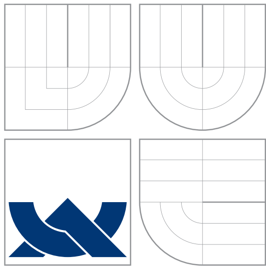
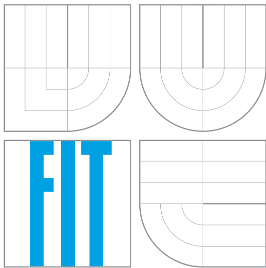


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

SYSTÉM PRO PODPORU VÝUKY DYNAMICKÝCH DATOVÝCH STRUKTUR

SYSTEM FOR SUPPORT OF DYNAMIC DATA STRUCTURES EDUCATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — JIŘÍ TRÁVNÍČEK
AUTHOR

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

BRNO 2007

{Toto je náhradní stránka. Zde bude vloženo oficiální zadání…}

{This is a placeholder page. The official assignment goes here…}

{Toto je náhradní stránka. Zde bude vložena licenční smlouva...}

{This is a placeholder page. The licence agreement goes here...}

Abstrakt

Hlavním cílem této práce je navrhnout a implementovat aplikaci, která může být využita jako pomůcka pro výuku základů programování. Konkrétně je pozornost soustředěna na oblast dynamických datových struktur. Cílová aplikace bude implementována s využitím webových technologií, takže může být provozována v běžném WWW prohlížeči. Nejdříve stručný úvod zrekapituluje datové struktury, které budou pokryty. Poté práce shrnuje vhodné technologie dostupné ve webových prohlížečích, se zaměřením na konkrétní technologii (kterou je DHTML), jež se stane cílovou platformou. Nejvýznamnější část této práce pojednává o návrhu konečné aplikace. Tato spíše teoretická část je poté následována popisem praktické implementace. Obsahem je také krátká uživatelská příručka.

Klíčová slova

datové struktury, dynamické datové struktury, zásobník, fronta, seznam, strom, binární strom, binární vyhledávací strom, WWW prohlížeč, DHTML, HTML, CSS, DOM, JavaScript

Abstract

The main objective of this work is to design and implement an application that can be used as an aid for the education of programming essentials. Particularly, the attention focuses on the domain of dynamic data structures. The target application will be implemented with the use of web technologies so that it can be run in an ordinary WWW browser. First of all, a brief introduction recapitulates the data structures to be covered. Then the work summarizes the usable technologies available within the web browsers with the focus on the particular technology (which is DHTML) that will become the target platform. The most significant part of this work then discusses the design of the final application. This rather theoretical part is then followed by the description of the practical implementation. A short user manual is also included.

Keywords

data structures, dynamic data structures, stack, queue, list, tree, binary tree, binary search tree, WWW browser, DHTML, HTML, CSS, DOM, JavaScript

Citace

Jiří TRÁVNÍČEK: Systém pro podporu výuky dynamických datových struktur, diplomová práce, Brno, FIT VUT v Brně, 2007

Systém pro podporu výuky dynamických datových struktur

Prohlášení

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

....
Jiří TRÁVNÍČEK
v Brně 2007-05-22

Poděkování

Děkuji všem pedagogům, vědeckým pracovníkům a dalším zaměstnancům VUT v Brně i všem ostatním lidem, kteří mi předali svoje znalosti, zkušenosti i nadšení pro obor, čímž významnou měrou přispěli k realizaci této práce.

Velmi děkuji svému vedoucímu diplomové práce Ing. Bohuslavu Křenovi, Ph.D. za trpělivou a odbornou pomoc poskytnutou při řešení problémů spojených s realizací této práce.

Zvláštní poděkování patří také mému vedoucímu v zaměstnání (FEKT VUT v Brně) Ing. Radkovi Pokornému za mimořádné pochopení v době realizace této práce.

© Jiří TRÁVNÍČEK, 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

Obsah **1**

1 Úvod **3**
1.1 Typografické konvence . . . 4
1.2 Názvoslovné konvence . . . 4
1.3 Struktura práce . . . 6

2 Dynamické datové struktury **7**
2.1 Zásobník . . . 7
2.2 Fronta . . . 8
2.3 Seznam . . . 9
2.4 Binární vyhledávací strom . . . 10

3 Technologie WWW prohlížečů **11**
3.1 Java . . . 12
3.2 Flash . . . 12
3.3 DHTML . . . 13
3.3.1 HTML . . . 13
3.3.2 CSS . . . 13
3.3.3 DOM . . . 13
3.3.4 JavaScript . . . 14

4 Návrh aplikace **15**
4.1 Volba podporovaných datových struktur . . . 15
4.2 Volba implementační platformy . . . 15
4.3 Analýza současného stavu vývoje . . . 16
4.4 Odstranění globálních proměnných . . . 18
4.5 Změny struktury uživatelského rozhraní . . . 18
4.6 Abstrakce odlišností datových struktur . . . 19
4.7 Fázované operace . . . 20
4.8 Podpora výukového režimu . . . 21
4.8.1 Změna struktury komunikace . . . 22
4.9 Implementace dalších datových struktur . . . 23

5 Popis implementace **24**
5.1 Skutečný rozsah implementace . . . 24
5.2 Podporované prohlížeče . . . 25
5.3 Uživatelské rozhraní . . . 25

5.4 Předávání odkazů na metody a uzávěry . . . 29
5.5 Implementace zkušebního (výukového) režimu . . . 30
5.6 Struktura zdrojového kódu . . . 31
5.6.1 Modul `index.html` . . . 31
5.6.2 Modul `js/main.js` . . . 31
5.6.3 Modul `js/dom_utils.js` . . . 31
5.6.4 Modul `js/misc.js` . . . 31
5.6.5 Modul `js/widgets.js` . . . 31
5.6.6 Modul `js/ds_sup.js` . . . 32
5.6.7 Modul `js/dummy_sup.js` . . . 32
5.6.8 Modul `js/tree_sup.js` . . . 33
5.6.9 Modul `js/tree.js` . . . 33
5.6.10 Modul `js/list_sup.js` . . . 33
5.6.11 Modul `js/list.js` . . . 33

6 Závěr **34**

A Uživatelská příručka **35**
A.1 Popis částí uživatelského rozhraní . . . 35
A.2 Vstupní pole 'Value' . . . 35
A.3 Stručný přehled ovládání . . . 36
A.4 Ovládání klávesnicí . . . 36

Literatura **37**

Kapitola 1

Úvod

Datové struktury patří k základním stavebním prvkům počítačových programů. Tuto myšlenku koneckonců vyjadřuje i název jedné z významných publikací [20] z této oblasti, jehož název v češtině zní *Algoritmy + datové struktury = programy*. (Tato publikace je k dispozici také v českém překladu, název její české verze je ale odlišný.)

Proč jsou však tolik důležité? Budeme-li zkoumat činnost jakéhokoliv programu, při adekvátní úrovni abstrakce zjistíme, že základním principem je získání vstupních dat, jejich transformace na data výstupní a jejich výstup nějakou vhodnou formou. Neprodukuje-li program žádný výstup, nemá žádný smysl. (I takovéto druhy „programů“ se občas podaří vytvořit, jak se lze dočíst v [5, s. 45], kde jsou nazývány černými dírami.)

Během těchto popsaných transformací nastává potřeba vhodného uložení dat. Kritérium vhodnosti je samozřejmě značně závislé na druhu aplikace, která určuje konkrétní požadavky. Obvykle nás zajímá např. prostorová efektivita uložení dat a také rychlost a způsob přístupu (náhodný nebo sekvenční) k uloženým datům. Datových struktur existuje více druhů, přičemž některé se liší výrazně, některé méně. Obecně ale platí, že každá struktura má odlišné vlastnosti vzhledem k různým kritériím, které na ně a na data v nich uložená můžeme klást.

Vhodným výběrem použitých datových struktur s ohledem na konkrétní použití tedy můžeme často velmi výrazně ovlivnit efektivitu výsledného programu. Nejen to, navíc platí, že kvalitní návrh datových struktur může zásadním způsobem zjednodušit příslušný algoritmus nad touto strukturou pracující. Toto samozřejmě platí i naopak a proto má smysl věnovat jejich výběru a návrhu významnou pozornost.

Jak je tedy vidět, dobrá znalost datových struktur nepochybně patří (společně se základy algoritmizace) ke zcela základním znalostem každého programátora a je jen přirozené, že tato problematika je náplní výuky základů programování. Dobrý programátor však musí mít nejen přehled o vlastnostech těchto struktur, ale měl by znát i jejich princip fungování. Ačkoliv jsou obvykle dostupné knihovny, které nejrůznější struktury implementují (např. *STL* v C++), může v některých případech nastat potřeba vlastní implementace, která se bez detailnějších znalostí neobejde. Tyto znalosti také navíc podporují pochopení rozdílných vlastností jednotlivých struktur, které vycházejí právě z jejich principů.

Výuka jakékoliv problematiky se ale často neobejde bez vhodných výukových pomůcek. Ale i kdyby jich nebylo nutně třeba, jejich využití může zvýšit efektivitu výukového procesu a zjednodušit i urychlit pochopení probírané látky.

Také pro výuku datových struktur lze vytvořit nejrůznější podpůrné nástroje. V rámci této práce právě jeden takový navrhneme a implementujeme. Půjde o aplikaci, která by měla uživateli usnadnit pochopení problematiky vybraných datových struktur (konkrétně z kate-

gorie dynamických struktur) a dále mu umožnit prověřit si svoje znalosti v praxi. Aplikace bude zaměřena na činnosti jednotlivých operací na těmito strukturami (jako např. vložení a zrušení prvku, vyhledávání apod.), jež přímo souvisí se základními principy jednotlivých struktur, které, jak již bylo naznačeno, jsou základem pro pochopení této problematiky.

Jistou zvláštností výsledné aplikace je také pro tento druh programu ne zcela obvyklá implementační platforma. Aplikace totiž pro svůj běh využívá prostředí WWW prohlížeče. Z tohoto hlediska lze říci, že práce také poslouží jako jistý prototyp ověřující praktické možnosti této vývojové platformy pro účely, pro které nebyla původně navržena. Kromě toho zároveň ověří praktickou použitelnost některých aplikovaných technologií, které se obvykleji používají pro vývoj WWW stránek.

Návaznost na předchozí práce Touto prací navazuji na svůj vlastní ročníkový projekt. Oproti zadání tohoto projektu došlo ke změně zadání a to změnou okruhu zpracovávaných datových struktur a dále rozšířením o novou funkčnost – výukový režim, který umožní uživateli ověření si získaných znalostí.

1.1 Typografické konvence

Tato práce byla vysázena typografickým systémem LaTeX. Kromě standardního písma tohoto systému budou použita následující písma pro odlišení některých částí textu:

názvy Vybrané názvy (programů, produktů, WWW stránek či jiných významných entit) jsou vysázeny *kurzívou.*

termíny Nově se vyskytnuvší termíny jsou vysázeny **tučným písmem**. Toto zvýraznění je obvykle použito pro první výskyt termínu v určité části textu, kde tento bývá zároveň definován nebo jinak vysvětlen.

termíny Ostatní termíny (další výskyty termínů z předchozí kategorie a termíny bez definice) jsou vysázeny *skloněným písmem*. Toto zvýraznění slouží k odlišení termínu od ostatního textu při menším zvýraznění v porovnání s předchozí kategorií.

`ident()` Identifikátory (např. názvy proměnných, funkcí, objektů apod.), jména souborů (včetně celých souborových cest) a případné citace textových výstupů a vstupů (jako třeba zadávané hodnoty) jsou vysázeny písmem s `pevnou šířkou`.

[OK] Názvy kláves, klávesových sekvencí a jednotlivých prvků uživatelského rozhraní (tlačítek, zaškrtávacích polí, přepínačů apod.) budou vysázeny v [rámečku].

Pro zápis řetězců (zejména obsahují-li mezery) bude používáno 'ohraničení apostrofy'. U textů, které původně obsahovaly jediný druh uvozovek (např. názvy) bude tato vlastnost zachována použitím "anglických uvozovek". Ve všech ostatních případech budou používány standardní „české uvozovky“.

1.2 Názvoslovné konvence

Výpočetní technika je obor, jehož „mateřštinou“ je angličtina. Proto také významná část pojmů této oblasti je anglických a jejich české překlady jsou občas problematické (např. neexistují, nejsou příliš zažité, mohou být nejednoznačné atd.). Z tohoto důvodu by mohlo

být vhodné přiklonit se v takových případech spíše k použití anglických termínů. Na druhé straně ne vždy je výsledkem tohoto postupu dobře čitelný český text. Proto se budeme snažit držet se českých překladů kdykoliv to bude možné s tím, že pro vyloučení pochybností bude (obvykle při prvním použití) v závorce za českým výrazem uveden jeho původní anglický ekvivalent.

Přesto v tomto místě uvedeme některé výrazy, o nichž je známo, že budou v dalším použity a zároveň lze očekávat, že by mohly být zdrojem problémů či pochybností.

Karty* a *záložky K problematickým termínům, zdá se, patří pojmy z oblasti grafického uživatelského rozhraní (GUI) používajícího tzv. *záložky* (*tabs*) – viz např. [11]. Termín *tab* se do češtiny překládá různými způsoby. Nejvhodnějším překladem se mi jeví *záložka*. Naneštěstí stejné slovo je také překladem anglického slova *bookmark*, které má se světě WWW prohlížečů historicky zažitější název. (Některé prohlížeče používají odlišný název *favorites*, které má poměrně bezproblémový překlad *oblíbené*.) Popsanému problému se elegantně vyhneme zavedením konvence založené na tom, že nebudeme o záložkách ve smyslu *bookmarks* vůbec hovořit a to i přes to, že se budeme v oblasti webových prohlížečů pohybovat. Termínem *záložka* tedy budeme rozumět prvek GUI a pokud by tomu bylo jinak, bude použito upřesnění např. uvedením anglického ekvivalentu, jak bylo popsáno.

V této souvislosti je třeba ještě jedno zpřesnění. Záložky lze rozdělit na 2 základní části. První z nich jsou „ouška", která jsou vždy vidět u všech záložek a slouží k přepínání mezi nimi. Zbývají části záložek, ze kterých je vidět pouze ta, která patří k aktivní záložce. Často se tyto části nerozlišují a obě se nazývají záložkami. Pro odlišení si proto zavedeme dva nezávislé termíny. Pro zmíněná „ouška" nechť je i nadále používán termín **záložka** (**tab**). Zbývající část budeme nazývat **karta** (**tab pane**), což je také občas používaný termín. Podobně je tomu i s anglickými termíny, kde *tab pane* je také občas používaný termín, ale s *tab* se lze setkat častěji a používá se také univerzálně pro obě části. Občas bude také řeč o **pruhu záložek** (**tab bar**). Od anglických termínů jsou odvozeny názvy souvisejících identifikátorů ve zdrojových kódech.

Pseudonázev *Mozilla* Pro stručnost a zobecnění budeme používat pseudonázev prohlížeče **Mozilla**. Tento název má delší historii, jeho nejznámější použití z poslední doby bylo jako název prohlížeče od *Mozilla Foundation*, který byl vyvíjen v rámci projektu *SeaMonkey*. Po změnách na přelomu roků 2005/2006 tento projekt pokračuje pod svým kódovým jménem a původní název již není jménem konkrétního produktu. Protože *SeaMonkey* používá stejné technologie jako jiné prohlížeče od *Mozilla Foundation* (a tudíž má příslušná skupina prohlížečů potenciálně shodné vlastnosti), budeme uvedeným pseudonázvem obvykle rozumět libovolný z nich.

Zkratky Pro zvýšení přehlednosti a zestručnění bude pro některé časté termíny použito následujících zkratek:

- DS datová struktura
- BVS binární vyhledávací strom
- MSIE Microsoft Internet Explorer
- GUI grafické uživatelské rozhraní

Notace souborových cest Souborové cesty týkající se jednotlivých souborů aplikace budou zapisovány tak, jak je obvyklé v prohlížečích při zadávání URL. Při popisu architektury aplikace budou používány výhradně relativní cesty, neboť absolutní cesty zahrnují konkrétní instalační adresář aplikace, který není důležitý (může být libovolný). Relativní cesty budou vztaženy k bázovému adresáři aplikace (tj. adresáři nejvyšší úrovně, ve kterém se nacházejí soubory aplikace).

Zápis identifikátorů Identifikátory ze zdrojových kódů budou uváděny s přihlédnutím k obvyklé syntaxi značkovacího nebo programovacího jazyka, ke kterému se vztahují.

1.3 Struktura práce

Práci uvozuje teoretická část, kterou zahájíme rekapitulací oblasti datových struktur v kapitole 2. Zde také budeme blíže specifikovat zpracovávanou oblast a některé základní pojmy. Zejména bude uveden přehled a stručný popis konkrétních struktur, které mají být podporovány výslednou aplikací.

Následně splníme další prerekvizitu, a tou je zmapování technologií, které lze pro implementaci aplikace použít, tedy technologií poskytovanými webovými prohlížeči. Toto je náplní kapitoly 3, kde uvedeme stručný přehled nejvýznamnějších technologií a blíže popíšeme tu, která se stane implementační platformou.

Kapitolou 4 zakončíme čistě teoretickou část a zároveň se již přiblížíme části praktické. Obsahem této kapitoly je návrh aplikace. Protože implementace vychází z již hotového kódu (viz 1 na str. 4), začneme nejdřív zhodnocením stávajícího stavu a určením potřebných změn v kódu, které vycházejí z požadavků nového zadání. Poté provedeme návrhová rozhodnutí vedoucí k vytvoření kódu, který bude podporovat požadovanou funkčnost.

Praktickou realizaci popisuje kapitola 5. Obsahem je zejména popis konkrétní implementace na základě návrhu z předchozí kapitoly. Budou zde již uváděny některé konkrétní detaily. Dále bude popsán skutečný rozsah implementace a popsány případné problémy a jejich řešení.

Závěr práce nalezneme v kapitole 6. Jejím obsahem je stručné zhodnocení celé práce. Zmíníme i možnosti dalšího vývoje.

Stručná uživatelská příručka, která popisuje ovládání výsledné aplikace, je součástí přílohy A.

Součástí této práce je také datové médium (CD-R). Jeho detailní obsah není součástí tohoto dokumentu; za účelnější považuji tuto informaci umístit přímo na uvedené médium. Proto detailní informace najde čtenář v kořenovém adresáři tohoto média v souboru `README` (znaková sada ISO 8859-2, řádky zakončeny `LF`) nebo `README.txt` (znaková sada Windows 1250, řádky zakončeny `CR`+`LF`. Pouze jen stručně uveďme, že zde čtenář najde výslednou aplikaci (kterou lze ihned spustit) i tento dokument (a to ve formátech *PostScript* a *PDF*) včetně všech zdrojových kódů.

Kapitola 2

Dynamické datové struktury

Protože budeme vyvíjet aplikaci pro výuku dynamických datových struktur, budeme také při implementaci potřebovat alespoň jistou znalost této problematiky. Začněme tedy výklad touto oblastí. Následující text si neklade za cíl být v tomto směru kompletní ani vyčerpávající. Budeme se snažit pouze o lehkou rekapitulaci omezenou víceméně jen na struktury, které výsledná aplikace bude podporovat. Čtenářům s hlubším zájmem o tuto problematiku je k dispozici celá řada publikací, např. již dříve zmíněná publikace [20]. Tematiku zpracovávají také [4] a [3] (posledně jmenovaný zdroj je dostupný pouze v rámci Fakulty informačních technologií VUT), které byly významnou měrou použity při realizaci této práce.

Začněme zařazením uvažované podmnožiny množiny datových struktur. DS lze rozdělit podle různých kritérií. Jedním z možných rozdělení je dělení na *statické DS* a *dynamické DS*. **Statické datové struktury** jsou takové DS, jejichž velikost ani vnitřní uspořádání se za běhu nemění [4, s. 10], [10, část 6.1]. Tyto struktury jsou alokovány při překladu a existují tedy v paměti (tj. zabírají přidělené místo) po celou dobu běhu programu. Pro **dynamické datové struktury** platí analogicky opak, tedy že mohou měnit svoji velikost nebo vnitřní uspořádání. Navíc vznikají (tedy jsou alokovány v paměti) až za běhu programu a také mohou za běhu programu zanikat. Pro praktické použití mají samozřejmě smysl obě kategorie, nicméně předmětem našeho dalšího zájmu budou pouze dynamické DS.

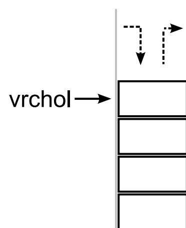
DS lze také klasifikovat podle vnitřního uspořádání jejich prvků. Podle této klasifikace rozlišujeme *lineární DS* a *nelineární DS*. Charakteristickou vlastností **lineárních datových struktur** je skutečnost, že každý prvek této DS má právě jednoho předchůdce a jednoho následníka [3, s. 43]. Pokud tato vlastnost neplatí, jde o **nelineární datovou strukturu**.

DS můžeme třídit i podle dalších kritérií, např. podle způsobu přístupu k jejich prvkům, podle homogenity apod. Pro zasazení zpracovávané oblasti však již nejsou významná a proto se jim zde nebudeme dále věnovat.

2.1 Zásobník

Zásobník je představitelem *lineárních* DS. Protože u něj dochází ke změně počtu prvků, patří mezi *dynamické* DS. Někdy je také označován jako struktura **LIFO** (last in, first out), což znamená, že prvky do něj uložené z něj lze zpět získat v opačném pořadí, než v jakém byly vloženy. Příklad zobrazení zásobníku je na obr. 2.1.

Zásobník (stack) je charakteristický tím, že umožňuje v jednom okamžiku přistupovat pouze k jednomu prvku, který se nachází na tzv. **vrcholu zásobníku** (top). Prvek, který



Obrázek 2.1: Zásobník

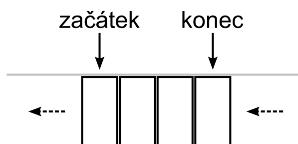
se nachází na vrcholu, je prvkem, který byl do zásobníku naposledy uložen. Jinam nežli na vrchol nelze prvky ukládat. Stejně tak pouze z vrcholu lze prvky odebírat. Odebráním dojde ke zpřístupnění prvku, který byl uložen před prvkem právě odebraným. Operace uložení na zásobník se nejčastěji nazývá **push** a operace odebrání **pop**. V některých implementacích *pop* zároveň vrací odebranou hodnotu (toto chování je běžné např. u strojové instrukce `pop`). Při požadavcích striktnější sémantiky se pro získání hodnoty na vrcholu zavádí zvláštní operace, která bývá nazývána **top**, a *pop* pak nevrací žádnou hodnotu. V tomto případě může být kombinovaná operace pojmenována např. **top-pop** ([3, s. 60] pro tuto operaci zavádí identifikátor `TopPop`). Kromě těchto základních operací DS implementuje obvykle další pomocné operace (test na prázdnost, inicializace apod.).

Zásobník lze implementovat nad jednosměrně vázaným seznamem. Pro implementaci lze také použít statickou strukturu (např. pole), výsledná struktura je ale potom pouze pseudodynamická, neboť maximální kardinalita (velikost) výsledné DS je v tomto případě omezena velikostí hostitelské struktury (v předchozím případě je omezujícím faktorem pouze velikost volné paměti).

Jedná se o významnou strukturu. Na principu zásobníku je např. založena řada hardwarových architektur včetně snad nejrozšířenější architektury *x86*. Zásobník se zde používá přímo na úrovni strojového kódu a to pro ukládání návratových adres a hodnot registrů. Do tohoto zásobníku se také např. ukládají lokální proměnné funkcí předávají se v něm parametry při volání těchto funkcí. Zásobník má také mnohé další aplikace na vyšších programových úrovních, např. vyhodnocování výrazů, syntaktická analýza, reverzace pořadí prvků (přímé využití vlastnosti *LIFO*) aj.

2.2 Fronta

Podobně jako zásobník je také fronta (queue) *lineární* i *dynamickou* DS. Na rozdíl od zásobníku je ale označována jako **FIFO** (first in, first out), čímž je vyjádřeno, že prvky z ní lze získat ve stejném pořadí, v jakém byly vloženy. Možné vyobrazení fronty je na obr. 2.2.



Obrázek 2.2: Fronta

Oproti zásobníku má fronta 2 významné „body“ – **začátek** (front) a **konec** (back). Přistupovat lze (podobně jako u zásobníku) pouze k prvkům v těchto dvou bodech, přičemž na konec se prvky pouze vkládají, zatímco ze začátku jsou výhradně odebírány. Názvy

operací u fronty se jeví více různorodé nežli tomu je u zásobníku. Operace pro vložení prvku do fronty může být nazvána např. **enqueue**, operace pro odebrání prvku zase **dequeue** [16]. Analogicky jako u zásobníku slouží operace **front** pro čtení hodnoty na začátku a další pomocné operace. [4, s. 52] a [3, s. 65] používají jiné názvy – reprezentované identifikátory `QueUp`, `Remove` a `Front`.

Také pro tuto strukturu lze použít seznam jako implementační strukturu. Tentokrát si však nevystačíme s obyčejným jednosměrně vázaným seznamem, ale bude třeba použít dvojsměrně vázaný seznam nebo jednosměrně vázaný seznam rozšířený o podporu ukazatele na poslední prvek. Stejně jako u zásobníku i zde je možná pseudodynamická alternativa s využitím pole.

I když fronta není (na rozdíl od zásobníku) typicky podporována přímo na úrovni strojového kódu, jde také o významnou DS. V hardware se využívá např. pro realizaci vyrovnávacích pamětí. Mimo tuto oblast lze uvést např. využití pro realizaci front zpráv či událostí v operačních systémech nebo různých systémech GUI.

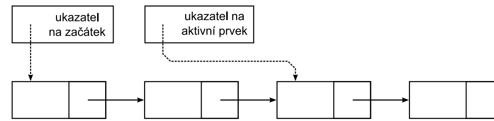
2.3 Seznam

Seznam (list) náleží stejně jako předchozí struktury do skupiny *dynamických lineárních* DS.

Je třeba poznamenat, že označení *seznam* je poněkud obecné. Lze definovat několik druhů seznamů. Nejjednodušším typem je jednosměrně vázaný (nebo krátce jednosměrný) seznam (singly-linked list). Jeho rozšířením pak vznikne dvojsměrně vázaný (či dvojsměrný) seznam (doubly-linked list). V [15] jsou tyto dva druhy společně kategorizovány jako lineárně vázané seznamy (linearly-linked lists). Kromě této skupiny seznamů totiž existují ještě kruhově vázané (zkráceně kruhové) seznamy (circularly-linked lists). Také tyto lze rozdělit na jednosměrné a dvojsměrné a i přes svůj název jsou podle výše uvedené definice lineárními dynamickými DS. Protože pro implementaci uvažujeme pouze jednosměrné seznamy, nebudeme se dalším variantám dále podrobněji věnovat. Budeme-li nadále hovořit o seznamech, budeme mít na mysli jejich jednosměrnou nekruhovou variantu, nebude-li uvedeno jinak.

Jednosměrně vázaný seznam je struktura, jejíž charakteristickou vlastností je možnost procházet jen jedním směrem. Každý prvek kromě své hodnoty uchovává navíc odkaz (ukazatel) na svého následníka. V rámci seznamu je uchováván také odkaz na první prvek, ke kterému je jako jedinému prvku seznamu možný přímý přístup (obdobně jako tomu bylo u zásobníku a fronty). K dalším prvkům lze přistupovat sekvenčně ve směru průchodu seznamem (což již naopak zase u zmíněných struktur možné nebylo). Konkrétní způsob zajištění přístupu k dalším prvkům záleží na implementaci. Např. [4, s. 46] používá variantu seznamu s ukazatelem na tzv. aktivní prvek, který slouží k tomuto účelu. Zároveň definuje pojem **aktivní seznam**, což je seznam, ve kterém je v daném okamžiku nějaký prvek aktivní. V opačném případě se jedná o **neaktivní seznam**. Jinou možností je přímá práce s ukazateli na jednotlivé prvky [15], nicméně tento postup porušuje zapouzdření a abstraktnost takové struktury. Při jednoduchých implementacích však toto řešení může být vyhovující. Řešením může být použití speciálních přístupových mechanismů, jako jsou např. iterátory (to je případ *STL* v C++), které jsou obdobné ukazatelům, ale umožňují zapouzdření dodržet. Na obr. 2.3 je uveden příklad zobrazení jednosměrného seznamu včetně ukazatele na aktivní prvek.

Podívejme se na typické operace nad seznamem. Je nutné poznamenat, že jména operací i související názvosloví se mohou mezi jednotlivými autory lišit (viz např. [10, část 6.5] a [15]). V následujícím přehledu vycházíme z [4, s. 47] a [3, s. 44]. Vložení prvku na začátek

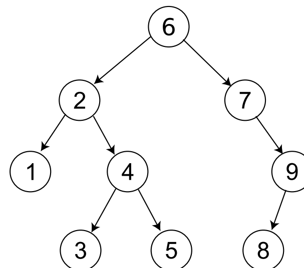


Obrázek 2.3: Jednosměrně vázaný seznam

seznamu provádí operace `InsertFirst`, jeho rušení `DeleteFirst`. Pro vložení prvku za aktivní prvek slouží `PostInsert` a rušení následníka aktivního prvku provádí `PostDelete`. Ukazatel na aktivní prvek je ovládán operacemi `First` (aktivace prvního prvku) a `Succ` (aktivace následníka). Operaci pro změnu hodnoty aktivního prvku autor nazval `Actualize`. Kromě uvedených jsou zde predikáty neprázdnosti a aktivity seznamu, operace pro získání hodnoty prvního a aktivního prvku a operace inicializace DS.

2.4 Binární vyhledávací strom

Binární vyhledávací strom je speciálním případem binárního stromu, u kterého navíc platí pravidla pro hodnoty jednotlivých uzlů: Pro každý uzel platí, že levý podstrom je buď prázdný nebo obsahuje uzly, jejichž hodnota je menší než hodnota tohoto uzlu, a analogicky pravý podstrom je buď prázdný nebo obsahuje uzly, jejichž hodnota je vyšší než hodnota tohoto uzlu [4, s. 126]. Pro úplnost ještě připomeňme, že binární stromy jsou stromy s aritou 2 a že patří mezi tzv. kořenové stromy a ty jsou acyklickými orientovanými grafy [4, s. 61]. Binární vyhledávací strom patří mezi *nelineární dynamické* DS. Typický způsob jeho znázornění je na obr. 2.4 (obrázek byl převzat z [19] a změnou hodnot uzlů upraven na BVS).



Obrázek 2.4: Binární vyhledávací strom

Typickými operacemi BVS jsou vložení a zrušení uzlu, vyhledávání zadaného prvku a různé průchody stromem. Kromě těchto operací lze odvodit i jiné pomocné (predikáty prázdnosti stromu či podstromů, ...) či komplexní operace (rušení či kopie celého stromu apod.).

Protože základní podpora BVS byla již implementována v rámci ročníkového projektu (kde byla tato problematika také blíže popsána), nebudeme zde zbytečně další popis opakovat. Pro úplnost dodejme, že za referenční zdroje informací byly v tomto směru považovány [4, s. 61 a 126] a dále v době implementace dostupné (tedy aktuální verze a verze dochované z doby mého studia tohoto předmětu) slidy přednášek z předmětu Algoritmy a datové struktury (ADS, IAL), které kromě již uvedené publikace pokrývá [3, s. 78 a 107]. Při dalším rozšíření dosavadní implementace této DS budou použity uvedené materiály.

Kapitola 3

Technologie WWW prohlížečů

Jak již víme, výsledná aplikace bude provozována ve webovém prohlížeči. Proto v této kapitole uvedeme krátkou rekapitulaci souvisejících technologií. Ani zde se nebudeme snažit o komplexní přehled; z praktického hlediska má smysl zabývat se zejména technologiemi, které jsou dostatečně rozšířené a tudíž běžně k dispozici. Volbě konkrétní technologie se budeme věnovat v kapitole 4.

Před tím, než se dostaneme k výčtu konkrétních technologií, je vhodné uvést, že technologie dostupné v prostředí WWW prohlížečů lze rozdělit na 2 základní kategorie. První kategorii tvoří technologie **aktivní na straně klienta** (client-side) a do druhé patří technologie **aktivní na straně serveru** (server-side).

Použití server-side technologií automaticky předpokládá, že aplikace bude uložena na WWW serveru, ze kterého bude příslušný obsah dodáván klientům. Na tomto serveru také poběží kód aplikace, který danou technologii využívá (ve skutečnosti může kód běžet i na jiném serveru či clusteru serverů, ale tento detail je pro klienta a uživatele dané aplikace transparentní).

Client-side technologie pro svoji činnost server nevyžadují, na druhé straně podmínkou jejich využití je podpora dotyčné technologie v prohlížeči, neboť příslušný kód běží právě v klientském prohlížeči. Volbu takovéto technologie je tedy nutné provádět obezřetně s ohledem na stupeň podpory a rozšíření v prohlížečích uživatelů, u kterých lze použití aplikace předpokládat.

Poznamenejme, že pro implementaci se budeme orientovat na technologie, které běží na straně klienta a proto se serverovými technologiemi nebudeme dále zabývat. Pro úplnost pouze uveďme, že do této kategorie patří rozšířené *PHP* a *ASP* či *ASP.NET*. Často se setkáme také s CGI skripty psanými v *Perlu*, *PHP* a jiných jazycích. Do této kategorie také patří *JavaScript* a *Java*, které, jak za chvíli uvidíme, patří také k technologiím běžícím na straně klienta. Popis dalších technologií s odkazy na bližší informace lze nalézt např. v [18].

Pro upřesnění je také třeba uvést, že v této kapitole uvedené informace o stavu podpory jednotlivých technologií v prohlížečích vycházejí z verzí prohlížečů dostupných v době, kdy byl započat vývoj aplikace a ověřován tento stav. Jedná se tedy konkrétně o tyto verze prohlížečů: *Mozilla* verze 1.7.6 (v této době ještě byl název *Mozilla* názvem skutečného produktu, viz 1.2 na str. 5) a *Internet Explorer* verze 6.0 SP1. Tyto informace se týkají také v této době aktuální verze prohlížeče *Firefox* a pravděpodobně i dalších z rodiny *Mozilla*.

V současnosti (tj. o 2 roky později) lze předpokládat, že je situace o něco lepší nebo přinejmenším obdobná. Protože původním smyslem těchto informací bylo potvrzení použitelnosti daných technologií, není zcela nutné provést opakované zmapování nejaktuálnějšího

stavu. U příslušných technologií nemáme důvod předpokládat ukončení jejich podpory.

3.1 Java

Java je moderním objektovým programovacím jazykem. Její významnou výhodou je přenositelnost kódu a to i na binární úrovni. Nejen tato výhoda přispívá k výrazné oblibě Javy v poslední době.

Java jako taková není na prohlížeči nijak závislá. Programy v ní napsané mohou běžet zcela nezávisle na prohlížeči, k jejich provozu je potřeba pouze instalace základního prostředí, které se nazývá *Java Runtime Environment* (JRE). Použití Javy přímo ve webovém prohlížeči kromě JRE vyžaduje instalaci příslušného plug-in modulu do tohoto prohlížeče. Modul je k dispozici pro většinu rozšířených prohlížečů. Javová aplikace postavená na této technologii se nazývá **applet**.

Z programátorského hlediska je Java čistě objektový programovací jazyk se silnou typovou kontrolou. Jde o kompilovaný jazyk, ve kterém se zdrojový kód překládá do tzv. bytekódu (bytecode). Výsledný bytekód je binárním tvarem javových programů a je interpretován tzv. **javovským virtuálním strojem** (Java virtual machine). Portabilita Javy je dána dostupností virtuálního stroje pro jednotlivé cílové platformy. Virtuální stroj poskytuje javovým aplikacím unifikované prostředí nezávislé na konkrétní platformě.

Java je tedy interpretovaným jazykem a to i přesto, že se kompiluje. Díky kompilaci do bytekódu se ale nevýhoda pomalé interpretace částečně eliminuje, protože interpretace bytekódu je výrazně efektivnější než interpretace zdrojového kódu.

S Java applety se lze relativně běžně setkat na některých WWW stránkách. Použitelnost této technologie pro vývoj naší aplikace dokazuje existence aplikací podobných té, kterou máme implementovat:

- *Red/Black Tree Demonstration*, URL http://www.ececs.uc.edu/~franco/C321/html/RedBlack/redblack.html
- *Tree Animation Tool*, URL http://www.engin.umd.umich.edu/CIS/course.des/cis350/treetool/index.html

3.2 Flash

Flash je zejména v poslední době populární technologií. Dá se dokonce říci, že se používá častěji než Java. S Flashem se lze setkat zejména na komerčních WWW stránkách, kde je využíván pro tvorbu nejrůznějších grafických efektů případně jsou v něm implementovány celé stránky. Poslední dobou lze také pozorovat stále častější využití ke zobrazování reklamních proužků (bannerů), kde nahrazuje dosud používané animované GIF obrázky.

Podobně jako Java je Flash řešen jako plug-in modul pro prohlížeč. Jistou výhodou Flashe oproti Javě je jeho výrazně menší prostorová náročnost. Také Flash podporuje většinu rozšířených prohlížečů, ačkoliv předchozí verze 8 nebyla např. dostupná pro Linux. Nová verze 9 tento problém již odstranila.

Na základě zkušeností s existujícími aplikacemi Flashe lze předpokládat, že i tuto technologii lze pro realizaci aplikace využít.

3.3 DHTML

DHTML je zkratka s významem **dynamické HTML** (dynamic HTML). Nejde přímo o konkrétní technologii, DHTML označuje soubor několika technologií. Tento soubor je tvořen [14] statickým značkovacím jazykem (např. *HTML*), skriptovacím jazykem (např. *JavaScript*), jazykem pro definici prezentace (např. *CSS*) a objektovým modelem dokumentu (*DOM*). Příklady uvedené v závorkách jsou typicky používanými zástupci jednotlivých kategorií [8].

3.3.1 HTML

HTML je hypertextový značkovací jazyk (hypertext markup language). Jde o jednu z nejstarších technologií v prohlížečích. Samotné HTML umožňuje tvorbu pouze statických dokumentů. Jde o tzv. hypertextové dokumenty, což jsou dokumenty obsahující odkazy na jiné dokumenty. Aktivací těchto odkazů (např. kliknutím myší) lze přejít na odkazovaný dokument.

HTML prošlo postupným vývojem. Aktuální verze 4.01 podporuje kromě textu a odkazů vložit do stránek další prvky. Lze vkládat obrázky i různé ovládací prvky, které se používají pro tvorbu formulářů, jako třeba tlačítka, přepínače, zaškrtávací pole apod. Tyto prvky lze využít také pro realizaci uživatelského rozhraní aplikace.

HTML verze 4.01 definuje standard [26] W3C (World Wide Web Consortium). Vzhledem k tomu, že tento standard již existuje několik let, dá se očekávat dobrá podpora ve většině prohlížečů. Mozilla i MSIE poslední verzi HTML bez zásadních problémů podporují.

3.3.2 CSS

CSS neboli kaskádové styly (cascading stylesheets) jsou prostředkem, který slouží k definici vzhledu HTML dokumentů. V posledních verzích HTML je to preferovaná metoda, která nahrazuje prezentační značky a atributy dřívějších verzí HTML. Ty sice existují i v tzv. přechodných (transitional) variantách aktuální verze HTML, nicméně jsou nedoporučovány (deprecated). Ve striktních (strict) variantách nejsou povoleny a CSS je jediným prostředkem.

Výhodou CSS jsou širší možnosti nastavení vzhledu v porovnání s možnostmi samotného HTML. Kromě toho CSS zavádí pojem tříd (class), které odpovídají stylům, se kterými se lze setkat v moderních textových procesorech a DTP programech.

Podobně jako HTML, i CSS prošlo vývojem, i když jeho historie je kratší. Po první verzi CSS level 1 následovalo CSS level 2, které možnosti předchozí verze zásadním způsobem rozšířilo. Pro realizaci aplikace je nejdůležitější a nutnou novinkou podpora polohování jednotlivých elementů dokumentu. Aktuální verzí je revize 1 této verze (CSS level 2 revision 1), zkráceně CSS 2.1.

CSS 2.1 je rovněž definováno standardem [21] W3C. Aktuální verze CSS je sice relativně nová, ale oproti CSS level 2, které již nějaký čas existuje, nepřináší zásadní změny. I zde tedy lze očekávat přijatelnou úroveň podpory v existujících prohlížečích. Mozilla i MSIE poslední verzi CSS podporují přijatelným způsobem.

3.3.3 DOM

DOM označuje objektový model dokumentu (document object model). Představuje objektovou reprezentací hierarchické struktury dokumentu. Definuje rozhraní objektů reprezen-

tujících strukturu dokumentu.

Obdobně jako předchozí technologie se také DOM vyvíjí. První verze DOM Level 1 obsahuje základní podporu jednotlivých elementů HTML. Následující verze DOM Level 2 je již rozdělena do několika částí. Oproti DOM Level 1 pokrývá např. styly CSS, zpracování událostí apod. Poslední verze DOM Level 3 přidává další vlastnosti.

Objektová rozhraní, která DOM definuje, lze využívat skriptovacími jazyky implementovanými v prohlížeči. Jednotlivým elementům HTML odpovídají objekty v DOM. Podpora stylů v DOM umožňuje obdobně pracovat s CSS styly.

Taktéž DOM Level 2 je definován standardem W3C. Jak již bylo řečeno, standard se skládá z několika částí: *Core* [22], *Views*, *Events* [23], *Style* [25], *Traversal and Range* a *HTML* [24]. Také tyto standardy jsou již nějaký čas k dispozici, proto lze předpokládat, že by mohly být ve stávajících prohlížečích podporovány na přijatelné úrovni. Mozilla i MSIE podporují dostatečně částí *Core*, *Style* a *HTML*. Hůře je na tom podpora *Views*, *Events* a *Traversal and Range*, ale bez těchto částí se lze obejít (nejzásadnější funkčnost části *Events* je možné implementovat prostředky DOM Level 0).

Kromě uvedených verzí standardu se lze setkat s označením DOM Level 0. Jde o soubor funkcionality společné verzím 3 prohlížečů Netscape a MSIE (viz *Glossary* v [24]. DOM Level 0 není oficiálně standardizován. Také aktuální verze Mozilly i MSIE některé z těchto funkcí podporují.

3.3.4 JavaScript

HTML i CSS poskytují základní kameny k vytvoření jakéhokoliv obsahu. Pokud však chceme vytvořit aplikaci a ne pouze statickou stránku, potřebujeme nějakým způsobem popsat dynamické chování. Tuto možnost nám dává skriptovací jazyk. DOM je pro skriptovací jazyk mostem pro přístup k obsahu dokumentu vytvořeného pomocí HTML a CSS.

JavaScript je objektový skriptovací jazyk. Na rozdíl od Javy není striktně objektový. Nemá také silnou typovou kontrolu; je to dynamicky typovaný jazyk, což znamená, že proměnná může obsahovat hodnotu libovolného typu a tento typ se může při běhu programu libovolně měnit přiřazením hodnoty odlišného typu. Jedná se o interpretovaný jazyk.

JavaScript byl v průběhu svého vývoje vydán v několika verzích a byl také standardizován organizací *Ecma International* pod názvem *ECMAScript*. JavaScript je nyní implementací ECMAScriptu od *Mozilla Foundation*. *Microsoft* má svoji vlastní implementaci nazvanou *JScript*. Obě tyto implementace obsahují některá rozšíření oproti standardu, která umožňují spolupráci s prohlížečem. Těmto rozšířením nebude možné se vyhnout, nicméně mezi prohlížeči by měla existovat společná podmnožina základních funkcí (z nichž některé patří do DOM Level 0), s níž bychom měli vystačit.

Dle [6] (kapitola *JavaScript Overview*, část *JavaScript and the ECMAScript Specification*) JavaScript ve verzi 1.5 odpovídá aktuální verzi ECMAScriptu dle 3. edice standardu ECMA-262 [1]. Nejnovější verzí JavaScriptu je verze 1.7. Protože se však jedná o poměrně novou verzi, bude vhodné omezit se na vlastnosti podporované již staršími verzemi. Kromě zmíněného standardu [1] tento jazyk také dobře popisují [6] a [7], které jsou méně formálního charakteru, ale zato dokumentují podporu jednotlivých vlastností v různých verzích JavaScriptu. Mimo to popisují prvky, které nepokrývá standard ECMA-262 a které jsou specifikem JavaScriptu.

Mozilla podporuje aktuální verzi JavaScriptu. MSIE podle výsledků testů podporuje JavaScript verze 1.3, lze však najít i vlastnosti novějších verzí, např. podporu výjimek.

Kapitola 4

Návrh aplikace

Po rekapitulaci teoretických základů, na kterých práce staví, můžeme již přistoupit k návrhu aplikace. Ze všeho nejdříve zvolíme podporované DS a popíšeme volbu technologie, pomocí které bude aplikace implementována. Poté budeme pokračovat analýzou stávajícího stavu, kde popíšeme současný stav vývoje a určíme změny potřebné pro jeho další pokračování. Dále budeme postupně procházet jednotlivými kroky návrhu, které doprovodíme souvisejícími informacemi. Při tomto se budeme věnovat návrhovým rozhodnutím zásadnějšího koncepčního charakteru.

4.1 Volba podporovaných datových struktur

Volba podporovaných DS vychází z doporučené množiny, která je součástí zadání: seznam, zásobník a fronta. Protože je k dispozici také hotová implementace binárního vyhledávacího stromu a protože i tato struktura patří k základním (a má tedy smysl ji v aplikaci tohoto druhu podporovat) a navíc do zpracovávané oblasti dynamických DS náleží, doplníme náš seznam o tento BVS. Pokryjeme tedy právě struktury popsané v kapitole 2.

4.2 Volba implementační platformy

Jak jsme v kapitole 3 předeslali, budeme se orientovat na client-side technologie. Jednoduchým vysvětlením této volby je skutečnost, že takto zněl požadavek zadání ročníkového projektu. A pro další vývoj pak nenastal důvod k orientaci na jinou kategorii technologií.

Pomineme-li tyto důvody, jsem toho názoru, že pro aplikaci tohoto typu je zvolená kategorie technologií vhodnější. Pokud bychom chtěli použít čistě serverové technologie, dostáváme se do stavu, kdy veškerá interaktivita aplikace je realizována generováním změněného obsahu na serveru, což znamená, že pro jakoukoliv změnu musí prohlížeč stáhnout novou stránku. Pomineme-li zbytečnou režii tohoto řešení, je zde hlavní problém, kterým je zhoršená plynulost celé aplikace, která je způsobena jednak latencí výsledného systému klient-server a také už jen tím, že pro každou změnu musí prohlížeč načíst a vykreslit nový obsah, což také nějaký čas trvá a působí rušivě. Podobná řešení se běžně používají, ale ne pro aplikace tohoto charakteru. Výhodou jsou minimální požadavky na prohlížeč, u kterého může stačit pouze podpora statického obsahu.

Existují i technologie, které odstraňují nutnost načítání celých stránek, např. AJAX [12], zde se však již jedná o kombinaci klientské a serverové technologie. Tím už ale ztrácíme výhodu předchozí koncepce. Protože takové řešení již vyžaduje od prohlížeče podobnou míru

funkčnosti jako DHTML, mělo by takové řešení smysl, pokud bychom realizovali funkčnost, která by z přítomnosti serveru mohla nějak těžit. Žádná taková funkčnost však v tuto chvíli plánována není a proto v podobném řešení nespatřuji zásadní význam.

Klientské technologie se tedy jeví jako nejvhodnější pro implementaci naší aplikace. Zbývá tedy zvolit konkrétní technologii z této skupiny. Zvolil jsem využití DHTML a to z důvodu zkušeností s jednotlivými technologiemi (kromě DOM). Výhodou jsou i minimální nároky na vývojářské prostředí – není třeba žádný překladač ani jiné specializované nástroje. Vhodný je však JavaScript debugger a DOM inspektor, což jsou nástroje, které Mozilla poskytuje v základní výbavě (alespoň *SeaMonkey*; *Firefox* možná vyžaduje dodatečnou instalaci rozšíření, které je ale v takovém případě k dispozici).

Poznamenejme, že tato volba již byla učiněna při prvním zahájení vývoje v rámci dosavadní práce. Pro další vývoj se této technologie budeme držet, protože v rámci předchozího vývoje se ukázala použitelnou i přes některé drobné nedostatky existujících implementací ve webových prohlížečích. Díky tomu také může další vývoj vyjít z dosud napsaného kódu, což ostatně také bývá obvykle ekonomičtější nežli začít zcela nový vývoj.

DHTML nám poskytuje skriptovací jazyk, ve kterém bude možné realizovat jak základní řídící logiku celé aplikace včetně GUI tak i potřebné algoritmy nad jednotlivými DS. Vizualizačním prostředkem nám budou elementy HTML, řídit jejich vzhled a polohu nám umožní CSS, přičemž tyto 2 technologie propojuje se skriptovacím jazykem právě DOM.

Popsané řešení vizualizační stránky však není zcela přímočaré. Pomocí HTML elementů jsme schopni realizovat pouze tvary, které odpovídají obdélníkům. Při jednotkovém (obecně malém) jednom z rozměrů také dokážeme vytvořit svislé a vodorovné čáry, ale tím naše možnosti končí. Nedokážeme vytvořit např. šikmé čáry ani jiné tvary, např. křivkové. To je viditelným omezením např. pro vizualizaci stromů, u kterých je běžné zobrazovat uzly kulaté (viz obr. 2.4). Řešení by mohly nabídnout (rastrové) obrázky (což je běžná technika při tvorbě WWW stránek), ale toto řešení má svoje omezení. Realizace šikmých čar je ještě složitější. V tomto směru bychom si mohli vypomoci vektorovými obrázky, ale jednotná podpora vektorového formátu (např. SVG) v prohlížečích se nezdá být ideální [17]. Proto se spokojíme s aproximací nerealizovatelných tvarů hranatými variantami (obdélníky či čtverce místo kruhů, kombinace vodorovných a svislých čar místo šikmých apod.).

Je vhodné zmínit se o potenciálním řešení tohoto problému, které nabízí HTML element `<canvas>` [13], [9]. Řešení lze považovat za pouze potenciální z několika důvodů. Jednak tento element není součástí HTML 4.01, ale HTML 5, které je zatím ve stádiu návrhu (draft). Dalším problémem je absence podpory v MSIE, nicméně existuje několik implementací, které umožňují podporu do MSIE doplnit [13]. S touto technologií jsem krátce experimentoval a dá se říci, že prvotní testy dopadly poměrně dobře. Použitelnost této technologie také dokazuje zajímavý projekt, kterým je čistě JavaScriptová implementace emulátoru 8bitového počítače MSX nazvaná *jMSX*, URL http://jsmsx.sourceforge.net/. Tento emulátor pro svoji činnost samozřejmě využívá element `<canvas>`. Použití tohoto elementu nebylo pro účely realizace této práce nikdy vážně zvažováno. Mohlo by se zdát, že důvodem byly uvedené nedostatky, ale hlavním důvodem je skutečnost, že jsem tuto technologii objevil teprve nedávno (a víceméně náhodou). Protože ale představuje zajímavou alternativu, bylo by škoda se o ní nezmínit.

4.3 Analýza současného stavu vývoje

Výsledkem stávajícího vývoje v rámci ročníkového projektu je aplikace, která slouží pro demonstraci *nelineárních dynamických* DS (tuto množinu DS jsme po dohodě s vedoucím

uvedené práce vymezili na stromy). Výsledná aplikace je plně funkční, ale podporuje zatím pouze jedinou dynamickou DS a sice binární vyhledávací strom. Aplikace se jmenuje *DHTML Tree*.

Na uvedené DS je implementována kompletní sada vybraných operací v tzv. **demonstračním režimu** (demonstration mode), tedy takových operací, které provádějí vizualizaci svého průběhu. Dvě z těchto operací jsou dále k dispozici i v tzv. **normálním režimu** (normal mode), což znamená, že probíhají okamžitě bez jakékoliv vizualizace, vidět je pouze výsledek. Jedna z těchto operací slouží ke zrušení celého stromu a je přístupná zvláštním tlačítkem. Druhá slouží pro vložení uzlu a není k ní v tuto chvíli prostřednictvím GUI přístup (byla používána při vývoji pro vytvoření počátečního obsahu stromu). Tuto množinu může být vhodné doplnit o chybějící operace včetně zajištění jejich zpřístupnění pomocí GUI, jako je tomu u operací v *demonstračním režimu*.

Srovnáme-li dosavadní stav se zadáním navazující práce, vidíme, že změnou v novém zadání je změna okruhu DS, které mají být podporovány. Zcela novým požadavkem je implementace výukového režimu. Především tyto změny budou určovat směr dalšího vývoje.

Novou oblastí zpracovávaných DS je obecně množina *dynamických* DS, tedy jde o rozšíření původní množiny. Úkolem proto bude vypracovat podporu nově požadovaných DS.

Z bližšího pohledu je však zřejmá závažná skutečnost: Původní aplikace byla napsána pouze pro podporu stromů. U jednotlivých typů stromů lze najít velké podobnosti, z čehož také vyplynula poměrně jednodušší struktura aplikace. Např. bylo počítáno s obdobným repertoárem operací nad jednotlivými podporovanými DS, což se odrazilo mj. i v tom, že aplikace podporovala jedinou sadu ovládacích prvků pro spouštění těchto operací. S tím zároveň souvisí další části kódu, které také nepočítají s podporou více odlišných struktur. Pokud by se v rámci dřívějšího vývoje podařilo implementovat další stromy, je možné, že by se i zde projevily jisté rozdíly. Je otázka, zda by v těchto případech postačilo místní ošetření jednotlivých rozdílů, nebo zda by si návrh již v této době vyžádal zásadnější strukturální změny. Pro další vývoj se však zdají být velmi vhodné, ne-li nevyhnutelné.

Tyto změny se tedy týkají, nejen GUI ale i vnitřní struktury aplikace. Bude třeba nejen zavést podporu různé sady ovládacích prvků pro jednotlivé DS ale také provést takové změny kódu, aby byl schopen podporovat jejich odlišné vlastnosti.

Z hlediska GUI je nutná i zdánlivě kosmetická změna: GUI aplikace používá záložky, kterými uživatel vybírá požadovanou DS. Tyto záložky byly doposud umístěny nad pracovní plochou, což korespondovalo s tím, že zbývající části GUI (včetně ovládacích prvků) již byly společné. Karty jednotlivých DS tedy obsahovaly pouze pracovní plochu, na které byla DS zobrazena a na které s ní mohl uživatel pracovat. Toto však již, jak jsme právě vysvětlili, nebude vyhovovat. Abychom popsané změny odrazili v uživatelském rozhraní aplikace, přesuneme na jednotlivé karty také zbývající části GUI, tedy stavový panel i ovládací prvky. Pro každou DS tak vznikne zcela samostatná karta, která bude potenciálně moci podporovat zcela jiné rozhraní pro každou DS.

Nejde však o čistě kosmetickou změnu. Původní aplikace obsahovala nejen jedinou sadu ovládacích prvků a jediný stavový panel, ale i jedinou pracovní plochu. Přepínání pracovních ploch bylo simulováno skrytím aktuálního obsahu (zejména vizualizace příslušné DS) a zobrazením obsahu příslušejícího nově zvolené DS. Podle popsaného návrhu bude vhodné zavést skutečné oddělené instance jednotlivých prvků GUI (pracovní plochy, stavového panelu a ovládacích prvků) včetně HTML elementu, který je bude obsahovat (a tak realizovat kartu příslušející k dané záložce). Tato změna zjednoduší přepínání jednotlivých karet, protože bude pouze stačit skrýt celou kartu a zobrazit nově zvolenou. Díky popsanému zanoření obsahu karty v příslušném elementu (v rámci objektového modelu dokumentu) se společně

s elementem karty skryje či zobrazí i její obsah.

Tato zdánlivě kosmetická změna si vyžádá větší reorganizaci kódu, který vytváří GUI aplikace. Bude třeba umožnit vytvoření více instancí jednotlivých částí a navíc vyžadujeme, aby některé části (ovládací prvky) mohly být různé. Poslední požadavek budeme řešit zároveň s řešením podpory odlišných vlastností různých DS (viz 4.6). Z popsaného také vyplývá, že jedním z důsledků úprav bude změna vnitřních datových struktur, ve kterých si aplikace uchovává odkazy (reference) na jednotlivé součásti GUI.

Tím se dostáváme k dalšímu problému, který je vhodné pro následující vývoj řešit. Kód samotné aplikace používá globální proměnné (toto se netýká implementace DS ani vizualizačních komponent, které jsou implementovány objektově). Tato skutečnost není způsobena ani tak ignorancí doporučených zásad programování jako spíš potřebou nějak zpřístupnit jednotlivé části GUI z obslužných podprogramů událostí (event handler). U těchto podprogramů často nebývá možnost přístupu k potřebným datům jiným způsobem nežli přes globální proměnné (tento problém nastává v různých programovacích jazycích a není specifický pro zvolenou platformu).

Konečně nezapomínejme na další důležitý úkol, kterým je podpora výukového režimu. Doplnění této nové funkčnosti si taktéž vyžádá některé změny ve struktuře aplikace, zejména pokud jde o vztahy a způsob komunikace mezi jednotlivými jejími součástmi.

Při řešení jednotlivých etap s výhodou využijeme již hotový BVS, který nám pomůže prototypovat a testovat navržené změny.

4.4 Odstranění globálních proměnných

I když jsme definovali několik úkolů, kterými se budeme dále zabývat, je nutné poznamenat, že některé z nich lze řešit současně. Např. potřeba odstranění globálních proměnných je podpořena restrukturalizací GUI a souvisejících vnitřních struktur aplikace a je vhodné je provádět společně. Pro přehlednost však jednotlivé kroky popíšeme postupně.

Současný kód aplikace má podobu několika globálních proměnných a většího množství funkcí. Většina proměnných obsahuje odkazy na jednotlivé elementy tvořící GUI. V dalších si aplikace udržuje např. odkazy na objekty implementující jednotlivé datové struktury.

Odstranění globálních proměnných můžeme provést převedením aplikace do objektové podoby. Funkce převedeme na metody objektu aplikace a globální proměnné se stanou vlastnostmi tohoto objektu. Tím pádem budou mít metody aplikace automaticky přístup k příslušným vlastnostem. Metodami budou samozřejmě i příslušné podprogramy pro obsluhu událostí, čímž se vyřeší hlavní problém jejich přístupu k potřebným údajům.

4.5 Změny struktury uživatelského rozhraní

Požadujeme takovou změnu struktury GUI, která bude podporovat a také odrážet rozdílnost jednotlivých DS. Tyto změny se projeví jak po vizuální stránce tak i ve struktuře kódu.

Bude třeba provést změnu struktury stávajícího kódu, který v dokumentu vytváří elementy, z nichž je složeno celé GUI. Tato změna spočívá v rozdělení kódu tak, aby bylo možné některé části volat vícekrát, což umožní vytvořit více karet namísto původní jediné. Toto se tedy týká i kódu vytvářejícího jednotlivé komponenty, které tvoří obsah karty. Vizuální změnu celého GUI (zejména změnu polohy pruhu záložek) pak provedeme úpravami kódu, který řeší rozmístění jednotlivých komponent.

Kromě toho minimálně pro ovládací prvky požadujeme, aby aplikace byla schopná vytvořit odlišné sady těchto prvků pro různé DS. Tuto schopnost zatím nebudeme požadovat pro pracovní plochu ani pro stavový panel, čímž se zachová alespoň základní jednotný vzhled aplikace. Uvedené komponenty budeme s největší pravděpodobností využívat u všech DS, ale společné ovládací prvky nebudou v žádném případě vyhovující.

Poslední požadavek k již popsané změně struktury přináší další úkol navíc. Bude třeba umožnit, aby část kódu, která vytváří ovládací prvky, byla volitelná podle konkrétní DS. Jak jsme již zmínili, tento požadavek budeme řešit v rámci obecnějšího úkolu, kterým bude celková podpora odlišností jednotlivých DS.

4.6 Abstrakce odlišností datových struktur

Protože potřebujeme být schopni pracovat i s poměrně odlišnými DS, je třeba zavést nějaký koncept, který nám umožní abstrahovat hlavní kód aplikace od těchto rozdílů. Je asi pochopitelné, že nebudeme chtít (a možná ani moci) zcela unifikovat funkčnost objektů implementujících jednotlivé DS natolik, aby se všemi aplikace mohla zacházet jednotně.

Jak již víme, pro každou DS chceme podporovat různé operace, které se mezi jednotlivými DS obecně liší. To se však neodráží jen v tom, že potřebujeme různé ovládací prvky pro volbu jednotlivých operací. Aplikace musí vědět, jakou metodu pro danou operaci volat, jaké parametry tato metoda vyžaduje a odkud či jakým způsobem má získat jejich hodnoty.

Obdobně také návratové hodnoty jednotlivých operací bývají různého typu (a případně mohou vyžadovat různou interpretaci), ale tento detail můžeme pro další vývoj ignorovat díky dynamickému typování JavaScriptu, díky čemuž můžeme výsledek různých operací zobrazovat společným způsobem. Je ale vhodné při návrhu počítat i s touto možností a vytvořit koncepci umožňující pozdější doplnění specifické prezentace výsledků každé operace.

Kromě potřeby zvládnutí popsaných rozdílů se mohou v průběhu vývoje objevit i další odlišnosti, se kterými může být třeba se vyrovnat. Naším úkolem tedy je navrhnout takovou strukturu kódu, která nám umožní co největší obecnost. Přitom zároveň vyžadujeme, aby výsledná struktura byla dobře pochopitelná a umožňovala snadné rozšíření o podporu dalších datových struktur. A i když jsme vyloučili možnost unifikace objektů implementujících jednotlivé DS, nebude na škodu, když bude možné, aby aplikace mohla shodně zacházet s několika různými DS, které jsou si dostatečně podobné (tedy máme zájem o částečné umožnění zjednodušené struktury tak, jak je ve stávající implementaci).

Možným řešením by bylo doplnění kódu, který zajistí potřebnou unifikaci, přímo do objektu implementujícího DS. Nicméně toto, jak již bylo v úvodu této části poznamenáno, nechceme. Důvodem je dodržení omezené sémantiky objektů implementujících DS. Je třeba přiznat, že při této snaze jsem poněkud zapomněl na skutečnost, že tyto objekty již nyní překračují funkčnost jimi implementované DS – řeší totiž např. vlastní vizualizaci. Přesto však lze říci, že tato funkčnost je v případě aplikace tohoto druhu na místě. Vždyť tyto objekty slouží jako DS pro demonstrační a výukové účely, proto je zahrnutí této funkčnosti zcela přirozené. Na druhé straně zahrnutí kódu, který by zajistil potřebnou unifikaci již tento rámec výrazně překračuje. Tato podpora je spíše zodpovědností aplikace. Kromě toho by právě postup, který jsme právě zavrhli, neumožňoval znovupoužití podpůrného kódu pro více podobných struktur.

Přesto, že popsaná cesta nebyla správnou, přesunutí příslušné funkčnosti do objektu není špatným nápadem. Pro tyto účely ale vytvoříme nový, tzv. **podpůrný objekt**. Podpora jednotlivých DS je již nyní v aplikaci konfigurována uvedením příslušného objektu, který tuto DS implementuje. Při zavedení popsaného konceptu tedy pouze doplníme tuto

konfiguraci o příslušný *podpůrný objekt*. Do tohoto objektu bude převedena veškerá funkčnost, kterou dosud realizovala aplikace a která se může mezi jednotlivými DS lišit. Instance téhož *podpůrného objektu* mohou být použity pro více DS, což splňuje náš další požadavek. Podpůrný objekt bude mít jednotné rozhraní k aplikaci (které bude zajišťovat např. vytvoření instance objektu implementujícího DS) a bude přizpůsoben pro určitou třídu DS se stejným rozhraním. V krajním případě bude daný *podpůrný objekt* podporovat jen jednu DS, což ale není na závadu. Výhodou oddělení funkcí jednotlivých objektů zůstává lepší struktura výsledného kódu a jeho snazší pochopení.

Navržená struktura odpovídá objektovému návrhovému vzoru [2]. Podpůrný objekt zde vystupuje jako strategický objekt podle návrhového vzoru *strategie*.

4.7 Fázované operace

Než se začneme zabývat řešením dalšího problému, který jsme si vytyčili v úvodu této kapitoly, bude vhodné popsat koncept, jenž již byl v aplikaci během stávajícího vývoje použit a na kterém budeme stavět další výklad.

Jedná se o koncept tzv. **fázované operace** (phased operation). Jde o způsob řešení jedné z nepříjemností implementační platformy, která spočívá v tom, že k překreslení obsahu okna prohlížeče obecně dojde až po skončení běhu skriptu.

Pokud by překreslování bylo prováděno průběžně, byla by implementace jednotlivých operací DS v *demonstračním režimu* poměrně přímočará. V zásadě by stačilo doplnit běžný algoritmus (jak je definován např. v [4]) o kód, který zajistí potřebné zobrazení průběhu demonstrované operace (např. změnu barvy právě procházeného uzlu apod.).

Bohužel skutečnost je jiná a právě proto byl zaveden popisovaný koncept. Jeho podstata spočívá v takové úpravě algoritmů jednotlivých operací, aby je bylo možno provádět postupně po jednotlivých fázích. Po skončení každé takové fáze dojde k okamžitému návratu z funkce, která danou operaci implementuje, čímž je umožněno ukončení běhu skriptu a tak může dojít k potřebnému překreslení obsahu okna prohlížeče.

Jde tedy o poměrně jednoduchý princip. Bohužel převod algoritmů do fázovaného tvaru není zcela přímočarý. Aby vůbec bylo možné realizovat funkci, která po každém zavolání pokračuje krokem následujícím po tom právě provedeném, je také nutné uchovávat si stav této funkce mezi jednotlivými zavoláními. Jednou z možností je použití statických proměnných, které je však samo o sobě zdrojem různých problémů (např. taková funkce je z principu nereentrantní). Také je v tomto případě složitější zajistit rozumným způsobem inicializaci těchto proměnných (mimo funkci nejsou viditelné). Mimo to se zdá, že statické proměnné nejsou v JavaScriptu přímo podporovány a proto tuto možnost nepoužijeme.

Raději využijeme možnosti uložení stavu mimo příslušnou funkci. Pro tyto účely si vytvoříme zvláštní objekt, u kterého využijeme skutečnosti, že je vždy předáván odkazem. Tudíž jej můžeme při volání *fázované operace* předat funkci, která ji realizuje, a ta si do něj může ukládat informace o průběhu operace. Zároveň využijeme toho, že objekty v JavaScriptu mají také vlastnosti strukturovaného typu (známého jako *struktura* či *záznam*), takže takový objekt nám poskytne strukturované úložiště stavových informací.

Popsaný objekt budeme nazývat **pokračovací kontext** (resume context). Při jeho použití využijeme skutečnosti, že vlastnosti objektů v JavaScriptu vznikají dynamicky (není třeba je nikde deklarovat). Do kontextového objektu budeme ukládat např. různé příznaky udávající, jaké kroky již byly provedeny. Pokud se bude jednat o boolovské příznaky, můžeme s výhodou využít i toho, že neexistující složky struktury mají hodnotu `undefined`,

která se při vyhodnocování boolovského výrazu rovná `false`. Proto můžeme takové příznaky bezpečně testovat ještě předtím, než je nastavíme, a teprve když nastane příslušná událost, je můžeme změnit na `true`. Kromě těchto příznaků bude třeba do kontextového objektu ukládat různé lokální proměnné, jejichž životnost je delší než jednu fázi operace.

4.8 Podpora výukového režimu

Nyní se podívejme, jaké změny bude vyžadovat implementace výukového režimu. Tento režim společně s *normálním* a *demonstračním* režimem tvoří trojici režimů, které bude výsledná aplikace podporovat. V rámci aplikace je označen jako **zkušební režim** (examination mode). Nejdříve však specifikujme, co si pod tímto pojmem představit.

Zadání tento režim definuje jako *speciální interaktivní mód, který umožní studentovi ověřit jeho vědomosti.* Realizaci tohoto režimu tedy navrhneme např. tak, že po volbě operace, jejíž znalosti mají být prověřeny, bude uživatel interaktivním způsobem realizovat jednotlivé kroky podle pokynů aplikace. Např. při vkládání nového uzlu do stromu může být úkolem uživatele určit rodičovský uzel vkládaného uzlu a zároveň určit, zda nový uzel bude levým nebo pravým synem.

Odpovědi uživatele by měly probíhat pokud možno co nejpřirozenějším způsobem s návazností na zobrazenou strukturu. To znamená, že odpovědi by měly probíhat s využitím tohoto zobrazení, např. kliknutím na příslušný uzel stromu. Také je vhodné, aby změny prováděné v rámci zkoušené operace byly pokud možno zobrazovány přímo v jejím průběhu.

Z hlediska implementace GUI tyto požadavky přinášejí nové potřeby. Pro to, aby mohla interakce probíhat pomocí kliknutí na části zobrazené DS, bude třeba rozšířit systém obslužných rutin událostí např. právě o obsluhu události `onclick` pro jednotlivé části zobrazené DS. Stávající aplikace již této možnosti využívala pro přenos hodnoty uzlu, na který uživatel klikl, do vstupního pole pro zadání hodnoty, takže tato hodnota pak mohla být použita jako argument pro další operaci. Tuto příjemnou možnost budeme chtít zachovat, ale přitom budeme chtít podporovat právě navrhované nové funkce. Bude tedy třeba provést změnu způsobu předávání událostí tak, aby bylo možné zajistit oba způsoby jejich zpracování.

Kromě této změny by mohlo být vhodné rozšířit implementaci vizualizace jednotlivých DS tak, aby bylo umožněno kliknutí na některé specifické části (které pro běžnou vizualizaci nejsou přímo potřeba) zobrazené DS. Třeba u stromu (např. v případě již diskutované operace vložení uzlu) by mohlo být vhodné umožnit určení syna (levý vs. pravý) kliknutím na rodičovský uzel v místě, odkud bude vycházet hrana k příslušnému synovskému uzlu. Alternativou může být zobrazení speciálních objektů ve funkci nulových ukazatelů, kliknutím na něž by uživatel mohl tento výběr provést. A pro realizaci operace vložení uzlu tak, jak byla popsána, bude také vhodné zobrazovat pseudouzel reprezentující ukazatel na kořenový uzel; tento pseudouzel by pak byl „rodičem“ uzlu vkládaného do prázdného stromu.

Ještě větší možnosti k zamyšlení v tomto směru přináší např. operace rušení uzlu BVS. Tato operace má několik různých variant podle toho, kolik podstromů má rušený uzel [4, s. 131]. Zde stojí za zvážení, jak hodně by měla operace ve *zkušebním režimu* uživatele navádět. Asi nejúčinější prověření by umožňovala taková forma, kdy by uživatel zcela samostatně prováděl všechny kroky včetně přesunů hodnot, rušení jednotlivých uzlů a změn vazeb mezi uzly. Toto provedení však vede na poměrně pracnější implementaci. Pravděpodobně se raději spokojíme s jednodušší variantou, která bude uživatele více vést a tudíž vystačí s jednodušší modelem interaktivity. Bude však vhodné, pokud výsledná struktura celého kódu nebude komplikovat případné další vylepšení tímto směrem.

4.8.1 Změna struktury komunikace

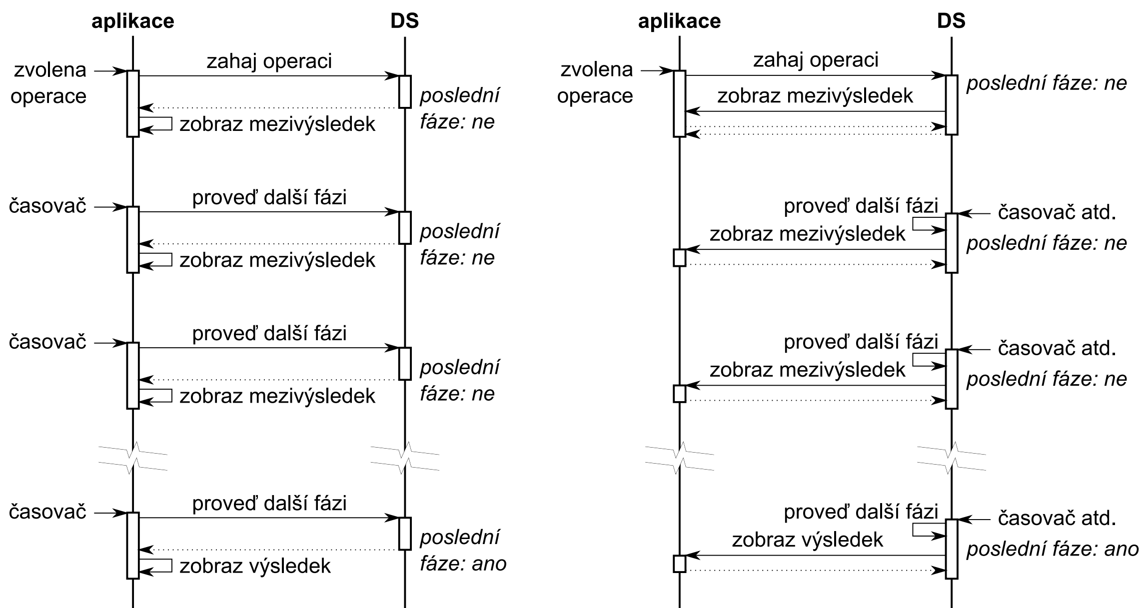
Ukázalo, že stávající struktura komunikace mezi komponentami aplikace již není pro rozumnou implementaci výukového režimu vyhovující. Tato struktura vyhovovala v době, kdy byl požadován pouze demonstrační režim, ale není už schopna dobře splnit nové požadavky.

Pro další popis využijeme diagramů na obr. 4.1. Použitá notace je inspirována interakčními diagramy užívanými v [2]. Svislé čáry představují jednotlivé objekty v časové ose, přičemž čas plyne shora dolů. Svislé obdélníky zobrazují časové úseky aktivity jednotlivých objektů. Během znázorněného úseku buď probíhá kód metody přímo aktivního objektu, nebo tento čeká na návrat z metody, kterou zavolal (v tomto intervalu je znázorněna také aktivita objektu, jehož metoda byla volána).

Objekt 'aplikace' na obr. 4.1(a) koresponduje s dosavadním neobjektovým kódem této aplikace. Funkčnost téhož objektu na obr. 4.1(b) bude ve skutečnosti implementována konkrétním *podpůrným objektem* pro danou DS (na který však lze pohlížet jako na součást aplikace. Objekt 'DS' v obou případech představuje objekt implementující příslušnou DS.

Vodorovné souvislé šipky směřující mezi objekty reprezentují volání metod (na které lze pohlížet jako na předávání zpráv) a směřují od volajícího objektu k volanému. Tečkované šipky představují návrat z volané metody (návraty z volání, kdy objekt volal svoji vlastní metodu, nejsou kvůli zachování přehlednosti zakresleny). Směr těchto šipek je opačný, odpovídá přesunu toku řízení při návratu z volané metody. Šipky směřující k objektu, které nevedou od jiného objektu, představují události vzhledem k objektům externí.

Informace, zda právě proběhlá fáze *fázované operace* byla poslední fází, je zobrazena skloněným písmem (tuto informaci si v původním modelu aplikace získávala explicitně, v novém modelu se ji dozví při obdržení požadavku na zpracování výsledku nebo mezivýsledku). Ostatní texty popisují volané metody a externí události.



(a) Původní struktura (b) Nově navržená struktura

Obrázek 4.1: Diagramy struktury komunikace

Jak je vidět na obr. 4.1(a), základní princip původní struktury spočíval v tom, že provádění jednotlivých fází *fázovaných operací* řídila sama aplikace. Zažádala si o zavolání příslušné metody po uplynutí požadovaného časového intervalu a v této metodě pak zaslala příslušnou žádost objektu implementujícímu DS. Po návratu aplikace obdržela výsledek operace (buď konečný nebo průběžný – podle toho, zda se jednalo o poslední fázi nebo ne) a zobrazila jej.

Nevýhoda tohoto řešení se projeví ve chvíli, kdy začneme uvažovat o potřebách operací ve *zkušebním režimu*. Také ony budou muset být řešeny jako *fázované operace*. Jednak proto, že potřebujeme aby probíhalo překreslování, ale také proto, že je to asi nejpřirozenější způsob, jak je integrovat s obsluhou událostí. Jak jsme již uvedli, ve *zkušebním režimu* by DS měla reagovat např. na kliknutí myší ale i na jiné události. A tím se dostáváme k jádru problému.

Stávající struktura podporuje pouze reakci na událost časovače. (Ve skutečnosti se nejedná z pohledu použitých technologií o událost jako takovou, ale je tu podobnost v tom, že stejně jako u událostí vede uplynutí požadovaného intervalu na volání nějaké zadané obslužné funkce. Z tohoto důvodu si můžeme dovolit malou nepřesnost v podobě zahrnutí mechanismu časovače k událostem.) My ale nadále potřebujeme schopnost reakce na širší spektrum událostí, která navíc nemusí být předem pevně dáno. A i když bychom byli schopni tuto množinu předem vymezit, bylo by nutné, aby aplikace (resp. *podpůrný objekt*) v každém okamžiku věděla, jaké události objekt implementující DS v dané chvíli očekává a jaké ne. Zejména by aplikace musela řešit, zda „natáhnout" časovač nebo zda bude operace pokračovat pouze po jiné události.

Jistě by bylo možné pro tyto účely vytvořit nějaký model pro předávaní těchto informací. Mnohem snazší se však jeví převést potřebnou funkčnost do objektu, který implementuje DS a umožnit mu tak, aby si zajistil potřebné podmínky sám. Tento model znázorňuje obr. 4.1(b). Začátek celého procesu je stejný, aplikace předá objektu požadavek na prováděnou operaci včetně parametrů. Přitom navíc předá implementujícímu objektu odkazy (reference) na metody (call-back funkce), které mají být volány za účelem zpracování výsledků. Objekt DS si pak už sám zařídí potřebné podmínky pro pokračování operace v podobě „natažení časovače" nebo registrace potřebných obslužných rutin událostí. (Ty mohou být ve skutečnosti zaregistrovány stále, pouze může být jejich volání ignorováno, nečeká-li DS na takové události. Toto řešení je i vhodnější s ohledem na to, že vizuální reprezentace DS může obsahovat větší množství HTML elementů, na každém z nichž by musela být registrace prováděna.) Příslušné call-back funkce kromě zobrazení výsledků také umožňují reakci na dokončení operace nad DS adekvátním způsobem.

4.9 Implementace dalších datových struktur

K vývoji podpory dalších DS je nanejvýš vhodné přistoupit teprve ve chvíli, kdy již budou dokončeny předchozí kroky. Kdybychom tento postup nedodrželi, byli bychom nuceni dosud popsané změny realizovat i na všech nově implementovaných DS. Na druhé straně implementace dalších DS po realizaci uvedených kroků prověří kvality našeho návrhu.

Pro tento krok je nutné učinit řadu dalších návrhových rozhodnutí: Určit způsob vizualizace DS, množinu podporovaných operací, zvolit způsob zobrazení průběhu v *demonstračním režimu*, vybrat vhodné způsoby interakce ve *zkušebním režimu* atd. Ačkoliv se jedná o nezanedbatelný počet kroků, které je v tomto směru třeba provést, související rozhodnutí již nejsou tak zásadního koncepčního charakteru jako ta doposud prezentovaná.

Kapitola 5

Popis implementace

Dále již budeme prezentovat konkrétní výsledky praktické části této práce. Naším úkolem je popsat výslednou implementaci aplikace podle návrhu popsaného v kapitole 4.

5.1 Skutečný rozsah implementace

Pro vymezení základního rámce dalšího popisu bude vhodné nejdříve uvést, do jaké míry se podařilo splnit původní požadavky a předpoklady.

Výsledkem je aplikace pojmenovaná *Dynamitorial* (název je odvozen od označení '*Dynamic* Data Structures Tu*torial*'). Jedná se o zcela funkční aplikaci, jejíž implementované části jsou ve stavu, který lze považovat za dokončený. Tím je myšleno, že dané části jsou plně funkční a zároveň jejich funkčnost lze považovat za kompletní; možnost jejich dalšího vylepšení nebo rozvoje tímto samozřejmě není vyloučena.

Z uvedeného také vyplývá, že ne veškerá funkčnost byla implementována. Co vše tedy bylo implementováno a co ne? Především byly zapracovány koncepční změny, které byly navrženy a jejichž realizaci jsme stanovili jako prerekvizitu pro další rozvoj. Výsledný kód je tedy nyní připraven pro podporu různých DS a to včetně podpory operací ve všech 3 režimech (viz 4.3 a 4.8).

Dále byla doplněna již dříve vytvořená implementace BVS, která byla rozšířena o kompletní množinu operací v *normálním režimu*. Na této DS byla také implementována 1 operace (přidání nového uzlu) ve *zkušebním režimu*. Podpora zbývajících operací téhož režimu nebyla implementována především z časových důvodů. Touto částí práce byl úspěšně prověřen koncept, jehož návrh je popsán v 4.8.

Za účelem potvrzení úspěšnosti další části návrhu byla implementována nová DS, kterou je (jednosměrný) seznam. Nad touto strukturou je implementována téměř kompletní sada operací v *normálním režimu*, která je popsána v [4, s. 47]. Byly vynechány operace získání hodnoty prvního a aktivního prvku a dále některé predikáty a pomocné operace. Vynechané operace lze snadno doplnit, problémem se jeví jen skutečnost, že v GUI již není příliš volného místa pro další ovládací prvky. Jedním z možných řešení je jejich reorganizace, jejíž vhodná podoba by mohla spočívat v umístění tlačítek pro volbu párových operací (tj. operace nad prvním prvkem a obdobné operace na prvkem aktivním) ve dvojicích vedle sebe na jeden řádek. Operace v dalších režimech nebyly implementovány zejména z časových důvodů. Zavedením dosud nepodporované DS (která je nepochybně od již podporovaného BVS odlišná) se podařilo úspěšně prověřit návrhy popsané v 4.5 a 4.5.

V porovnání s původními požadavky tedy vidíme, že mezi chybějící funkce patří podpora

dalších zadaných struktur (zásobník a fronta). Dále je třeba dokončit operace ve zbývajících 2 režimech u seznamu (a v případě potřeby ještě přidat další) a zbývající operace ve *zkušebním režimu* u BVS.

5.2 Podporované prohlížeče

Popisujeme-li skutečnou implementaci aplikace, která běží ve webovém prohlížeči, neměli bychom zapomenout také uvést, v jakých prohlížečích byla aplikace vyvíjena a testována. Ačkoliv byly používány technologie, které nejsou závislé na konkrétním prohlížeči, v praxi se lze setkat s různým stupněm podpory té které technologie v konkrétním prohlížeči.

Obecně platí, že aplikace by měla být schopna správně fungovat v každém prohlížeči, který implementuje technologie DHTML, které byly popsány v 3.3 ve verzích, jejichž standardy zde byly citovány. Podle těchto standardů byla aplikace implementována. Problematickou by se mohla stát snad jedině množina použitých vlastností dle DOM Level 0, který není formálně specifikován.

Hlavní vývojové a testovací práce na aplikaci probíhaly s využitím prohlížečů *SeaMonkey* verze 1.0.1 a *Internet Explorer* verze 6.0 SP1 (při práci ve Windows 2000) příp. 6.0 SP2 (ve Windows XP). Předchozí vývoj v rámci ročníkového projektu probíhal ve verzích dostupných v této době, které jsou uveden v 3 (vzhledem k tomu, že většina kódu byla v rámci dalšího vývoje otestována a přepracována, není tento údaj až tak důležitý).

Aplikace byla dále testována v aktuálních verzích prohlížečů, které jsou dostupné v počítačových laboratořích FIT VUT v Brně, tedy v prohlížečích *Firefox* verze 2.0 a *Internet Explorer* verze 7.0. (Při této příležitosti byla provedena jedna drobnější oprava kvůli drobné změně chování v uvedené verzí *Internet Explorer*u.)

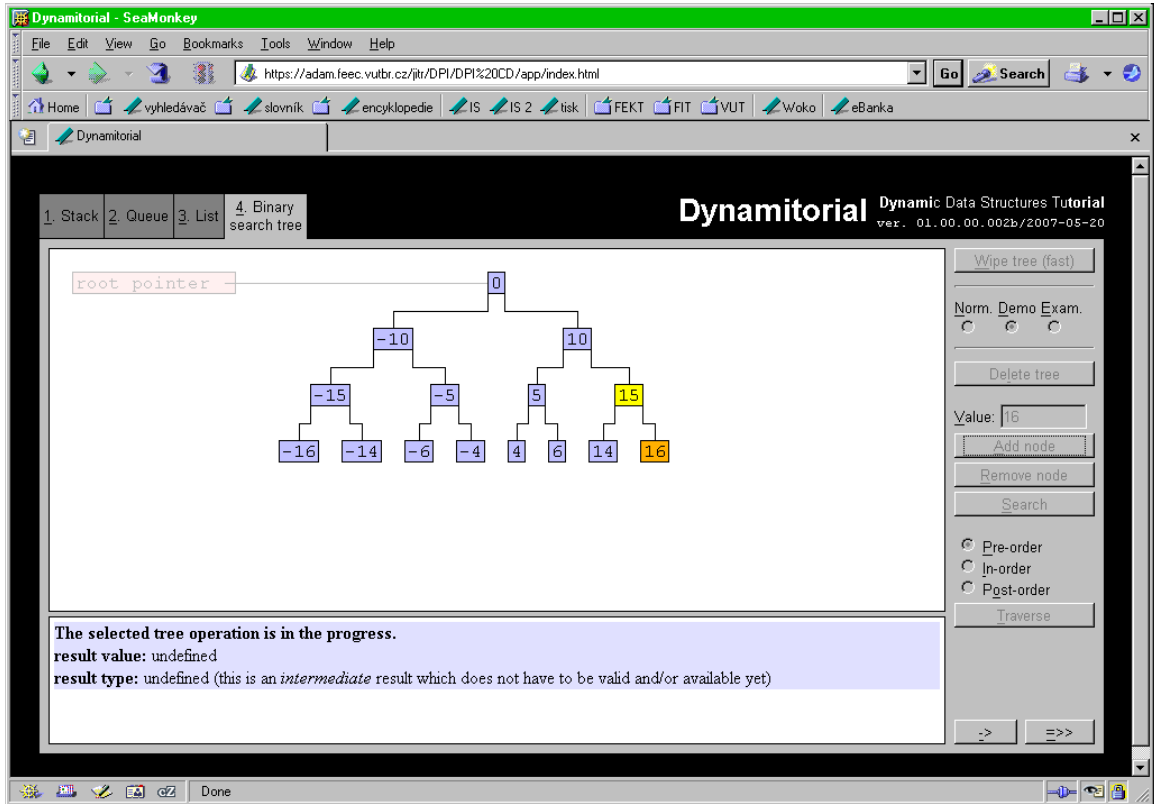
Skutečnost, že aplikace bez zásadních úprav fungovala i v novějších verzích prohlížečů poněkud potvrzuje portabilitu výsledného kódu. Z tohoto hlediska by ale bylo zajímavější testování na zcela jiném prohlížeči. Této možnosti nebylo využito pro minimální zkušenosti s jinými prohlížeči, protože nejsou běžně k dispozici na počítačích, ke kterým mám přístup, a také z časových důvodů.

5.3 Uživatelské rozhraní

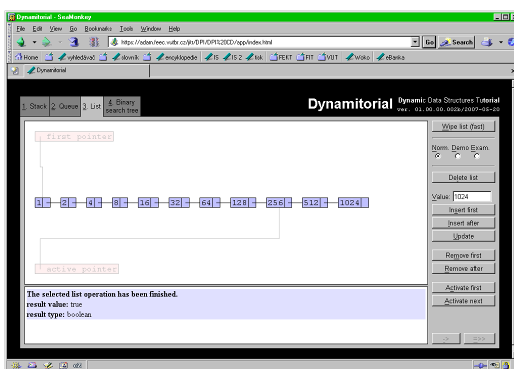
Výsledná podoba uživatelského rozhraní, které odráží navržené změny, je vidět na obr. 5.1 a 5.2. Jeho rozložení bylo inspirováno aplikací *Tree Animation Tool* (viz 3.1). Jednotlivé části tohoto GUI jsou popsány v A.1. (Pro porovnání lze na obr. 5.3 vidět i rozhraní původní aplikace.) Aplikace s uživatelem komunikuje v angličtině.

GUI aplikace bylo navrhováno při použití rozlišení 1024×768. Toto rozlišení v dnešní době jistě nepředstavuje výjimečný požadavek, možná by se dalo říci, že dnešní standard je již o něco náročnější. Kód aplikace však byl napsán s důrazem na možnost snadné změny rozměrů celého rozhraní. Geometrie jednotlivých částí se určuje ze dvou parametrů – celkové šířky a výšky. Díky tomu není složitá změna rozměrů podle potřeby, stačí v kódu upravit pouhé dvě hodnoty.

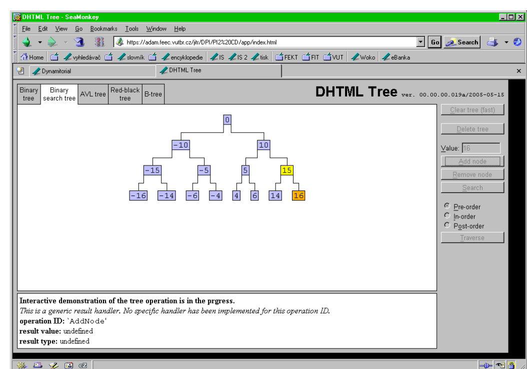
Po restrukturalizaci kódu se tyto parametry předávají přímo při vytváření aplikace ze souboru `index.html`. Je tedy možné doplnit např. i podporu zjištění velikosti okna a vytvořit aplikaci v takové velikosti, která plochu okna prohlížeče maximálně využije. Vzhledem k tomu, že kód pro výpočet geometrie byl vyfaktorizován do zvláštní metody, je



Obrázek 5.1: Uživatelské rozhraní nové aplikace (zobrazen BVS, na kterém právě v *demonstračním režimu* probíhá operace přidání nového uzlu s hodnotou 16



Obrázek 5.2: Uživatelské rozhraní nové aplikace (zobrazen seznam)



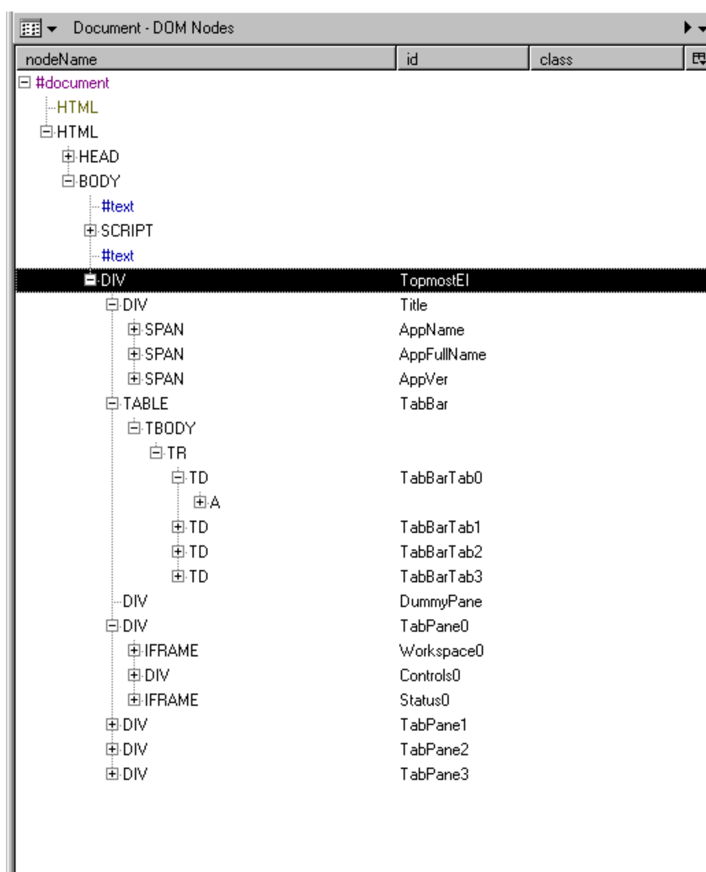
Obrázek 5.3: Uživatelské rozhraní původní aplikace *DHTML Tree*

možné doplnit i podporu změny velikosti aplikace při změně velikosti okna prohlížeče. Tato podpora spočívá pouze v aplikaci nových geometrických parametrů jednotlivým elementům.

Aplikace je tedy napsána nezávisle na konkrétní velikosti svého hlavního elementu. Pracovní plocha automaticky zabere dostupný prostor, pevně jsou dány pouze šířka elementu s ovládacími prvky a výška stavového panelu. Jediné omezení v tomto směru proto představují minimální rozměry, které jsou dány uvedenými pevnými rozměry a minimálními rozumnými rozměry pracovní plochy.

Polohování hlavního elementu v rámci okna prohlížeče je řešeno následovně: Pro horizontální centrování je nově využito CSS polohování, které zajistí automatickou reakci na změnu šířky okna bez nutnosti provedení skriptu. Vertikální centrování není nijak ošetřeno, jeho řešení by vyžadovalo použití skriptu, který by na základě zjištěných rozměrů okna provedl změnu polohy hlavního elementu. Je vhodné zmínit se, že zjištění rozměrů okna prohlížeče je funkčnost hraničící s DOM Level 0.

Na obr. 5.4 lze vidět částečnou vnitřní strukturu GUI tak, jak je implementována pomocí jednotlivých HTML elementů. Obrázek je výřezem zobrazení v DOM inspektoru prohlížeče *SeaMonkey*. Protože tato struktura vychází z dříve prezentovaného návrhu a protože obrázek je snad poměrně názorný, nebudeme ji dále komentovat. Poznamenejme snad jen, že hodnoty atributů `id` korespondují s identifikátory v kódu (číslice jsou zde navíc, neboť hodnoty těchto atributů musí být v celém dokumentu unikátní). Aplikace však tyto atributy nevyužívá, slouží pouze pro zpřehlednění zobrazené struktury.



Obrázek 5.4: Vnitřní struktura GUI

Je třeba vyzdvihnout fakt, že při návrhu GUI byl také kladen důraz na možnost ovládání

aplikace prostřednictvím klávesnice. Jsou např. využívány přístupové klávesy (access keys), které definuje standard HTML. Ze stejného důvodu jsou také pracovní plocha a stavový panel realizovány pomocí elementů `<iframe>`. Zdá se totiž, že poskytují nejlepší podporu ovládání klávesnicí. Některé prohlížeče neumožňují přepínat zaměření (focus) na jiné elementy nežli formulářové prvky a `<iframe>`. Realizace pomocí elementu `<div>` by jinak byla výrazně jednodušší.

I přes snahu o ovladatelnost klávesnicí však došlo k opomenutí v případě vizualizovaných DS. Např. kliknutí na uzel nelze klávesnicí realizovat. Jednotlivé uzly nelze zaměřit jinak, nežli kurzorem myši. Tento problém je možné vyřešit podobnou technikou, jaká byla použita pro realizaci pruhu záložek. Jak lze vidět z obr. 5.4 (element `TabBar` a obsažený `TabBarTab0`), jednotlivé záložky obsahují elementy `<a>`. Ty lze zaměřit i klávesnicí (obvykle klávesou Tab a tím popsaný problém řeší.

Element `<iframe>` přináší nový problém a tím je potřeba počáteční inicializace v něm obsaženého dokumentu. Typickým použitím tohoto elementu je zobrazení existujícího dokumentu, které se provádí nastavením příslušného atributu na URL požadovaného dokumentu. Inicializace prázdného dokumentu ale ve standardu popsána není. Problém se nakonec podařilo vyřešit a výsledkem je funkce `InitEmptyDoc()` v souboru `js/main.js`.

S tímto problémem souvisí ještě zvláštní jev spočívající v tom, že popsanou inicializaci dokumentu lze provádět až tehdy, když je jeho rodičovský element `<iframe>` sám vložen do nadřazeného dokumentu. Vyjmutím tohoto elementu (případně podstromu DOM, který jej obsahuje) z nadřazeného dokumentu bohužel dochází k poškození jeho obsahu. Toto chování bylo pozorováno u obou prohlížečů, které byly při vývoji použity (chybové zprávy MSIE byly ale dost nepochopitelné).

Po restrukturalizaci GUI se nově vyskytl problém v podobě kolizí přístupových kláves mezi jednotlivými kartami. Při přepínání karet totiž dochází pouze ke skrývání a odkrývání příslušných elementů (pomocí CSS vlastnosti `display`). Problém byl vyřešen zrušením všech přístupových kláves na skrývané kartě a jejich obnovením na té odkrývané při jejich přepínání. Metoda odstranění a vkládání elementů při přepínání karet není řešením, protože zde dochází k nežádoucím jevům zejména u elementů `<iframe>` (viz výše).

Dalším problémem souvisejícím s GUI je chyba v MSIE projevující se při pokusu vytvořit metodami DOM přepínací tlačítka (radio buttons). Svázání několika tlačítek do jedné skupiny se zajišťuje nastavením shodné hodnoty atributu `name` všem elementům v této skupině. Nastavení tohoto atributu však v MSIE nemá žádný efekt, díky čemuž nelze skupinu tlačítek vytvořit. Navíc dokonce takto vytvořená tlačítka nefungují ani samostatně. Problém řeší funkce `CreateRadioBtn()` v modulu `js/dom_utils.js`.

Poznamenejme, že i přes problémy s různými prohlížeči bylo snahou psát kód bez ohledu na konkrétní prohlížeč. Jedním z příkladů tohoto přístupu je již popsaná funkce `CreateRadioBtn()`. Dalším příkladem je funkce `GetContentDocument()` z téhož modulu, která v MSIE řeší absenci atributu `contentDocument` (definovaného standardem) použitím jiného atributu téhož významu. Podobné porušení standardu lze najít v MSIE v případě CSS atributu `float`. Ten se má v DOM zobrazit jako objektová vlastnost `cssFloat`, ale MSIE používá `styleFloat`.

Na závěr ještě uveďme poznámku ke způsobu pojmenování jednotlivých operací nad DS v rámci aplikace (resp. k popiskům na tlačítkách, kterými se tyto operace volí): Tlačítka byla pojmenována přirozeným jazykem se snahou o vytvoření logického názvu. Nejsou tedy použita např. označení z [4, s. 47] a [3, s. 44]. Motivací bylo vytvořit aplikaci s obecným použitím neomezeným pouze v rámci FIT VUT v Brně. Názvy z uvedených publikací jsou zde jistě zažité, jinde tomu tak být ale nemusí. V případě potřeby je však změna názvů ope-

rací snadná. Stačí upravit příslušné řetězce v kódu metod `CreateControls()` příslušných *podpůrných objektů*. V těchto řetězcích jsou zároveň definovány přístupové klávesy a proto nehrozí opomenutí jejich změny. Pouze je třeba zajistit, aby po přejmenování nedocházelo ke kolizím kláves v rámci karty.

5.4 Předávání odkazů na metody a uzávěry

Nyní budeme prezentovat techniku nově použitou při přepracování kódu v rámci této práce.

Zajímavou vlastností u objektového modelu v JavaScriptu je skutečnost, že volání metody nemusí nutně probíhat v kontextu objektu, jemuž tato metoda náleží (tj. klíčové slovo `this` může odkazovat na zcela jiný objekt). Ačkoliv jde o zajímavou vlastnost, kterou lze i zajímavě využít (či zneužít?), v některých případech se může stát nepříjemnou.

Popsaný jev se projeví, pokud voláme metodu přes odkaz na ni. Takový odkaz se chová jako standardní odkaz na funkci bez jakékoliv vazby na původní objekt. Bohužel velmi často potřebujeme předat odkaz na funkci v případech, kdy se jedná např. o obslužnou rutinu události. Je-li touto rutinou právě metoda objektu, která ve svém kódu přistupuje k vlastnostem svého objektu, pak bude po zavolání přes odkaz přistupovat ke zcela jinému objektu!

Protože aplikace využívá objekty pro většinu svých činností, je popisovaná vlastnost dost nepříjemná. Např. vizualizační komponenty jsou implementovány jako objekty, které si pro svoji činnost vytvářejí jeden nebo více DOM objektů (resp. HTML elementů) pro svoji vizuální reprezentaci. Nastane-li nějaká událost nad těmito elementy, chceme aby byla volána příslušná metoda objektu, který tyto elementy spravuje. Tato metoda by samozřejmě měla být volána v kontextu příslušející instance spravujícího objektu a nikoliv v kontextu DOM objektu příslušejícího danému elementu. (Obslužné podprogramy událostí v DOM jsou volány v kontextu DOM objektu odpovídajícího elementu, na kterém událost nastala.) Původní aplikace tento problém řešila přidáním vlastnosti obsahující odkaz na spravující objekt do DOM objektu (tedy využila již dříve zmíněné dynamičnosti JavaScriptu).

Ale tato možnost nemusí být vždy k dispozici, typicky např. v případě obslužné funkce předávané funkci `setTimeout()`, která je volána po vypršení požadovaného časového intervalu. Zde není žádný objekt, do kterého bychom mohli přidat nějakou vlastnost. Původní aplikace v tomto případě na problém nenarazila, protože obsluhu prováděl neobjektový kód aplikace. Po převedení aplikace do objektové podoby již ale tento problém nastal a také se nově projevil u obslužných funkcí událostí, které se staly metodami.

Univerzální řešení v tomto směru nabízí použití „obalové“ (wrapper) funkce (dále budeme používat raději slovo **wrapper**). Tento wrapper musí být schopen akceptovat volání v kontextu libovolného objektu a jeho úkolem bude zajistit volání cílové metody ve správném kontextu. Pro volání metody v kontextu zadaného objektu JavaScript poskytuje funkce `call()` a `apply()` (jejich činnost je stejná, liší se formátem argumentů). Wrapper tedy pouze potřebuje znát odkaz na cílovou metodu a odkaz na objekt, v jehož kontextu má být metoda volána.

Nepříjemnou vlastností je nutnost nějakého způsobu předání uvedených hodnot, protože způsob volání wrapperu nemusíme být vždy schopni ovlivnit. Může tak např. vyvstat nutnost existence více funkcí, každá s jinak nakonfigurovanými parametry. Tyto parametry navíc budou muset být předány nejspíše globálními proměnnými, což není ideální řešení.

Naštěstí má JavaScript i jinou zajímavou vlastnost, která nám popsaný problém pomůže elegantně vyřešit. Jedná se o tzv. *uzávěr* (closure). Tato vlastnost je detailně popsána v [7]

(kapitola *Functions*, část *Nested functions and closures*), takže si jen stručně popíšeme její princip.

Princip spočívá v tom, že pokud nějaká funkce vrátí odkaz na funkci v ní zanořenou, je vrácen tzv. **uzávěr**. Zjednodušeně řečeno, je vrácen odkaz na novou(!) instanci této vnořené funkce, jejíž volné proměnné jsou navázány na hodnoty prostředí, ve kterém tato funkce byla deklarována. To např. znamená, že proměnné nadřazené funkce, ke kterým vnořená funkce přistupuje, budou v nové vrácené instanci mít hodnotu, kterou měly v době běhu nadřazené funkce.

Elegantní řešení s využitím této vlastnosti pak spočívá v tom, že wrapper, který řeší diskutovaný problém, budeme vytvářet dynamicky pomocnou funkcí, která na něj vrátí odkaz. Protože tímto vznikne právě uzávěr, využijeme toho k navázání volných proměnných tohoto wrapperu na potřebné hodnoty (tj. odkaz na metodu a objekt, v jehož kontextu má být volána). Výsledný wrapper pak již popsaným způsobem zajistí správné volání.

Popsané řešení je realizováno v podobě funkce `MkBoundCallFn()`, která se nachází v modulu `js/misc.js`. Tato funkce navíc dokáže vytvořit wrapper schopný předávat parametry, se kterými byl volán, hodnotu `this`, v jejímž kontextu byl volán, a parametry předem definované v době vytváření wrapperu. To vše v libovolné kombinaci i libovolném pořadí. Tyto možnosti jsou detailně popsány v uvedeném souboru před zdrojovým kódem této funkce.

Při využití této funkce se tudíž nepředávají přímo odkazy na metody, ale namísto toho se předává výsledek popsané funkce – odkaz na wrapper, který zajistí volání s vazbu na správný objekt.

5.5 Implementace zkušebního (výukového) režimu

Jak již bylo uvedeno, výsledná aplikace implementuje v tuto chvíli pouze jedinou operaci v nově podporovaném *zkušební režimu*. Popišme konkrétní implementaci tohoto režimu jako takovou i konkrétní realizaci zmíněné operace.

V implementaci BVS je pro *zkušební režim* (ve skutečnosti obecně pro všechny *fázované operace*) implementována kompletní infrastruktura umožňující reakci na kliknutí na uzel BVS a na libovolný odkaz (který se uživateli může zobrazit v rámci pokynů na stavovém panelu). Je navíc rozlišeno kliknutí na běžný uzel stromu a na pseudouzel reprezentující ukazatel na kořenový uzel BVS. Tato infrastruktura je úplně integrována společně s již dříve existující podporou reakce na vypršení nastaveného časového intervalu (mechanismus dostupný prostřednictvím funkce `setTimeout()`).

Identickou podporu zahrnuje také objekt implementující seznam, jehož kód byl odvozen z kódu BVS. Protože zde však zatím nebyly realizovány žádné operace v *demonstračním* ani *zkušebním režimu* (tj. nebyly vytvořeny žádné operace ve fázovaném tvaru), není tato infrastruktura v tuto chvíli využívána.

Zmíněná funkce kliknutí na odkaz může být využita k získání odpovědi uživatele na libovolný dotaz, který není možné rozumně zodpovědět jinými prostředky (např. kliknutím na část zobrazené DS).

Vzhledem k tomu, že nebyla realizována další rozšíření vizualizace BVS, jak je navrhováno v 4.8, je tato funkce v současné době využívána také pro volbu synovského uzlu při vkládání nového uzlu do stromu. Zbývající detaily implementace této operace již odpovídají původnímu návrhu.

5.6 Struktura zdrojového kódu

Zdrojový kód byl v rámci možností poskytovaných implementační platformou modularizován. Hlavním projevem této skutečnosti je jeho rozdělení do několika souborů. Dále bude stručně popsána funkce jednotlivých modulů (souborů).

5.6.1 Modul `index.html`

Tento soubor je z uživatelského pohledu hlavním modulem celé aplikace. Jeho otevřením v prohlížeči se celá aplikace spouští. Tento modul především načítá všechny ostatní moduly a proto obsahuje minimum kódu.

5.6.2 Modul `js/main.js`

Tentokrát se jedná o hlavní modul aplikace z pohledu vnitřní implementace. Jeho hlavním obsahem je základní kód samotné aplikace v podobě objektu `T_Dynamitorial`. Dále obsahuje pomocné funkce používané tímto objektem. (Některé obecnější funkce, které jsou po restrukturalizaci kódu využívány *podpůrnými objekty* DS, byly odtud přemístěny do jiných modulů (zejména do `js/misc.js`). V původním kódu se tento modul jmenoval `js/ui.js`.

5.6.3 Modul `js/dom_utils.js`

Jedná se o modul s pomocnými funkcemi pro pohodlnější manipulaci s DOM objekty. Kromě funkcí pro vytváření DOM elementů vhodně předkonfigurovaných pro použití ve vizualizačních komponentách obsahuje také funkce pro manipulaci s některými běžnými CSS vlastnostmi. Dále obsahuje funkce řešící některé problémy s kompatibilitou, které byly popsány v 5.3. Obsah tohoto modulu se příliš nezměnil, nově však byla přidána již popisovaná funkce `GetContentDocument()`, dále bylo provedeno přeformátování kódu, přejmenování některých identifikátorů a doplnění či zpřesnění komentářů.

5.6.4 Modul `js/misc.js`

Obsahem tohoto modulu jsou některé pomocné funkce, které jsou využívány ostatními moduly. (Většina z nich byla právě z tohoto důvodu přesunuta z modulu `js/ui.js`, ačkoliv JavaScript svou zjednodušenou koncepcí tento postup specifiky nevynucuje.) Dále sem byl přesunut objekt `T_PassByRef` (dříve `T_Reference`), který byl původně v modulu `js/dom_utils.js`, ačkoliv s DOM specificky nesouvisel a nebyl používán jen tímto modulem. Podobně sem byl z `js/trees.js` přesunut objekt `T_ResumeCtx`, který je nyní používán více datovými strukturami. Konečně sem byly přidány i některé nové funkce, především již popsaná a hojně užívaná funkce `MkBoundCallFn()`. Tento modul v původní aplikaci neměl obdoby.

5.6.5 Modul `js/widgets.js`

Tento modul, jak již vyplývá z názvu, implementuje jednotlivé vizualizační komponenty (widgets). Tyto komponenty jsou dále využívány každou DS pro zobrazení jejích součástí. Všechny komponenty jsou z hlediska své vizualizační funkčnosti kompletní. Objekty, které je dále používají, pouze přidávají funkčnost, která zajišťuje jejich další funkce dle potřeb jednotlivých DS. Jedná se o modul, jehož historie sahá až k samotným začátkům vývoje

původní aplikace. V rámci této práce se však dočkal rozšíření o další komponenty. V současnosti tedy modul implementuje následující komponenty:

- `T_WgHorizEdge` – tří-segmentová hrana, která propojí dva body vodorovnou hranou. Při rozdílných souřadnicích y generuje vodorovné segmenty na obou souřadnicích a uprostřed je spojí svislým segmentem.
- `T_WgVertEdge` – třísegmentová hrana, která propojí dva body svislou hranou. Při rozdílných souřadnicích x generuje svislé segmenty na obou souřadnicích a uprostřed je spojí vodorovným segmentem.
- `T_WgTreeEdge` – pětisegmentová hrana, která propojí dva uzly stromu. Z rodičovského uzlu vždy vystupuje směrem dolů a do synovského vždy vstupuje shora. Při rozdílné souřadnici x se chová stejně jako `T_WgVertEdge`. Je-li přípojný bod rodiče níže než přípojný bod syna, stále zachovává popsané směry definovanou minimální délkou svislých krajních segmentů. Na ně poté naváže vodorovnými hranami ke střednímu svislému segmentu, který je spojí.
- `T_WgHorizNode` – obecný horizontální uzel, který může mít několik hodnotových polí. Podporuje zobrazení stromových uzlů libovolné arity a také prvků seznamu (jednosměrného i dvojsměrného).

5.6.6 Modul `js/ds_sup.js`

Zde je definován bázový objekt `T_DSSupObj`, od kterého jsou odvozeny jednotlivé *podpůrné objekty* DS. Tento objekt definuje povinné metody, které musí každý *podpůrný objekt* poskytovat, neboť jsou volány objektem aplikace. Vzhledem k dynamickému typování JavaScriptu však není tato dědičnost pro kompatibilitu rozhraní objektů nutná. Protože většinu (obvykle všechny) z těchto metod budou jednotlivé *podpůrné objekty* chtít předefinovat, není nutné, aby od této třídy dědily. Bázový objekt takto slouží alespoň jako reference povinné minimální implementace. Již hotové *podpůrné objekty* od něj ale dědí. Kromě toho aplikace vytváří instanci tohoto objektu pro každou DS (resp. kartu), pro kterou není definován *podpůrný objekt*. (Tento stav je však smysluplný pouze v případě karty, pro kterou není implementována žádná DS. Bez *podpůrného objektu* by aplikace neměla jak s ní pracovat.) Smyslem této instance je poskytnout prázdnou implementaci pro požadované operace, které aplikace bude volat. Tento *podpůrný objekt* nikdy nevytváří žádnou instanci objektu implementujícího DS. Popsaný modul je zcela nový (stejně jako je nová i celá koncepce *podpůrných objektů*).

5.6.7 Modul `js/dummy_sup.js`

Obsahem je implementace objektu `T_DummySupObj`. Ten je příkladem minimální implementace *podpůrného objektu*, který nepodporuje (a tady ani nevytváří instanci) žádnou DS a pouze řeší vytvoření specifických ovládacích prvků. (Ve skutečnosti generuje nápis 'TODO' po celé ploše, která je pro ovládací prvky vyhrazena.) Činnost tohoto objektu lze pozorovat na kartách DS, které nebyly dosud implementovány. Pokud by pro tyto karty nebyl definován žádný *podpůrný objekt* (tedy by byl implicitně použit výchozí objekt `T_DSSupObj`), pak by tato plocha byla prázdná. Tento modul je také zcela nový.

5.6.8 Modul `js/tree_sup.js`

Tento modul definuje *podpůrný objekt* pro stromy `T_TreeSupObj` (momentálně jej využívá BVS). Funkce *podpůrných objektů* byla popsána v 4.6. Obsahem tohoto modulu je přepracovaná část kódu, který byl původně součástí aplikace v modulu `js/ui.js`. Jinak se ale jedná o nový modul.

5.6.9 Modul `js/tree.js`

Tento modul je věnován implementaci stromů, v tomto okamžiku však implementuje pouze BVS. Kromě DS samotné implementuje objekty pro vizualizaci jednotlivých částí stromů (prozatím zejména binárních), které staví nad vizualizačními komponentami, jak již bylo popsáno. Jedná se o původní modul, jehož některé části byly přepracovány a dále se dočkal některých rozšíření. Repertoár tohoto modulu tvoří následující vizualizační objekty:

- `T_TreeEdge` – běžná hrana libovolného stromu.
- `T_TreeRPNEdge` – hrana, která vede od běžného uzlu do pseudouzlu `T_TreeRPNode`.
- `T_TreeRPNode` – pseudouzel, který reprezentuje ukazatel na kořenový uzel libovolného stromu.
- `T_BinTreeNode` – běžný uzel libovolného binárního stromu.
- `T_BSTree` – implementace samotného BVS.

5.6.10 Modul `js/list_sup.js`

Tento modul je implementací podpůrného objektu pro seznam `T_ListSupObj`. Jedná se o nový modul, který vznikl úpravou `js/tree_sup.js`.

5.6.11 Modul `js/list.js`

Tento modul implementuje samotný seznam a jeho jednotlivé komponenty podobně, jako tomu je v modulu `js/tree.js`, jehož úpravou vznikl. Také tento modul je nový a obsahuje následující objekty:

- `T_LListEdge` – běžná hrana reprezentující ukazatel v libovolném seznamu.
- `T_LListPNEdge` – hrana, která vede od prvku do pseudouzlu `T_LListPNode`.
- `T_LListPNode` – pseudouzel, který reprezentuje ukazatel na první prvek a na aktivní prvek libovolného seznamu.
- `T_SLListNode` – prvek (uzel) jednosměrně vázaného seznamu.
- `T_SLList` – implementace samotného seznamu.

Kapitola 6

Závěr

V rámci této práce jsme vytvořili aplikaci, která může být použita jako výuková pomůcka pro výuku základů programování.

Zajímavou vlastností aplikace je, že běží v prostředí WWW prohlížeče. Tím jsme toto prostředí využili k ne zcela tradičnímu účelu. Lze říci, že jsme takto získali příležitost prověřit možnosti takového použití, a že toto ověření můžeme považovat za úspěšné.

Takto realizovaná aplikace má také výhodu ve svojí portabilitě: protože její požadavky na hostitelské prostředí se omezují na samotný prohlížeč, lze ji provozovat na libovolné platformě, pro kterou máme vyhovující prohlížeč k dispozici.

Díky snaze o implementaci dodržující obecně a dostupné standardy očekáváme možnost provozovat aplikaci i na jiných prohlížečích, než na kterých byla její funkčnost ověřována.

Další výhodou aplikace je její maximální podpora ovládání myší (až na jisté nedostatky popsané v 5.3). I kdyby tato možnost nebyla použita výhradně, lze kombinací s použitím myši aplikaci ovládat efektivněji.

Jak bylo uvedeno, nepodařilo se realizovat všechny požadované či plánované úkoly. I přes ne zcela kompletní realizaci se však domnívám, že byla učiněna a realizována všechna návrhová rozhodnutí zásadnějšího koncepčního charakteru. Správnost prezentovaného návrhu dokazuje i takto omezená implementace.

Při realizaci této práce byly získány mnohé cenné poznatky, z nichž některé již byly detailněji popsány. Za zásadní můžeme považovat úspěšné prověření použitelnosti technologií, které byly užity. V tomto směru je nutné připomenout, že se projevily některé nedostatky existujících implementací. O některých z nich jsme se již zmínili, další poznámky na toto téma pak lze nalézt ve zdrojovém kódu na místech, kterých se toto týká.

Zajímavým získaným poznatkem bylo např. také zjištění, že specifikace DTD (document type definition) v HTML není prohlížeči zcela ignorována, jako tomu bývalo dříve.

V rámci případného budoucího vývoje je jistě vhodné dokončit realizaci dosud neimplementovaných funkcí. Dalšími možnostmi jsou:

- přidat podporu dalších DS, vylepšit již podporované,
- zobrazovat právě prováděný kód algoritmu operace ve známém jazyce (např. C),
- umožnit uživateli volit rychlost běhu operací v *demonstračním režimu*,
- další rozšíření *zkušebního režimu*, např. podpora hodnocení apod.,
- přesuny uzlů (a jiné změny) zobrazovat jako plynulou animaci,
- ...

Příloha A

Uživatelská příručka

Protože aplikace funguje ve webovém prohlížeči, používá komponenty, které by již měly být uživateli známy. Navíc byla snaha o jeho maximálně intuitivní podobu. Z tohoto důvodu popíšeme pouze stručně základní detaily, o kterých lze předpokládat, že je uživatel již bude znát. Více se zaměříme pouze na některé ne zcela zřejmé rysy.

A.1 Popis částí uživatelského rozhraní

Typická podoba GUI je zobrazena na obr. 5.1 a 5.2.

Největší plocha, na níž se zobrazuje DS, je nazývána **pracovní plochou**.

Pod touto plochou se nachází **stavový panel**, kde se zobrazují stavové informace, výsledky operací a prostřednictvím něho aplikace také s uživatelem komunikuje. Uvedené prvky tvoří obsah **karty**, která existuje zvlášť pro každou podporovanou DS.

Na kartě jsou také **ovládací prvky**, které se nacházejí na její pravé straně. Slouží k volbě prováděných operací, řízení jejich průběhu a zadávání jejich argumentů.

Nad viditelnou kartou se nachází **pruh záložek**. Aktivací jednotlivých **záložek** lze mezi jednotlivými kartami přepínat, čímž se provádí volba DS.

A.2 Vstupní pole 'Value'

Toto vstupní pole je doplněno o zajímavou funkci. U některých operací, kde to má smysl (např. vkládání nových uzlů), bude v případě, že uživatel nezadá žádnou hodnotu (tj. toto pole bude v momentě aktivace operace prázdné) automaticky vygenerována náhodná hodnota z intervalu [0; 100). V takovém případě se tato hodnota také zobrazí ve vstupním poli 'Value', ale zobrazí se odlišnou barvou, která se používá pro text neaktivních prvků. Tím je dáno najevo, že hodnota je ve vstupním poli pouze zobrazena a bude při dalším požadavku náhodného čísla přepsána novou hodnotou.

Pokud uživatel tuto hodnotu změní její editací (nebo kliknutím na uzel či prvek DS, které způsobí přepsání této hodnoty hodnotou daného uzlu nebo prvku), změní se její barva na standardní a nebude nadále přepisována. Také pak ani nebude generována náhodná hodnota, ale bude používána ta, která je ve vstupním poli vložena. Pro opětovné povolení generování náhodných čísel stačí obsah vstupního pole vymazat.

Pro úplnost ještě dodejme, že v současné chvíli všechny podporované DS pracují s celočíselnými hodnotami a proto jsou platnými jen hodnoty splňující toto kritérium. Nejsou však omezeny na výše uvedený rozsah, který se používá pro generování náhodných hodnot.

A.3 Stručný přehled ovládání

Tlačítko popsané Wipe *struct* (kde *struct* označuje jméno aktuální DS) slouží ke zrušení obsahu celé aktuálně zvolené DS. Provedení této akce si vyžádá potvrzení uživatele.

Pod tímto tlačítkem se nachází trojice přepínacích tlačítek (radio buttons), která slouží k volbě požadovaného režimu operací (jejich bližší popis viz 4.3 a 4.8). Jednotlivé režimy jsou (zleva) *normální*, *demonstrační* a *zkušební režim*.

Další tlačítka a jiné prvky (kromě dvojice tlačítek zcela dole) slouží k volbě jednotlivých operací ve zvoleném režimu. I zde platí, že operace rušení celé struktury je chráněna dotazem.

Všechny doposud popsané prvky jsou k dispozici pouze v pohotovostním stavu aplikace. Běží-li operace, jsou zakázány až do jejího skončení. Zakázány mohou být také prvky, které odpovídají v daném okamžiku nedostupným operacím (např. nepodporovaná či neimplementovaná varianta).

Dvojice tlačítek úplně dole slouží k ovládání průběhu operace. Na rozdíl od ostatních jsou tato dvě tlačítka proto v pohotovostním stavu zakázána. Povolena jsou pouze během provádění operace (což znamená, že pro operace v *normálním režimu* nemají význam, protože tyto operace probíhají okamžitě; využití má smysl pouze pro *fázované operace*).

Tlačítko -> slouží k okamžitému provedení následujícího kroku (fáze). Pro operace v *demonstračním režimu* to znamená ukončení časového intervalu mezi předchozím a následujícím krokem. V případě operací ve *zkušebním režimu* lze tímto tlačítkem vynechat aktuálně zkoušený krok a přejít k dalšímu.

Tlačítko =>> má funkci předchozího tlačítka prodlouženou až do konce operace. Po jeho stisknutí jsou tedy všechny dosud zbývající kroky provedeny okamžitě.

A.4 Ovládání klávesnicí

Aplikace v maximální míře podporuje ovládání klávesnicí (výjimkou je kliknutí na uzel, které lze zatím provést výhradně pomocí myši nebo jiného ukazovacího zařízení či jeho emulace).

Přístupové klávesy jsou zvýrazněny obvyklým způsobem. Obvykle je pro jejich využití třeba stisknout klávesu Alt a odpovídající přístupovou klávesu. Novější prohlížeče *Mozilla* vyžadují kombinaci kláves Shift+Alt společně s přístupovou klávesou. Bohužel *Internet Explorer* verze 7 si některé přístupové klávesy zásadně „rezervuje“ pro své účely.

Pro aktivaci prvku, na kterém je (např. po předchozím kroku) zaměření (focus), může být podle jeho typu (příp. i podle konkrétního prohlížeče) ještě třeba stisknout další klávesu. Obvykle je to *mezerník*, někdy (např. na odkazy a záložky) poslouží klávesa Enter

Literatura

[1] ECMAScript Language Specification. [online]. Standard ECMA-262; 3rd Edition – December 1999. [cit. 2007-05-20].
URL http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf

[2] Gamma, E.; Helm, R.; Johnson, R.; aj.: *Návrh programů pomocí vzorů: stavební kameny objektově orientovaných programů*. Praha, Česká republika: Grada, 2003, ISBN 80-247-0302-5, 388 s.

[3] Honzík, J. M.: Alogitmy: IAL: Studijní opora. [online]. Dokument je přístupný pouze v rámci FIT VUT v Brně. Verze: 3 – 2007-02-27. [cit. 2007-05-17].
URL http://www.fit.vutbr.cz/study/courses/IAL/private/Opora/Opora-IAL-2007-02-RP-verze-3.pdf

[4] Honzík, J. M.; Hruška, T.; Máčel, M.: *Vyrbané kapitoly z Programovacích technik*. Brno, Česká republika: Vysoké učení technické v Brně, tředí redukované vydání, 1991, ISBN 80-214-0345-4, 218 s.

[5] Kopeček, I.; Kučera, J.: *Programátorské poklesky*. Praha, Česká republika: Mladá fronta, 1989, ISBN 80-204-0068-0, 165 s.

[6] Core JavaScript 1.5 Guide. [online]. Součást wiki Mozilla Developer Center. Last modified 17:14, 1 March 2007. [cit. 2007-05-20].
URL http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Guide

[7] Core JavaScript 1.5 Reference. [online]. Součást wiki Mozilla Developer Center. Last modified 15:40, 22 April 2007. [cit. 2007-05-20].
URL http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Reference

[8] DHTML. [online]. Součást wiki Mozilla Developer Center. Last modified 17:16, 1 March 2007. [cit. 2007-05-19].
URL http://developer.mozilla.org/en/docs/DHTML

[9] HTML:Canvas. [online]. Součást wiki Mozilla Developer Center. Last modified 21:28, 1 January 2007. [cit. 2007-05-20].
URL http://developer.mozilla.org/en/docs/HTML:Canvas

[10] Padrta, D.: Algoritmy a programování. [online]. Aktualizováno: 26. března 2001. [cit. 2007-05-18].
URL http://www.sweb.cz/david.padrta/pascal/alg_prog.html

[11] Tomeš, P.: Překlad slova "tab" v KDE, GNOME, Mozille… [online]. Poslední úprava: 8.7. 14:47. [cit. 2007-05-16].
URL http://www.abclinuxu.cz/blog/quid_novi/2006/7/4/139571

[12] Ajax (programming). [online]. Součást encyklopedie Wikipedia. Last modified 20:33, 19 May 2007. [cit. 2007-05-20].
URL http://en.wikipedia.org/wiki/Ajax_%28programming%29

[13] Canvas (HTML element). [online]. Součást encyklopedie Wikipedia. Last modified 19:24, 19 May 2007. [cit. 2007-05-20].
URL http://en.wikipedia.org/wiki/Canvas_%28HTML_element%29

[14] Dynamic HTML. [online]. Součást encyklopedie Wikipedia. Last modified 19:41, 16 May 2007. [cit. 2007-05-19].
URL http://en.wikipedia.org/wiki/Dynamic_HTML

[15] Linked list. [online]. Součást encyklopedie Wikipedia. Last modified 03:30, 16 May 2007. [cit. 2007-05-17].
URL http://en.wikipedia.org/wiki/Linked_list

[16] Queue (data structure). [online]. Součást encyklopedie Wikipedia. Last modified 01:57, 9 May 2007. [cit. 2007-05-18].
URL http://en.wikipedia.org/wiki/Queue_%28data_structure%29

[17] Scalable Vector Graphics. [online]. Součást encyklopedie Wikipedia. Last modified 15:47, 20 May 2007. [cit. 2007-05-20].
URL http://en.wikipedia.org/wiki/Scalable_Vector_Graphics

[18] Server-side scripting. [online]. Součást encyklopedie Wikipedia. Last modified 13:02, 19 May 2007. [cit. 2007-05-19].
URL http://en.wikipedia.org/wiki/Server-side_scripting

[19] Tree (data structure). [online]. Součást encyklopedie Wikipedia. Last modified 19:13, 15 May 2007. [cit. 2007-05-17].
URL http://en.wikipedia.org/wiki/Tree_%28data_structure%29

[20] Wirth, N.: *Algorithms + Data Structures = Programs*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1978, ISBN 0-13-022418-9.

[21] Cascading Style Sheets, level 2 revision 1: CSS 2.1 Specification. [online]. W3C Working Draft 06 November 2006. [cit. 2007-05-20].
URL http://www.w3.org/TR/CSS21/

[22] Document Object Model (DOM) Level 2 Core Specification. [online]. Version 1.0; W3C Recommendation 13 November, 2000. [cit. 2007-05-20].
URL http://www.w3.org/TR/DOM-Level-2-Core/

[23] Document Object Model (DOM) Level 2 Events Specification. [online]. Version 1.0; W3C Recommendation 13 November, 2000. [cit. 2007-05-20].
URL http://www.w3.org/TR/DOM-Level-2-Events/

[24] Document Object Model (DOM) Level 2 HTML Specification. [online]. Version 1.0; W3C Recommendation 09 January 2003. [cit. 2007-05-20].
URL http://www.w3.org/TR/DOM-Level-2-HTML/

[25] Document Object Model (DOM) Level 2 Style Specification. [online]. Version 1.0; W3C Recommendation 13 November, 2000. [cit. 2007-05-20].
URL http://www.w3.org/TR/DOM-Level-2-Style/

[26] HTML 4.01 Specification. [online]. W3C Recommendation 24 December 1999. [cit. 2007-05-19].
URL http://www.w3.org/TR/html4/