Příloha 1 – DVD s následujícím obsahem:

1. soubor `xtravn14_bp.pdf` – technická zpráva
2. složka `domaci_ulohy` – složka obsahující implementované domácí úlohy
3. složka `interpret` – složka obsahující modifikovaný interpet