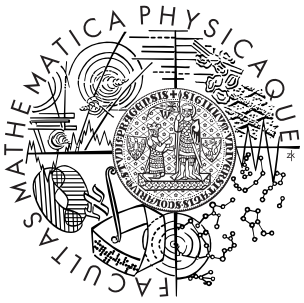
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Lukáš Krížik

Vizualizácia algoritmov

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Jan Kofroň, Ph.D.

Studijní program: programování

2009

Poďakovanie: Vedúcemu práce za trpezlivosť a za to, že mi poskytol možnosť robiť to, čo ma baví.

Prehlasujem, že som svoju bakalársku prácu napísal samostatne a výhradne s použitím citovaných prameňov. Súhlasím so zapožičiavaním práce a jej zverejňovaním.

V Prahe dňa 28.5.2009 Lukáš Krížik

Obsah

Názov práce: Vizualizácia algoritmov
Autor: Lukáš Krížik
Katedra (ústav): Katedra softwarového inženýrství
Vedúci bakalárskej práce: RNDr. Jan Kofroň, Ph.D.
E-mail vedúceho: Jan.Kofron@mff.cuni.cz

Abstrakt: Cieľom práce je vytvoriť aplikáciu pre zobrazovanie algoritmov. Jej úlohou by malo byť vhodné zobrazenie práce algoritmu pri prechode z jedného stavu do druhého. Práca algoritmu by mala byť ľahko pochopiteľná aj pre ľudí, ktorí konkrétny algoritmus nepoznajú a okrem vizualizácie prechodov by mala vedieť aj slovne opísať daný krok algoritmu. Samozrejmosťou by mala byť možnosť pustiť algoritmus na ľubovolné vstupné dáta zadané užívateľom alebo vygenerované samotnou aplikáciou.

Kľúčové slová: algoritmus, dátové štruktúry, vizualizácia

Title: Algorithms' visualization
Author: Lukáš Krížik
Department: Department of Software Engineering
Supervisor: RNDr. Jan Kofroň, Ph.D.
Supervisor's e-mail address: Jan.Kofron@mff.cuni.cz

Abstract: The goal of the project is to create an application for algorithms' visualization. Its objective should be proper visualization of algorithms, including work such as passing from state to state. The algorithm should be easily understandable for people too, who don't know how it works but can operate it using this visualization, literally describing each step. The matter, of course, should be the possibility to run algorithm on user input data or data generated by application.

Keywords: algorithm, data structures, visualization

Kapitola 1

Úvod

Nie nadarmo sa hovorí: „Je lepšie raz vidieť ako tri razy počuť“. Človek získava pomocou zrakového receptora až 90% informácií o vonkajšom prostredí a nepomerne rýchlejšie sa učí[10]. Otázka, prečo vizualizovať rôzne postupy, ktorými jednotlivé algoritmy sú, je preto ľahko zodpovedateľná.

Pri výučbe algoritmov, dátových štruktúr a algoritmov s nimi spojených, nie jeden učiteľ ocení vizualizačné pomôcky. Výuka s nimi je názornejšia a prebieha rýchlejšie.

1.1 Úvod do problematiky vizualizácie

Na vizualizáciu algoritmov sa ako na väčšinu problémov dá pozerať z rôznych uhlov. Ťažko povedať, ktorý je ten správny a pre rôznych ľudí môžu byť tie správne úplne odlišné. V tomto projekte je snaha zozbraziť algoritmus, tak aby sa moc neodlišoval od jeho implementácie a implementácie štruktúr nad ktorými sa algoritmus používa a aby bol zároveň čo najlepšie pochopiteľný.

Príkladom môžu byť triediace algoritmy. Práve u nich sa na prvky môžete pozerať ako na body v priestore a na ich hodnoty ako na zmenu ich pozícií. Na druhú stranu, keď sa niečo triedi, tak sa triedi zoznam prvkov a ten si drvivá väčšina ľudí predstaví lineárne uložený v priestore. Hodnota prvku môže byť potom zobrazená numericky a už aj pri letmom pohľade na rozumne dlhý zoznam možno vidieť, ktorý prvok je väčší a ktorý menší.

1.2 Cieľ projektu

Ako bolo spomenuté v úvode, cieľom projektu je vytvoriť aplikáciu, ktorá bude slúžiť ako pomôcka pri výuke. Malo by sa jednať hlavne o samostatnú výuku, ale rovnako použiteľná by mala byť aj pri vyučovaní. Aplikácia bude ľahko rozšíriteľná a nebude úzko zameraná na určitú skupinu algoritmov. Hlavná časť aplikácie bude požadovať od modulov iba dodržanie pevne dohodnutého rozhrania, ale spôsob zobrazenia a dokonca aj prostriedky pre zobrazenie algoritmu nechá úplne na autoroch modulov.

Implementované boli algoritmy, ktoré by mali byť minimom znalostí každého programátora možno až na AVL a červeno-čierne stromy, ktoré ale aspoň ukazujú, že aj binárne vyhľadávacie stromy sú s menšou úpravou použiteľné v praxi. Ku každému algoritmu, až na tie úplne triviálne, je možnosť nahliadnuť na pseudokód, kde sú vyznačené riadky, ktoré budú vykonané v ďalšom kroku. Pseudokód slúži ako slovný doprovod k algoritmu. Ďalšie dialógové okienko s viac zrozumiteľným slovným doprovodom by aplikáciu len viac zneprehľadnilo. Pseudokód je kompaktný a neformálny vysokoúrovňový opis programovania algoritmu, ktorý spĺňa štruktúru programovacieho jazyka, ale je určený pre čítanie človekom a nie strojom[11]. Ako syntax kódu som vybral tú, ktorá je použitá v jazyku C. Je dostatočne prehľadná a ako dôkaz môže poslúžiť to, že mnoho jazykov, ktoré vznikli neskôr, prebralo práve túto syntax. Ako niektoré kľúčové slová som však použil slová z jazyka Pascal, ktoré lepšie opisujú konkrétny element. Napríklad u dátových typov je slovo *real* presnejšie na opis reálneho čísla oproti tomu z jazyka C, kde sa na to používajú identifikátory *float* a *double*. Podobne je to aj s inými typmi, zápisom hlavičiek procedúr a funkcií a niektorými operátormi. Snaha bola vybrať z týchto dvoch jazykov to najlepšie.

Kapitola 2

Analýza

Dôvodov pre vizualizáciu môže byť viacero. Ten svoj som uviedol už v úvode a samozrejme nie je jediný. Vizualizácia môže pomôcť napríklad pri analýze, verifikácií alebo aj optimalizácií. Ťažko však urobiť aplikáciu tak, aby zahrňovala všetky možné pohľady. Ako analýza, verifikácia, tak aj optimalizácia bude klásť dôraz na rýchlosť výpočtu a prehľadnosť zobrazenia stavu aj pre veľké množstvo prvkov. Na druhú stranu pri štúdiu prípadne výuke nie je nutné zobrazovať obrovské množstvo dát, ale len niektoré špecifické situácie, ktoré nastávajú už pri pomerne malom množstve prvkov. Dôležité bude ale kvalitne zobraziť prechod medzi dvoma stavmi.

Snaha bola nezameriavať sa na jeden konkrétny algoritmus alebo skupinu algoritmov, ale vytvoriť akúsi platformu pre zobrazovanie algoritmov, ktorá bude ľahko rozšíriteľná a ponechá autorom jednotlivých implementácií čo najväčšiu voľnosť. Autor sa môže preto rozhodnúť akým spôsobom algoritmus zobrazí a akú mu dá funkčnosť.

Pri každom projekte je nutné si najprv rozmyslieť, čo vlastne ideme robiť, pre koho, aký je cieľ projektu a nakoniec samozrejme ako to všetko budeme robiť. Je nutné zachovať postup, lebo odpovede na neskoršie otázky závisia na odpovediach na tie predchádzajúce. V tejto kapitole odpoviem aj na tú implementačnú otázku.

2.1 Požiadavky

Samozrejmosťou by mala byť možnosť spustenia algoritmu na ľubovoľné vstupné dáta, ktoré budú buď zadané užívateľom alebo nejakým spôsobom vygenerované aplikáciou. V opačnom prípade by sa jednalo o akúsi formu

prezentácie, pre ktorú existujú iné a lepšie prostriedky ako ju naprogramovať. Ďalej treba užívateľovi umožniť zastaviť beh v rozumnom počte krokov, aby mal možnosť premyslieť si v akom stave sa algoritmus nachádza a do akého stavu by sa mal dostať pre overenie svojich znalostí. Zastavenie behu sa dá využiť aj pri vysvetľovaní fungovania algoritmu na slovný doprovod vyučujúceho, ktorý môže počas prestávky medzi dvoma krokmi objasniť prečo algoritmus vykonal to, čo vykonal. Ako „slovný doprovod“ môže poslúžiť tiež pseudokód, ktorý by mal byť súčasťou všetkých modulov, ktoré takýto doprovod potrebujú.

Tou najdôležitejšou časťou je ale prechod z jedného stavu do druhého. Pokiaľ by sa zanedbala táto časť, mohla by byť vizualizácia stavu akokoľvek dobrá, projekt by pravdepodobne nesplnil svoj účel, ktorým je naučiť fungovanie jednotlivých algoritmov. Prechod musí byť názorný. Pri zmene stavu môže dôjsť ku zmene pozícií mnohých prvkov a pre užívateľa môže byť náročné všimnúť si všetky zmeny. Preto je žiadané aby existovala možnosť zobraziť prechod v užívateľom nastaviteľnom čase pomocou animácie. V prípade snahy dostať sa čo najrýchlejšie do konkrétneho stavu môžu byť animácie nežiadúce a mali by sa dať vypnúť.

2.2 Návrh

Kvôli rozšíriteľnosti je aplikácia rozdelená na hlavnú časť a moduly, ktoré implementujú jednotlivé algoritmy. Hlavná časť moduly spravuje[1] a spúšťa vizualizácie. Informácie o moduloch sú uložené v stromovej štruktúre, ktorej vnútorné vrcholy predstavujú triedy algoritmov a listy jednotlivé moduly. Autorom modulov je ponechaná pri implementácií veľká voľnosť a stačí aby modul obsahoval svoje meno, opis a implementoval funkcie pre vytvorenie dialógu a uvoľnenie modulu. Takéto malé rozhranie dovoľuje autorom implementovať prakticky čokoľvek a nemusí sa dokonca jednať ani len o vizualizáciu nejakého algoritmu.

Veľká voľnosť pri tvorbe modulov môže mať aj svoje negatíva. Je žiadúce aby rozhrania pre algoritmy spadajúce do tej istej triedy boli veľmi podobné, ak nie úplne totožné. V opačnom prípade sa stane aplikácia neprehľadnou. Jedná sa ako o ovládanie, tak aj o spôsob vizualizácie. Napríklad u triediacich algoritmov by malo byť základné ovládanie všade rovnaké[2]. Pri tvorbe

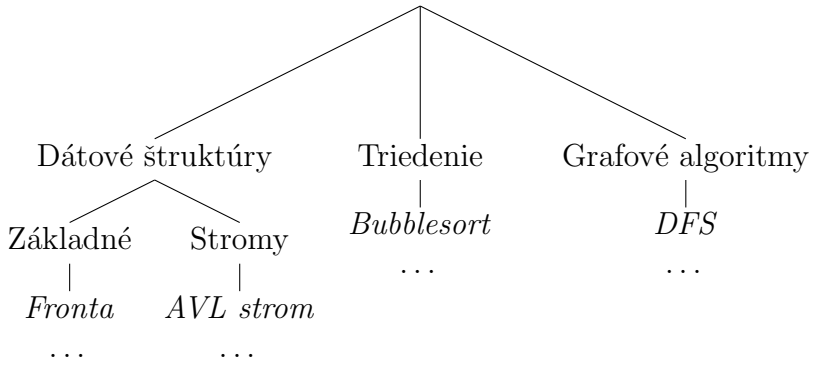
[1] Kategorizuje, načítava a uvoľňuje moduly z pamäte.

[2] Triedime zoznam prvkov. Meníme len postup, ktorým tento zoznam triedime.

modulov som sa zameral na zachovanie konzistencie rozhrania u algoritmov spadajúcich do tej istej triedy aj napriek tomu, že to znamená zväčšenie ich pamäťových nárokov[3]. Na druhú stranu nám to umožní neskoršiu úpravu vizualizácie a zmenu funkčnosti niektorých algoritmov[4].

2.3 Správa algoritmov

Projekt sa nezameriava na jeden algoritmus alebo skupinu algoritmov, ale poskytuje možnosť rozšírenia o ľubovoľný ďalší a jeho zaradenie medzi ostatné. Algoritmy môžu byť zatriedené podľa kategórií do stromovej štruktúry (obrázok 2.1). Samotné zatriedenie je však na konkrétnom užívateľovi, ktorý rozhoduje kam ktorý zaradí. Vzhľadom na to, že samotná implementácia vizualizácie spotrebuje miesto v pamäti, je možnosť modul do pamäte načítať, prípadne z nej uvoľniť.



Obrázok 2.1: Príklad zaradenia algoritmov do stromovej štruktúry.

2.4 Užívateľské rozhranie

Pri implementácií rozhrania na ovládanie je nutné nájsť vhodné vyváženie medzi jeho prehľadnosťou a pohodlnosťou, inak povedané robustnosťou. Ja

[3]Každý modul implementuje tú istú vec a v prípade, že je v pamäti načítaných viac podobných modulov, nachádza sa v pamäti aj jej viac kópií.

[4]Najlepším príkladom môže byť pridanie výberu metódy voľby pivota v quicksorte.

som sa snažil vytvoriť minimálne rozhranie čo sa veľkosti týka. Všetko potrebné na úpravu dátovej štruktúry nad ktorou algoritmus beží a ovládania samotného algoritmu je v hlavnom dialógu modulu. Hlavný dialóg taktiež poskytuje možnosť zobraziť alebo skryť ďalšie dialógy, ktoré užívateľ nemusí považovať za potrebné vidieť.

Udržať čo najväčšiu konzistenciu rozhrania je veľmi dôležité. Pre algoritmy patriace do tej istej triedy by sa malo rozhranie líšiť minimálne. Algoritmy budú s najväčšou pravdepodobnosťou pracovať nad rovnakou dátovou štruktúrou a aj keď postup je rozdielny, ovládanie algoritmu by malo byť rovnaké. Napríklad u triediacich algoritmov sa predpokladá, že triedený bude zoznam a teda generovanie zoznamu a jeho úprava bude pre všetky algoritmy rovnaká.

2.5 Platforma a jazyk

V súčastnosti už nie je problém urobiť multiplatformovú aplikáciu. Multiplatformové programovanie aplikácií s GUI však stále nemá požadovanú podporu, lebo je nutné používať knižnice implementované na viacerých platformách[5], ktoré sa výzorovo líšia od aplikácií programovaných natívne pre danú platformu a tím strácajú na kvalite. Na druhú stranu už ťažko povedať, že by bola vo svete nejaká platforma absolútne dominantná a výberom jednej by sa mohlo zdať, že sa automaticky „ublíži“ užívateľom ostatných. Nemusí to byť úplne pravda. Virtualizácia ako obor informatiky zaznamenáva v poslednom období pomerne veľký pokrok. Jej podpora neustále rastie a nezdá sa, že by sa tento vývoj spomaľoval. Ja som sa rozhodol pre kvalitu a spolieham sa na pokrok vo virtualizácií. Ako platformu je teda dobré si zvoliť ten zatiaľ najpoužívanejší, aj keď už nie tak dominantný systém. Microsoft Windows je stále dosť rozšírený na to, aby mal každý možnosť dostať sa k počítaču s týmto operačným systémom.

Obhajoba výberu jazyka bude už o dosť ťažšia. Každý jazyk má svoje pre a proti. Pri výbere jazyka som sa zameral hlavne na moju znalosť tohto jazyka. Možno existujú jazyky, ktoré by sa hodili k takémuto projektu viac ako C++[9], ale naučiť sa ľubovoľný jazyk v aspoň trochu pokročilej forme trvá dlho a programovať aplikáciu v jazyku len so základnými znalosťami by nebola veľmi šťastná voľba. Na druhú stranu C++[9] by bol horúci kandidát na podobný projekt. Jeho podpora v operačnom systéme Microsoft Windows

[5]Pre C++ je to napríklad knižnica SDL, pre programovací jazyk Java zasa Swing.

síce klesá, ale stále je dosť veľká na to, aby sa v ňom dali robiť kvalitné aplikácie. Spolu s jazykom je pomerne pevne zviazaná knižnica MFC[7], ktorá zaobaľuje veľkú časť rozhrania operačného systému programovaného v jazyku C do objektov v jazyku C++.

2.6 Grafické rozhranie

Pod systémom Microsoft Windows je na výber hneď niekoľko spôsobov ako vykresľovať na obrazovku. Samotný systém poskytuje rozhranie nazývané GDI[6]. Toto rozhranie však nie je navrhnuté na vykresľovanie zložitejších štruktúr ako sú 3D objekty a má malú vykresľovaciu rýchlosť. Aplikácie náročnejšie na vykresľovanie používajú rozhranie DirectX alebo OpenGL. Tieto rozhrania sú priamo podporované hardvérom grafických kariet, majú veľmi vysokú rýchlosť vykresľovania, podporujú vykresľovanie 3D objektov a preto sú používané v najmodernejších hrách.

DirectX je produkt firmy Microsoft a tak je jeho podpora v operačnom systéme Windows od tej istej firmy veľmi vysoká. Jeho podpora v iných operačných systémoch je naopak minimálna až žiadna. OpenGL bolo vyvinuté firmou Silicon Graphics Inc. (SGI) ako multiplatformové rozhranie pre vykresľovanie 2D a 3D grafiky[11]. Vzhľadom na to, že som v sekcií o výbere platformy spomenul, že som sa striktne zameral na operačný systém Microsoft Windows sa dá DirectX pokladať za ideálnu voľbu pre podobný projekt. Musím však opäť použiť podobný argument ako pri výbere programovacieho jazyka a to je znalosť rozhrania OpenGL, ktorú mám podstatne väčšiu ako znalosť DirectX. Treba však spomenúť, že pri podobnom projekte je OpenGL aj pod systémom Microsoft Windows úplne postačujúce čo sa týka výkonu a funkčnosti[7].

2.7 Moduly

Aplikácia je navrhnutá tak, aby sa dal do nej pridať ľubovoľný algoritmus. Ten je vo všeobecnosti len postupnosťou krokov, ktoré vedú k vyriešeniu

[6]GDI (Graphics Device Interface) je súčasťou aplikačného rozhrania systému Microsoft Windows a jadra operačného systému zodpovedného za reprezentáciu grafických objektov a ich prenos na výstupné zariadenia ako napríklad obrazovku alebo tlačiareň[11].

[7]Zvláda rýchle vykresľovanie 3D grafiky pre animácie.

problému. Je nutné implementovať aplikáciu tak, aby sa do nej dala pridať ľubovoľná takáto postupnosť. Úplne prvý nápad je mať túto postupnosť uloženú v súbore a spolu s ňou aj to akým spôsobom má byť vizualizovaná. Samozrejme, že celý postup by musel byť písaný v nejakom jazyku, ktorému by aplikácia rozumela a vedela si ho preložiť. Toto všetko však nie je nutné implementovať vzhľadom na to, že podobné mechanizmy už existujú. Jazyky na opis postupnosti krokov pri programovaní[8] a ich prekladače do jazyku, ktorému rozumejú stroje už existujú a dajú sa použiť aj v tejto situácií. Firma Microsoft vyvinula pre operačné systémy Microsoft Windows a OS/2 zdieľanú knižnicu DLL (Dynamic-link library), ktorá môže byť za behu pripojená k aplikácií, môže implementovať ľubovoľný algoritmus a preto je tou najlepšou implementačnou možnosťou pre tvorbu modulov do aplikácie.

[8]Inými slovami programovacie jazyky.

Kapitola 3

Algoritmy a dátové štruktúry

Vizualizoval som algoritmy a dátové štruktúry, ktoré by mali byť známe každému programátorovi. U dátových štruktúr sú to základné dve: fronta a zásobník. Ďalej sú to binárne vyhľadávacie stromy a aby mal užívateľ možnosť vidieť, že aj tie sú použiteľné, tak ich vyváženejšie verzie: AVL a červeno-čierne stromy. Triedenie a prehľadávanie grafov považujem taktiež za všeobecné vzdelanie každého programátora.

3.1 Základné dátové štruktúry

Medzi základné dátové štruktúry patrí práve fronta a zásobník. Obidve sú zoznamom a líšia sa len spôsobom pridávania a odoberania prvku z tohto zoznamu. Práve tieto dve štruktúry sa často používajú ako podklad u iných algoritmov, ktoré si potrebujú ukladať dáta a podľa výberu, ktorú použijú, sa líši ich chovanie.

3.2 Fronta

Fronta je zoznam, ktorý sa riadi princípom FIFO (*First In, First Out, slovensky Prvý dnu, prvý von*). Jedná sa o abstraktný dátový typ a nie je definovaný spôsob jeho implementácie. Na fronte existujú dve základné operácie: *pridaj (push)* a *odober (pop)* prvok. Samozrejme nie je vylúčené implementovať aj iné operácie ako napríklad *nájdi (find)* alebo *zničˇ (destroy)*.

Pridávaný prvok sa pridá na začiatok zoznamu. V prípade, že implementujeme frontu ako spojový zoznam, je táto operácia triviálna a vykonaná

v konštantnom čase. V prípade, že použijeme statické pole, môže sa cena pridávania prvku zväčšiť až na dĺžku fronty[1].

Prvky odoberáme z konca zoznamu. Pri spojovom zozname si môžeme pamätať ukazovateľ na koniec zoznamu a tým zjednodušiť časovú zložitosť operácie na konštantnú. V prípade, že si tento ukazovateľ pamätať nebudeme, časová zložitosť bude lineárna. Pri statickom poli je táto zložitosť konštantná.

3.3 Zásobník

Zásobník je zoznam, ktorý sa riadi princípom LIFO (*Last In, First Out, slovensky Posledný dnu, prvý von*). Takisto ako u fronty sa jedná o abstraktný dátový typ a nie je presne definovaný spôsob jeho implementácie. Zásobník, podobne ako fronta, poskytuje dve základné operácie: *pridaj (push)* a *odober (pop)* prvok.

Prvky pridávame na začiatok zoznamu. V prípade použitia spojového zoznamu má operácia konštantnú časovú zložitosť. Pri použití statického poľa môže cena operácie narásť na veľkosť zásobníku v prípade, že jeho veľkosť po pridaní prvku prekročí veľkosť naalokovaného miesta.

Odoberáme prvý prvok zoznamu[2] a v oboch spomenutých prípadoch implementácie bude mať táto operácia konštantnú časovú zložitosť.

Zatiaľčo u fronty je dobré si pamätať začiatok aj koniec zoznamu[3], u zásobníku si stačí pamätať jeho vrch.

3.4 Binárne stromy

O niečo zložitejšími štruktúrami ako prvé dve uvedené sú binárne stromy. Všeobecne sa jedná o stromovitú štruktúru, v ktorej má každý vrchol 0 až 2 potomkov. Vrchol so žiadnym potomkom nazývame list, vrchol s jedným alebo dvoma potomkami nazývame vnútorný a vrchol, ktorý nemá predka koreň.

[1] V prípade, že by pridávaný prvok zasahoval mimo pole.

[2] Prvok, ktorý tam bol vložený ako posledný.

[3] Buď ukazovateľ alebo index.

3.5 Binárny vyhľadávací strom

Jedná sa o jednoduché binárne stromy, v ktorých platí pravidlo, že vrcholy v ľavom podstrome vnútorného vrcholu majú menšiu hodnotu ako je hodnota vrcholu a prvky v pravom podstrome majú naopak hodnotu väčšiu. Nad štruktúrou sa používajú operácie *nájdi (find)*, *pridaj (insert)* a *odober (remove)*. V prípade, že je strom pomerne vyvážený[4], je zložitosť všetkých operácií približne logaritmická. Binárne stromy však môžu veľmi ľahko degenerovať a v prípade, že do takejto štruktúry vkladáme usporiadanú postupnosť pomocou operácie *pridaj*, strom degeneruje na lineárny spojový zoznam.

Operácia *nájdi* je veľmi jednoduchá. Pri prechode stromu začíname v jeho koreni. Podľa hodnoty vo vrchole sa rozhodneme do ktorého podstromu budeme pokračovať. V každom vrchole máme tri možnosti. Buď sme hodnotu našli, hodnota vo vrchole je väčšia alebo menšia ako hodnota, ktorú hľadáme. V druhom a treťom prípade pokračujeme prehľadávaním podstromu daného vrcholu, ak je to možné. Ak taký podstrom neexistuje, strom hodnotu neobsahuje.

Operácia *pridaj* funguje tak, že sa najprv pokúsime danú hodnotu v strome nájsť a ak sa nám to nepodarilo, tak ju vložíme ako list tam, kam patrí.

Odstraňovanie hodnoty zo stromu je už o niečo zložitejšie. Opäť musíme hodnotu, ktorú chceme odstrániť, nájsť a podľa počtu potomkov vrcholu, ktorý danú hodnotu obsahuje, sa rozhodneme, čo budeme robiť ďalej. Ak je vrchol list alebo má len jeden podstrom, odstránime vrchol triviálne. Ak má vrchol obidvoch potomkov, musíme nájsť buď najvyššiu hodnotu v ľavom podstrome alebo najnižšiu hodnotu v pravom podstrome. Hodnoty týchto dvoch vrcholov potom zameníme a odstránime hodnotu z nanovo nájdeného vrcholu. Treba si uvedomiť, že tento vrchol má nanajvýš jedného potomka.

Ako som už spomínal, binárne vyhľadávacie stromy pomerne ľahko degenerujú a preto sa namiesto nich používajú ich vyváženejšie verzie, ktoré používajú operáciu *rotuj (rotate)* na vyvažovanie.

Pri implementácií bola operácia *pridaj* rozdelená na nasledujúce stavy v ktorých je možné algoritmus zastaviť:

1. *Testovanie prázdnosti stromu* $\rightsquigarrow$ 2, 3 - Ak je strom prázdny, pokračujeme do stavu 2, inak nastavíme koreň ako prehľadávaný vrchol a pokračujeme v stave 3.

[4]Hĺbky podstromov každého vrcholu sa moc nelíšia.

2. *Strom je prázdny* $\rightsquigarrow$ koniec - Vkladanú hodnotu uložíme ako koreň stromu a algoritmus ukončíme.

3. *Porovnanie vkladanej hodnoty s hodnotou vo vrchole* $\rightsquigarrow$ 4, 5, 6 - V prípade, že má prehľadávaný vrchol rovnakú hodnotu ako vkladaná hodnota, pokračujeme do stavu 6. V prípade, že je hodnota vrcholu väčšia, pokračujeme do stavu 4 so zmenou prehľadávaného vrcholu na jeho ľavého potomka. Ak je hodnota menšia, řiešime symetricky so zmenou do stavu 5.

4. *Hľadanie v ľavom podstrome* $\rightsquigarrow$ 3, 7 - Ak prehľadávaný vrchol existuje, zmeníme stav na 3. V opačnom případe zmeníme stav na 7.

5. *Hľadanie v pravom podstrome* $\rightsquigarrow$ 3, 7 - Ak prehľadávaný vrchol existuje, zmeníme stav na 3. V opačnom případe zmeníme stav na 7.

6. *Hodnota je nájdená* $\rightsquigarrow$ koniec - Ukončíme algoritmus.

7. *Vkladanie* $\rightsquigarrow$ 8, 9 - Zmeníme stav podľa toho, či vkladáme hodnotu do ľavého alebo pravého potomka.

8. *Vkladanie do ľavého podstromu* $\rightsquigarrow$ koniec - Uložíme hodnotu a končíme.

9. *Vkladanie do pravého podstromu* $\rightsquigarrow$ koniec - Uložíme hodnotu a končíme.

Operácia *odober* bola rozdelená na nasledujúce stavy, v ktorých je možné algoritmus zastaviť:

1. *Nájdi* $\rightsquigarrow$ 2, 3, 4, koniec - Podľa hodnoty prehľadávaného vrcholu rozhodneme, v ktorom stave budeme pokračovať. Ak nie je kam pokračovať, algoritmus ukončíme.

2. *Nájdi v ľavom podstrome* $\rightsquigarrow 1$[4].

3. *Nájdi v pravom podstrome* $\rightsquigarrow 1$[4].

4. *Nájdená* $\rightsquigarrow 5$[4].

[4]Nastáva len zmena stavu. Stav bol pridaný kvôli prehľadnosti a animáciám.

5. *Odstránenie* $\rightsquigarrow$ 6, 7, 8 - Ak sa jedná o triviálne odstránenie vrcholu, pokračujeme stavmi 6 a 7, inak pokračujeme stavom 8.

6. *Odstráň z ľavého podstromu* $\rightsquigarrow$ koniec - Triviálne odstránime vrchol a končíme.

7. *Odstráň z pravého podstromu* $\rightsquigarrow$ koniec - Triviálne odstránime vrchol a končíme.

8. *Inicializácia hľadania náhradného vrcholu* $\rightsquigarrow$ 9 - Hľadajme najmenší vrchol v pravom podstrome tak, že náhradníka nastavíme ako pravého potomka nájdeného vrcholu.

9. *Hľadaj náhradný vrchol* $\rightsquigarrow$ 9, 10 - Ak existuje ľavý potomok náhradníka, nemeníme stav a nastavíme náhradníka na jeho ľavého potomka, inak zmeníme stav na 10.

10. *Vymeň hodnoty* $\rightsquigarrow$ 11 - Vymeníme hodnoty náhradníka a nájdeného vrcholu.

11. *Odstráň vrchol* $\rightsquigarrow$ koniec - Triviálne odstránime náhradníka a končíme.

3.6 AVL strom

AVL strom je samovyvažovacia štruktúra pomenovaná po jej autoroch *G.M. Adelson-Velskii* a *E.M. Landis*[11]. Platia v ňom všetky pravidlá, ktoré platili aj pre binárne vyhľadávacie stromy a ešte niektoré ďalšie. V AVL stromoch sa hĺbka ľavého a pravého podstromu každého vrcholu líši nanajvýš o jedna, čím sa zabezpečí približne logaritmická hĺbka stromu. Pri pridávaní a odoberaní vrcholov sa môže táto vlastnosť narušiť, čo sa rieši pomocou operácie *rotuj (rotate)*.

Za povšimnutie v skutočnosti stoja len dva prípady. Rozoberiem len ich ľavú verziu[5].

Hĺbka ľavého podstromu vrcholu je o 2 väčšia ako hĺbka jeho pravého podstromu a hĺbka ľavého podstromu jeho ľavého potomka je o 1 menšia ako hĺbka jeho pravého podstromu. V tomto prípade rotujeme okolo jeho ľavého potomka, pravého potomka tohto potomka (rotujeme teda doľava) a dostaneme sa do druhej situácie.

[5]Pravá verzia sa robí symetricky.

Hĺbka ľavého podstromu vrcholu je o 2 väčšia ako hĺbka jeho pravého podstromu a hĺbka ľavého podstromu jeho ľavého potomka je o 1 väčšia ako hĺbka jeho pravého podstromu. V tomto prípade rotujeme okolo vrcholu jeho ľavého potomka (rotujeme teda doprava). Po rotácií bude podstrom vyvážený.

Po rotácií sa môže zmeniť hĺbka celého podstromu a je teda nutné prípadnú chybu distribuovať smerom nahor.

Operácie *pridaj* a *odober* majú rovnaké delenie na stavy ako rovnomenné operácie na binárnom vyhľadávacom strome. Každá má však pridaný jeden stav na vyvažovanie, ktorý je iba volaním vyvažovacej funkcie. Operácia *pridaj* má však inú vyvažovaciu funkciu ako *odober*.

3.7 Červeno-čierny strom

Červeno-čierny strom je samovyvažujúca štruktúra, pre ktorú platia rovnaké pravidlá ako pre binárny vyhľadávací strom a ešte:

- Každý vrchol je sfarbený buď na čierno alebo na červeno.
- Koreň je vždy čierny.
- Listy sú tiež čierne a neobsahujú žiadnu hodnotu.
- Každý červený vrchol má obidvoch potomkov čiernych.
- Počet čiernych vrcholov na ceste z koreňa do listu je pre všetky listy rovnaký.

Červeno-čierne stromy sú vyvažované opäť pomocou rotácií. Avšak situácií[11], ktoré stoja za povšimnutie, je omnoho viac ako pri AVL stromoch.

Operácie *pridaj* a *odober* majú rovnaké delenie na stavy ako rovnomenné operácie na binárnom vyhľadávacom strome. Každá má však pridaný jeden stav na vyvažovanie, ktorý je iba volaním vyvažovacej funkcie. Operácia *pridaj* má však inú vyvažovaciu funkciu ako *odober*.

3.8 Triediace algoritmy

Triediace algoritmy asi nikto z programátorov už nikdy a nikde implementovať nebude, pretože sú implementované hádam vo všetkých jazykoch,

všetkými možnými spôsobmi a nad všetkými možnými štruktúrami. Ja som sa ich ale rozhodol zaradiť medzi algoritmy, ktoré by mal poznať každý programátor, už len preto, aby si osvojil rôzne techniky v nich používané ako napríklad *rozdeľ a panuj* u triedenia zlievaním.

3.9 Bublinkové triedenie (Bubblesort)

Tento algoritmus bol uvedený ako všeobecne zlý a nemal by byť používaný. Na druhú stranu, jeho zložitosť v najhoršom prípade (klesajúca postupnosť) je rovnaká ako zložitosť najpoužívanejšieho algoritmu na triedenie, quicksortu. V najhoršom prípade je nutné vykonať kvadratický počet operácií na utriedenie zoznamu. Bublinkové triedenie prechádza zoznamom od začiatku do konca a ak nájde dva susedné prvky, ktoré sú zle usporiadané, vymení ich a pokračuje ďalej. Zoznam prechádza dovtedy, kým sa mu pri prechode podarí prehodiť aspoň jednu dvojicu prvkov.

Ak sa pozriete na prácu algoritmu, ľahko pochopíte odkiaľ dostal názov. Ak by sa náhodou najmenší prvok nachádzal úplne posledný v zozname, pri každom prechode zoznamom by sa posunul o jednu pozíciu smerom na prvé miesto. Aby prvok "prebublal" až na prvú pozíciu, je treba vykonať n prechodov zoznamom.

Bublinkové triedenie je rozdelené na nasledujúce stavy:

1. *Inicializácia* $\rightsquigarrow 2$[5].
2. *Inicializácia cyklu* $\rightsquigarrow 3$[5].
3. *Otázka* $\rightsquigarrow 4$[5].
4. *Výsledok* $\rightsquigarrow$ 5, 6 - Ak je hodnota pravého prvku menšia ako ľavého, zmeníme stav na 5, inak na 6.
5. *Prehodenie* $\rightsquigarrow$ 6 - Vymeníme hodnoty susedných prvkov a zmeníme stav na 6.
6. *Zmena indexu* $\rightsquigarrow$ 2, koniec - Ak sme prehodili aspoň jednu dvojicu prvkov, zmeníme stav na 2, inak končíme.

[5]Nastáva len zmena stavu. Stav bol pridaný kvôli prehľadnosti a animáciám.

3.10 Quicksort

Asi najpoužívanejší algoritmus na triedenie. Jeho zložitosť v najhoršom prípade je kvadratická, avšak v priemernom prípade je logaritmická. Algoritmus sa dá jednoducho opísať v niekoľkých krokoch:

1. Nájdi pivota.
2. Prvky menšie ako pivot presuň naľavo.
3. Prvky väčšie ako pivot presuň napravo.
4. Pretrieď prvky menšie ako pivot.
5. Pretrieď prvky väčšie ako pivot.

Rýchlosť algoritmu súvisí aj so správnou voľbou pivota. Pivot by mal rozdeliť prvky zoznamu do dvoch približne rovnako veľkých skupín. Voľba pivota nie je úplne jednoduchý problém, avšak v priemernom prípade postačí zvoliť niečo ako aritmetický priemer.

Quicksort je rozdelený na nasledujúce stavy:

1. *Inicializácia* $\rightsquigarrow$ 2 - Nastáva len zmena stavu.
2. *Utrieď zoznam* $\rightsquigarrow$ 7 - Pokúsime sa utriediť aktuálny zoznam.
3. *Vyber pivota* $\rightsquigarrow$ 4 - Nájdeme pivota v aktuálnom zozname.
4. *Rozdeľ prvky* $\rightsquigarrow$ 5 - Menšie prvky ako pivot presunieme naľavo, väčšie napravo.
5. *Utrieď menšie* $\rightsquigarrow$ 7 - Pokúsime sa utriediť menšie prvky.
6. *Utrieď väčšie* $\rightsquigarrow$ 7 - Pokúsime sa utriediť väčšie prvky.
7. *Zastavenie rekurzie* $\rightsquigarrow$ 3, 6, koniec - Nájdeme časť zoznamu, ktorá má byť triedená ako nasledujúca. Ak taká neexistuje, končíme.

3.11 Halda (Heapsort)

Aj napriek tomu, že sa halda implementuje ako pole, treba na ňu pozerať ako na binárny strom, v ktorom platí:

1. Hodnota každého vrcholu je menšia ako hodnota jeho potomkov.
2. Halda je strom, v ktorom je každý riadok úplne zaplnený, až na posledný, ktorý ja zaplnený zľava.

Na to aby sa dala halda použiť na triedenie stačí, aby podporovala operácie *pridaj (insert)*, *minimum* a *odstráň minimum (remove minimum)*. Za použitia týchto operácií sa potom na prvý prechod zoznamom vytvorí halda (*pridaj*) a na druhý prechod sa halda zničí (*minimum, odstráň minimum*). Každá operácia má logaritmickú zložitosť a tak ľahko vidno, že výsledná zložitosť je $n \cdot log(n)$.

Pri vkladaní prvku do haldy vložíme prvok na koniec posledného riadku a overujeme narušenie prvej vlastnosti. Minimum je prvým prvkom zoznamu alebo tiež koreňom stromu. Pri odoberaní minima nahradíme pôvodne minimum posledným prvkom haldy a overujeme porušenie prvej vlastnosti.

Vytvorenie haldy je rozdelené na nasledujúce stavy:

1. *Inicializácia* $\rightsquigarrow$ 2 - Nastáva iba zmena stavu.
2. *Vloženie do haldy* $\rightsquigarrow$ 3, inicializácia ničenia - Ak sme haldu naplnili, začneme s jej ničením, inak zmeníme stav na 3.
3. *Inicializácia vkladania* $\rightsquigarrow$ 4 - Vložíme prvok na koniec haldy.
4. *Bublanie* $\rightsquigarrow$ 2, 5 - Ak je hodnota prvku menšia ako jeho predka, zmeníme stav na 5, inak na 2.
5. *Výmena* $\rightsquigarrow$ 4 - Vymeníme hodnoty prvku a jeho predka.

Zničenie haldy je rozdelené na nasledujúce stavy:

1. *Inicializácia* $\rightsquigarrow$ 2 - Nastáva iba zmena stavu.
2. *Inicializácia ničenia* $\rightsquigarrow$ 3, koniec - Ak sme haldu už zničili, ukončíme algoritmus, inak zmeníme stav na 3.
3. *Odstránenie minima* $\rightsquigarrow$ 4 - Vymeníme posledný prvok v halde s prvým.

4. *Inicializáca bublania* $\rightsquigarrow$ 2, 5 - Ak je nejaký potomok menší ako opravovaný vrchol (pri prvom navštívení je to koreň), zmeníme stav na 5, inak na 2.

5. *Nájdenie minimálneho indexu* $\rightsquigarrow$ 2 - Nájdeme menšieho z potomkov.

6. *Otázka bublania* $\rightsquigarrow$ 2, 7 - Ak je nutné vymeniť prvky (potomka a predka), zmeníme stav na 7, inak na 2.

7. *Výmena* $\rightsquigarrow$ 4 - Vymeníme hodnoty opravovaného prvku a jeho menšieho potomka.

3.12 Triedenie zlievaním (Mergesort)

Klasický príklad použitia techniky *rozdeľ a panuj*. Triedenie zlievaním sa dá napísať do niekoľkých krokov:

1. Rozdeľ zoznam na dva približne rovnako veľké zoznamy.

2. Pretrieď prvý zoznam.

3. Pretrieď druhý zoznam.

4. Zlej zoznamy dohromady.

V štvrtom kroku zlievame dohromady dva utriedené zoznamy. Zo zoznamov vždy vyberieme menší prvok, pridáme do výsledného zoznamu a posunieme sa na ďalší prvok vo výslednom zozname a aj v zozname, z ktorého sme prvok brali.

Mergesort je rozdelený do nasledujúcich stavov:

1. *Inicializácia* $\rightsquigarrow$ 2 - Nastáva iba zmena stavu.

2. *Utrieď zoznam* $\rightsquigarrow$ 3 - Pokúsime sa utriediť aktuálny zoznam.

3. *Zastavenie rekurzie* $\rightsquigarrow$ 4, 5, 6 - Rozhodneme sa ktorú časť zoznamu budeme ďalej triediť.

4. *Utrieď dolné indexy* $\rightsquigarrow$ 3 - Pokúsime sa utriediť dolné indexy.

5. *Utrieď horné indexy* $\rightsquigarrow$ 3 - Pokúsime sa utriediť horné indexy.

6. *Zlej zoznamy* $\rightsquigarrow$ 7, 8 - Zlievame susedné zoznamy.

7. *Pridaj dolný index* $\rightsquigarrow$ 6, 9 - Ak existuje v ľavom zozname prvok na pridanie, tak ho pridáme do výsledného zoznamu, posunieme index a zmeníme stav na 6. Inak zmeníme stav na 9.

8. *Pridaj horný index* $\rightsquigarrow$ 6, 9 - Ak existuje v pravom zozname prvok na pridanie, tak ho pridáme do výsledného zoznamu, posunieme index a zmeníme stav na 6. Inak zmeníme stav na 9.

9. *Skopíruj výsledok* $\rightsquigarrow$ 5, 6, koniec - Skopírujeme zliaty zoznam do triedeného zoznamu. Stav zmeníme podľa toho, ktorú ďalšiu časť zoznamu treba utriediť.

3.13 Grafové algoritmy

S grafmi sa stretávame pri programovaní skoro všade. Väčšina dátových štruktúr je nejakou ich formou. Riešenie problémov sa dá takisto reprezentovať pomocou grafov a preto je ich prehľadávanie dôležité.

3.14 Prehľadávanie do hĺbky a do šírky (DFS a BFS)

Pri základných dátových štruktúrach som spomínal, že sa používajú ako podklad pre rôzne algoritmy. Prehľadávanie do šírky a prehľadávanie do hĺbky sú toho pekným príkladom. Obidva algoritmy sú totožné až na to, že na ukladanie dát používajú rôzne štruktúry. Obidva algoritmy sa dajú v krokoch vyjadriť takto:

1. Pridaj vrchol do zoznamu a označ ho ako preskúmaný.
2. Odober vrchol zo zoznamu.
3. Pridaj do zoznamu všetkých nepreskúmaných susedov posledne odobratého vrcholu a označ ich ako preskúmaných.
4. Ak nie je zoznam prázdny, pokračuj krokom 2.

Prehľadávanie do šírky používa ako zoznam frontu, zatiaľčo prehľadávanie do hĺbky používa zásobník.

3.15 Dijkstrov algoritmus

Algoritmus slúži na nájdenie najkratšej cesty v grafe z ľubovoľného vrcholu do všetkých ostatných. Dá sa opísať v niekoľkých krokoch:

1. Všetkým vrcholom v grafe nastav vzdialenosť $+\infty$.
2. Vrcholu, z ktorého chceme vzdialenosť merať, nastav vzdialenosť na 0.
3. Ak existuje vrchol, ktorý je nepreskúmaný, vezmi ten s najnižšou vzdialenosťou a označ ho v, inak ukonči algoritmus.
4. Označ v ako preskúmaný.
5. Vezmi všetkých susedov v, ktorý sú nepreskúmaný a prepočítaj ich novú vzdialenosť, ak by tam viedla cesta z v. V prípade, že je kratšia, ulož novú vzdialenosť.
6. Pokračuj krokom 3.

Ľahko vidieť, že každý vrchol bude preskúmaný práve raz. Nájdenie prvku s najmenšou vzdialenosťou ale nie je triviálne a má logaritmickú zložitosť. Celý algoritmus má teda zložitosť $n \cdot log(n)$.

Kapitola 4

Programátorská dokumentácia

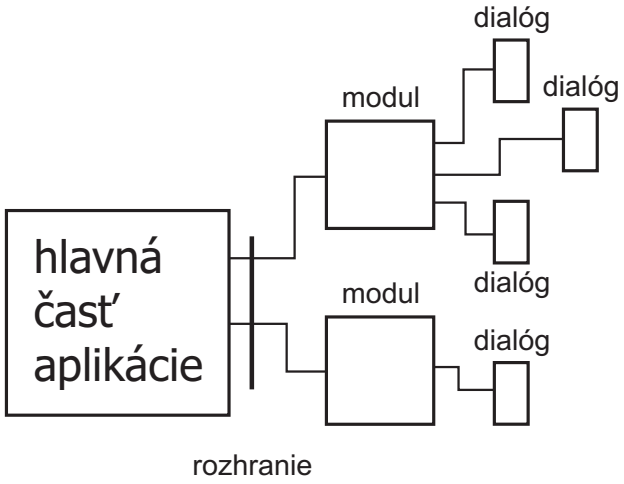
V tejto kapitole sa pokúsim opísať tie najdôležitejšie časti aplikácie ako je rozhranie, ktorého znalosť je nutná k programovaniu modulov, spoločné črty všetkých modulov, spôsob uloženia pseudokódu, ako som docielil to, aby sa dali algoritmy krokovať a hlavnú časť aplikácie. Samotný kód je okomentovaný a nemal by byť problém s jeho chápaním. Dokumentáciu vygenerovanú dokumentačným systémom Doxygen[3] je možné nájsť na priloženom CD.

4.1 Architektúra

Prostriedky, ktoré boli použité pri tvorbe aplikácie sú spomenuté v kapitole týkajúcej sa implementácie. Ako jazyk je použitý programovací jazyk C++, aplikácia bola vytvorená pre operačný systém Microsoft Windows a moduly sú dynamicky linkované knižnice vyvinuté firmou Microsoft pre tento systém. Táto sekcia sa týka toho, ako je to všetko zapojené dohromady. Architektúru aplikácie najlepšie vystihuje obrázok 4.1, na ktorom je zobrazené prepojenie hlavnej časti s modulmi. Komunikácia prebieha skrz rozhranie, ktoré musí každý modul implementovať[1]. Pri načítavaní modulu do pamäte je postupne overované, či modul implementuje dohodnuté rozhranie, či obsahuje svoje meno (*std::wstring title;*), opis (*std::wstring desc;*), implementuje funkciu pre vytvorenie dialógu (*create_dlg_fnc create_dlg;*) a funkciu volanú pri uvoľňovaní modulu z pamäte (*free_lib_fnc free_lib;*). Ak sa tak nestane, modul bude z pamäte uvoľnený a táto skutočnosť bude oznámená užívateľovi. Pomocou funkcie *create_dlg*, ktorú volá hlavná časť

[1] Viac o rozhraní je možné nájsť v sekcií týkajúcej sa čisto rozhrania.

aplikácie je modulu oznámené, že má vytvoriť dialóg s vizualizáciou. Tých môže byť počas behu vytvorených viacero a preto je na module aby si viedol informácie o tom aké dialógy má momentálne otvorené. Tesne pre uvoľnením modulu z pamäte je volaná funkcia *free_lib*. Modul má možnosť skontrolovať si všetky svoje otvorené dialógy, pokúsiť sa ich korektne zavrieť a ak sa mu to nepodarí, ohlási to hlavnej časti aplikácie pomocou návratovej hodnoty. Tá to oznámi užívateľovi a pokúsi sa o uvoľnenie neskôr.



Obrázok 4.1: Architektúra aplikácie

4.2 Hlavná časť aplikácie

Úlohou hlavnej časti aplikácie je načítavať a uvoľňovať moduly z pamäte a spúšťať vizualizácie. Táto časť aplikácie taktiež kategorizuje moduly do stromovej štruktúry. Vnútorné vrcholy stromu obsahujú informácie o triedach algoritmov, v listoch sú uložené informácie o konkrétnych moduloch. Všetky dáta sú obsiahnuté v objekte triedy *gen_tree*, ktorá reprezentuje strom s variabilným počtom potomkov a dokáže ukladať rôzne typy pre vnútorné vrcholy a listy. Jedná sa o šablónu, ktorá obsahuje dva parametre. Prvý je typ pre vnútorné vrcholy stromu a druhý je typ pre listy.

```
template<typename inner_t, typename leaf_t> class gen_tree;
```

Trieda obsahuje tri vnútorné triedy, ktoré reprezentujú vrcholy stromu. Trieda *tree_node* reprezentuje ľubovoľný vrchol stromu. Triedy *inner_node* a

leaf_node reprezentujú vnútorné vrcholy stromu a listy. Posledné dve triedy sú priamymi potomkami triedy *tree_node* a ich jedinou úlohou je udržovanie hodnoty vrcholu. Preto stojí za zmienku iba prvá z nich a jej rozhranie. To sa dá rozdeliť na tri časti. Prvá sa týka indikačných funkcií, druhá funkcií vracajúcich hodnoty vrcholu a posledná poskytuje možnosť prechádzať strom.

```
// general tree node interface.
class tree_node {

        // indicators.
        bool valid() const;
        bool leaf() const;

        // getting values.
        inner_t &inner_value();
        leaf_t &leaf_value();

        // tree browsing.
        tree_node *parent() const;

        typedef typename vector<tree_node *>::iterator iterator;
        iterator begin();
        iterator end();

};
```

Funkcia *valid()* vracia informáciu o tom, či je vrchol platný. Štandardný konštruktor dovolí vytvoriť iba neplatný vrchol a tým znemožňuje užívateľovi vytvoriť vrchol mimo strom a predať ho nejakej funkcií. Funkcia *leaf()* vracia informáciu o tom, či je vrchol list alebo je vnútorným vrcholom stromu. Jedná sa o veľmi dôležitú funkciu, lebo pred každým prevzatím hodnoty zo stromu by sa malo skontrolovať z akého druhu vrcholu hodnotu berieme.

Funkcia *inner_value()* vracia referenciu na hodnotu uloženú vo vnútornom vrchole. V prípade, že bude volaná na list, vyvolá sa *ASSERT()*. Podobne sa chová aj funkcia *leaf_value()* až na to, že vracia referenciu na hodnotu v liste.

Funkcia *parent()* vracia ukazovateľ na rodiča vrcholu. V prípade, že sa jedná o koreňový vrchol, vracia nulový ukazovateľ. Typ *iterator* umožňuje prechádzať potomkov vrcholu. Chová sa rovnako ako iterátor v STL triede

vector. Taktiež rovnako ako ich ekvivalenty v tejto STL triede sa chovajú aj funkcie *begin()* a *end()*.

4.3 Rozhranie

Snaha bola docieliť aby bolo rozhranie pre moduly minimálne a bola tak ponechaná veľká voľnosť autorom modulov. Kód rozhrania je uložený v súbore *iface.h* a každý modul by ho mal obsahovať. Celé rozhranie sa nachádza v mennom priestore *iface*.

iface.h

```
namespace iface {

        // module creation and destruction.
        typedef CWnd *(* create_dlg_fnc)(void (* fnc)(CWnd *wnd));
        typedef bool (* free_lib_fnc)();

        // module information.
        const std::string title("title");
        const std::string desc("description");
        const std::string create_dlg("create_dlg");
        const std::string free_lib("free_lib");

}
```

Skladá sa z dvoch častí. Prvá časť obsahuje definície typov funkcií, ktoré sa volajú pri vytváraní dialógu a pri uvoľňovaní modulu. Druhá časť obsahuje názvy funkcií a premenných, ktoré očakáva, že bude modul definovať.

Pre vytvorenie dialógu zavolá hlavná časť aplikácie funkciu typu *create_dlg_fnc*. Ako prvý parameter dostane funkcia ukazovateľ na inú funkciu, ktorú má modul zavolať po zatvorení dialógu. V prípade nedodržania tohto pravidla vznikne nekonzistencia medzi hlavnou aplikáciou[2] a modulom. Funkcia, ktorá ja volaná pri zatvorení dialógu (parameter *fnc*) má ako parameter ukazovateľ na zatvorený dialóg (skoro všetky objekty v MFC sú podedené po triede *CWnd*). Funkcia typu *free_lib_fnc* je volaná hlavnou aplikáciou vždy pri uvoľňovaní modulu z pamäti. Modul má možnosť pomocou

[2]Tá si bude myslieť, že dialóg existuje aj napriek tomu, že bol už zatvorený

návratovej hodnoty oznámiť hlavnej aplikácií, že by jeho uvoľnenie z pamäti mohlo spôsobiť problémy. V prípade, že tak nastane, aplikácia sa pokúsi modul uvoľniť neskôr.

Každý modul musí obsahovať definície funkcií a premenných pomenovaných podľa mien uvedených v druhej časti. Premenné a funkcie spolu s typmi a ich významom, ktoré musia byť definované, sú uvedené v nasledujúcom zozname:

- *std::wstring title;* - Titulok modulu.
- *std::wstring description;* - Opis modulu.
- *create_dlg_fnc create_dlg;* - Funkcia pre vytvorenie dialógu.
- *free_lib_fnc free_lib;* - Funkcia volaná pri uvoľňovaní modulu z pamäte.

4.4 Moduly všeobecne

Snaha bola pre všetky moduly uchovať čo najviac spoločných rysov aj napriek tomu, že je autorom ponechaná veľká voľnosť pri ich tvorbe. Týka sa to najmä implementačnej časti. Pre vykresľovanie sa používa multiplatformové API OpenGL. S ním súvisí použitie písma a kamery v scéne. Všetky moduly taktiež používajú v počítači najpresnejší možný časovač.

4.4.1 OpenGL

Všetok kód týkajúci sa OpenGL je uzavrený v mennom priestore *gl*. Konkrétne sa jedná o triedy *wnd*, *scene*, *font* a *camera*.

Trieda *wnd* reprezentuje okno, ktoré používa OpenGL na vykresľovanie svojho obsahu. Používa sa na vizualizáciu algoritmu.

```
// OpenGL window.
class wnd : public CFrameWnd {

        // creation and destruction.
        bool creategl(const wchar_t *title, scene *scn);
        void destroygl();

        // gl functions.
        bool init();
```

```
        void deinit();
        void reshape(
        const unsigned int width,
        const unsigned int height
        );

};
```

Štandardný konštruktor nevytvára žiadne okno. Pre vytvorenie a zničenie okna treba zavolať funkcie *creategl()* a *destroygl()*. Funkcia *creategl()* má ako druhý parameter ukazovateľ na objekt typu *scene*, ktorý sa stará o vykreslenie obsahu okna. Jedná sa o ekvivalenty funkcií *CWnd::CreateEx()* a *CWnd::DestroyWindow()* až na to, že majú menej parametrov a vytvoria prípadne zničia okno podporujúce OpenGL. Rozhranie triedy takisto poskytuje funkcie pre štandardnú prácu s OpenGL. Nemusia byť volané, ale je to nanajvýš odporúčané. V opačnom prípade bude nutné OpenGL inicializovať ručne.

Ďalšou triedou v mennom priestore *gl* je *font*. Jedná sa o písmo, ktoré je možné vykresľovať do OpenGL scény. Písmo je rastrové a jeden objekt môže držať práve jedno písmo určitej veľkosti.

```
// OpenGL font.
class font {

        // creation and destruction.
        void build(
                const CDC *dc,
                const wchar_t *name,
                const unsigned int size
        );
        void destroy();

        // printing.
        void print(const char *str);
        void print(const std::string &str);

        // manipulators.
        class col;
        class pos;
```

```
        // stream operators.
        font &operator <<(const char *str);
        font &operator <<(const std::string &str);
        font &operator <<(const int i);
        font &operator <<(const double d);
        font &operator <<(const bool b);

};
```

Pre vytvorenie písma je nutné vlastniť ukazovateľ na objekt typu *CDC*, čo je trieda poskytujúca funkcie pre prácu s *device contextom*. Pri vytváraní písma vyberáme jeho meno a veľkosť.

Zatiaľ máme triedu pre OpenGL okno a triedu pre OpenGL písmo. Keďže oknu treba „povedať“, čo má vykresľovať, potrebujeme nejakú reprezentáciu scény.

```
// OpenGL scene.
class scene {

        // virtual methods.
        virtual void init();
        virtual void draw() = 0;
        virtual bool update(const float ms);

};
```

Pokúsim sa teraz zhrnúť ako má vyzerať vytvorenie plne funkčného OpenGL okna. Programátor podedí po triede *gl::scene* nejakú svoju triedu a ukazovateľ na objekt tejto triedy predá objektu vytvorenému z *gl::wnd* počas vytvárania okna. Autor podedenej triedy musí implementovať čiste virtuálnu metódu *draw()*, ktorú samotné okno používa na vykreslenie jeho obsahu. Metódy *init()* a *update()* nemusia byť prepísané, ale odporúča sa to. Metóda *init()* je volaná po vytvorení okna, ktoré aktualizovalo niektoré vlastnosti objektu ako *device context*, *rendering context* a ukazovateľ na majiteľa scény. Ja používam túto metódu na vytvorenie objektu písma použitého v scéne. Ten potrebuje práve *device context*. Metódu *update()* volá okno vždy, keď si myslí, že bude vhodné aktualizovať scénu a ako parameter predá čas od poslednej aktualizácie v milisekundách. Návratová hodnota funkcie indikuje, či bola aktualizácia scény kompletná alebo treba scénu aktualizovať znova.

4.4.2 Časovač

Časovač používa počítadlo tikov CPU na výpočet času. API systému Microsoft Windows poskytuje k tomuto účelu dve funkcie. *QueryPerformanceFrequency()* vráti v parametri, ktorý je predaný ako referencia, počet tikov za sekundu a *QueryPerformanceCounter()* vráti takisto v parametri predanom ako referencia počet tikov od nejakého konkrétneho momentu. Výpočet času je už potom jednoduchý a za zmienku teda stojí už len ukážka rozhrania triedy.

```
// performance timer.
class pttimer {

        // timer control.
        bool start();
        float stop();
        float lap();

        // state indication.
        bool running();

};
```

Funkcia *start()* vracia hodnotu podľa toho, či sa podarilo časovač spustiť. Funkcie *stop()* a *lap()* vracajú čas od posledného „kola“ v milisekundách. Funkcia *running()* indikuje, či je časovač spustený.

4.5 Krokovanie

Každý algoritmus sa dá rozdeliť na rôzne stavy a jasne definovať prechody medzi týmito stavmi. Ja som založil krokovanie na úplne rovnakom princípe. Vytvoril som stavy algoritmu a definoval prechody medzi nimi. Krokovacia funkcia potom vyzerá následovne:

```
// do algorithm step.
bool step(..., const bool anim = false)
{
        // switch phases.
        switch (phase) {
```

```
                // xth phase - ...
                case x: {
                        // phase code.
                        phase = ...;
                        return true/false;
                }

        }
}
```

Funkcia sa pomocou aktuálneho stavu rozhodne, aký kód vykoná. Návratová hodnota oznamuje volajúcemu, či algoritmus skončil svoju činnosť. Na konci kódu patriacemu nejakému stavu by preto mala byť zmena aktuálneho stavu a volanie *return* s parametrom podľa toho, či sa jedná o koncový stav alebo nie.

4.6 Pseudokód

V predchádzajúcej kapitole bolo spomenuté, že algoritmy sú delené na stavy (fázy). Každá z týchto fáz má pridelené svoje meno, počet riadkov v zdrojovom kóde, ktoré sa tejto fázy týkajú a zoznam týchto riadkov. Samotný kód je len dvojrozmerné pole znakov (riadok je jednorozmerné pole znakov a kód je pole riadkov). Kód, počty riadkov a zoznamy týchto riadkov sú definované ako konštanty v *.cpp* súbore. Uvediem konkrétny príklad pre bubblesort.

```
// phases names.
const wchar_t *BUBBLE_PHASES_NAMES[] = {
        L"Algorithm initialization",
        L"Loop initialization",
        ...
};

// 2-dimensional array of source.
const wchar_t *BUBBLE_SOURCE[] = {
        L"integer array[array_size];",
        L"integer index = 0;",
        ...
};

// lines selection and lines count for each phase.
```

```
const unsigned int BUBBLE_SOURCE_PHASE0[] = {0, 1, 2, 4, 5};
const unsigned int BUBBLE_SOURCE_PHASE0_LINES = 5;
const unsigned int BUBBLE_SOURCE_PHASE1[] = {7, 8};
const unsigned int BUBBLE_SOURCE_PHASE1_LINES = 2;
...

// array of lines selections.
const unsigned int *BUBBLE_SOURCE_PHASES[] = {
        BUBBLE_SOURCE_PHASE0,
        BUBBLE_SOURCE_PHASE1,
        ...
};

// array of lines counts.
const unsigned int BUBBLE_SOURCE_PHASES_LINES[] = {
        BUBBLE_SOURCE_PHASE0_LINES,
        BUBBLE_SOURCE_PHASE1_LINES,
        ...
};
```

Trieda implementujúca algoritmus[3] obsahuje funkcie, ktoré vracajú celý kód, počet riadkov tohto kódu, aktuálny výber riadkov v kóde a počet riadkov tohto výberu.

[3]Trieda je podedená po triede *gl::scene*, aby mohla vykresľovať vizualizáciu algoritmu do okna.

Kapitola 5

Záver

Cieľom práce bolo vytvoriť aplikáciu pre vizualizáciu algoritmov, ktorá by mala slúžiť ako pomôcka pri štúdiu prípadne výuke. Aplikácia sa nezameriava na jeden konkrétny algoritmus alebo skupinu algoritmov, ale je ľahko rozšíriteľná o ďalšie algoritmy. Jedným z hlavných dôvodov prečo som sa rozhodol pre tento projekt bola nespokojnosť s už existujúcimi projektmi.

5.1 Iné projekty

Na internete možno nájsť množstvo malých projektov zameraných na vizualizáciu algoritmov. Väčšina sa však zameriava buď na jeden konkrétny algoritmus alebo na malú skupinu algoritmov. Hlavným nedostatkom u veľkého množstva z nich bolo pre mňa, že snaha bola vizualizovať iba niektoré konkrétne situácie a nebolo možné algoritmus spustiť na iné vstupné dáta. Tím, ktoré prijímali vstupné dáta, chýbal slovný doprovod k tomu, čo sa deje. Najčastejšie bol pre tvorbu použitý jazyk Java a tak bolo možné pustiť vizualizáciu vo webovom prehliadači. Výhoda je v podpore na veľkom množstve platforiem, nevýhoda v závislosti na prídavných aplikáciach a pomalosti. Tento projekt bol naopak zameraný na samostatné štúdium a slovný doprovod bol teda nutný. Ponecháva veľký priestor na jeho rozšírenie o ľubovoľné ďalšie algoritmy a zameriava sa len na jednu platformu, pričom skvalitňuje vizualizáciu.

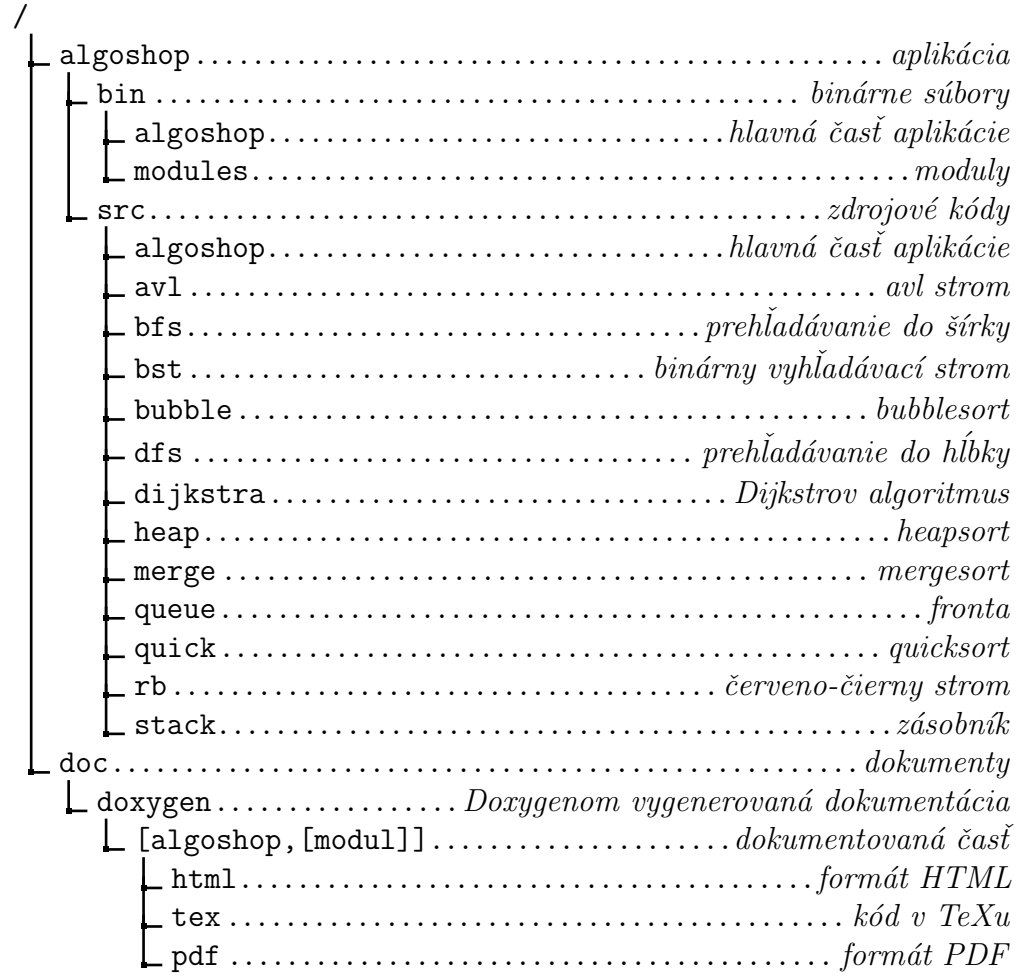
Pre študentov Matematicko-fyzikálnej fakulty Univerzity Karlovej je samostatnou kapitolou projekt Algovision[1]. Projekt je vytvorený v jazyku Java a preto je možné ho spúšťať cez webový prehliadač z internetu. Pýši sa veľkým množstvom vizualizovaných algoritmov a jeho dostupnosťou.

Ovládanie je prehľadné a dostatočne konzistentné. Proti však hovorí jeho rýchlosť a funkčnosť. Veľké množstvo užívateľov malo problém niektoré algoritmy spustiť a u iných zasa nastali chyby pri behu.

5.2 Obsah CD

Na priloženom CD sa nachádza samotná aplikácia, jej zdrojové kódy a dokumentácia spolu s týmto textom[1]. Užívateľská a programátorská dokumentácia sú obsiahnuté v tomto texte. Dokumentácia vygenerovaná dokumentačným systémom Doxygen[3] vo formáte HTML a PDF spolu so zdrojovým kódom v TeXu sa nachádza na CD. Strom adresárovej štruktúry priloženého CD s popisom k adresárom možno nájsť na obrázku 5.1.

[1] Zdrojový kód v TeXu a výsledný súbor vo formáte PDF.



Obrázok 5.1: Stromová adresárová štruktúra obsahu priloženého CD.

Literatúra

[1] Algovision: *http://kam.mff.cuni.cz/˜ludek/*.

[2] Andrei Alexandrescu: *Modern C++ design : generic programming and design patterns applied*, Addison-Wesley, 2001.

[3] Doxygen: *http://www.stack.nl/˜dimitri/doxygen/*.

[4] Scott Meyers: *Effective C++, Second Edition*.

[5] Scott Meyers: *More Effective C++*, 1996.

[6] MSDN Library: *http://msdn.microsoft.com*.

[7] MSDN MFC Reference:
http://msdn.microsoft.com/en-us/library/d06h2x6e.aspx.

[8] NeHe Productions: *http://nehe.gamedev.net*.

[9] Bjarne Stroustrup: *The C++ Programming Language, Third Edition*, Addison-Wesley, 1997.

[10] Tahaky-Referaty.sk: *http://www.tahaky-referaty.sk/*

[11] Wikipedia, The Free Encyclopedia: *http://www.wikipedia.org*.

[12] Windows API Reference:
http://msdn.microsoft.com/en-us/library/aa383749(VS.85).aspx.

Dodatok A

Užívateľská dokumentácia

Samotná aplikácia je rozdelená na jej hlavnú časť a moduly reprezentujúce algoritmy. Hlavná časť aplikácie má na starosti načítavanie a uvoľňovanie modulov z pamäti, ktoré sú na nej až na niekoľko pravidiel absolútne nezávislé. Moduly môžu byť pohodlne do aplikácie pridávané a odoberané pomocou úpravy konfiguračného súboru o čom sa dočítate viac v sekcií venovanej hlavnej časti aplikácie.

A.1 Hlavná časť aplikácie

Táto časť aplikácie sa stará o načítavanie a uvoľňovanie modulov z pamäte. Dialóg na obrázku A.1 môžeme rozdeliť na 4 časti:

- Dostupné moduly/Available modules.
- Načítané moduly/Loaded modules.
- Opis/Descripition.
- Ovládanie/Controls.

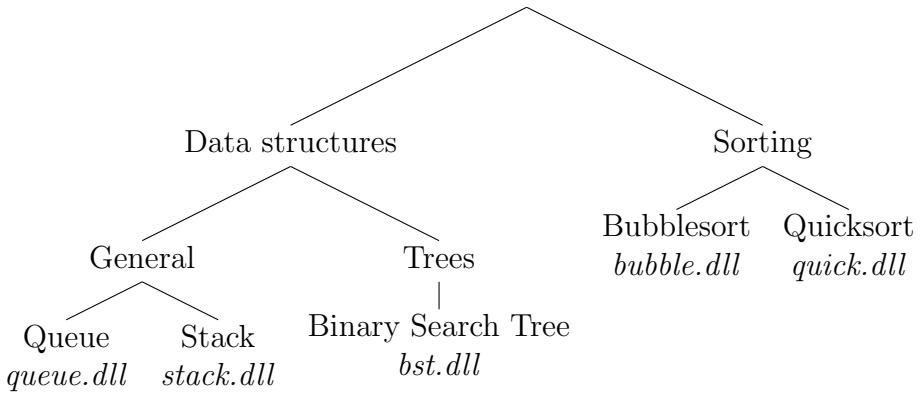
Dostupné moduly sú uložené v stromovej štruktúre, ktorej vnútorné vrcholy sú triedy algoritmov a listy samotné moduly. Rozdelenie do stromovej štruktúry uľahčuje orientáciu medzi jednotlivými algoritmami. Štruktúra sa dá upravovať pomocou editácie súboru '*modules.lst*'. Jedná sa o textový súbor, ktorého riadky predstavujú vrcholy stromu. Oddeľovačom položiek na riadku je znak '*:*'. Prvá položka riadku obsahuje buď znak *M (Module)* alebo znak *D (Directory)*. *M* znamená, že riadok obsahuje informácie

o module teda liste daného stromu a *D* znamená, že sa riadok týka triedy algoritmov a teda vnútorného vrcholu stromu. Ďalšia položka udáva meno vrcholu a jej odsadenie pomocou medzier udáva polohu v strome. Vrcholy s rovnakým odsadením pod sebou patria tomu istému vrcholu nad ním, ktorý má odsadenie o jednu medzeru menšie. Správa sa to podobne ako sa správajú skupiny príkazov v jazyku *Python*. V prípade, že sa jednalo o vnútorný vrchol stromu, je táto položka aj poslednou položkou na riadku. V prípade, že sa jednalo o modul, riadok obsahuje ešte jednu položku a to cestu k dynamicky-linkovanej knižnici, ktorá modul implementuje. Príklad súboru '*modules.lst*':

modules.lst

```
D:Data structures
D: General
M:  Queue:queue.dll
M:  Stack:stack.dll
D: Trees
M:  Binary Search Tree:bst.dll
D:Sorting
M: Bubblesort:bubble.dll
M: Quicksort:quick.dll
```

V tomto príklade bude vyzerať výsledný strom následovne:



Načítané moduly sú uložené v lineárnom zozname. Moduly by sa mali načítavať len vtedy, ak sa budú naozaj používať, lebo implementácia algoritmov zaberá miesto v pamäti a ich zbytočné načítanie môže spôsobiť

spomalenie behu aplikácie. V zozname je uložené meno modulu, ktoré bolo načítané zo samotného modulu a môže sa líšiť od mena v strome dostupných modulov a počet okien, ktoré má modul otvorené. Ak sa modul chová podľa dohodnutého rozhrania, mali by byť všetky informácie správne. Chybné chovanie môže spôsobiť chybné informácie v tomto zozname.

Opis je statický text, ktorý vypĺňa samotný modul tím, že ho vyberieme zo zoznamu načítaných modulov. Mal by obsahovať základné informácie o algoritme a poskytnúť tak užívateľovi možnosť nahliadnuť na to, či si vybral ten správny modul.

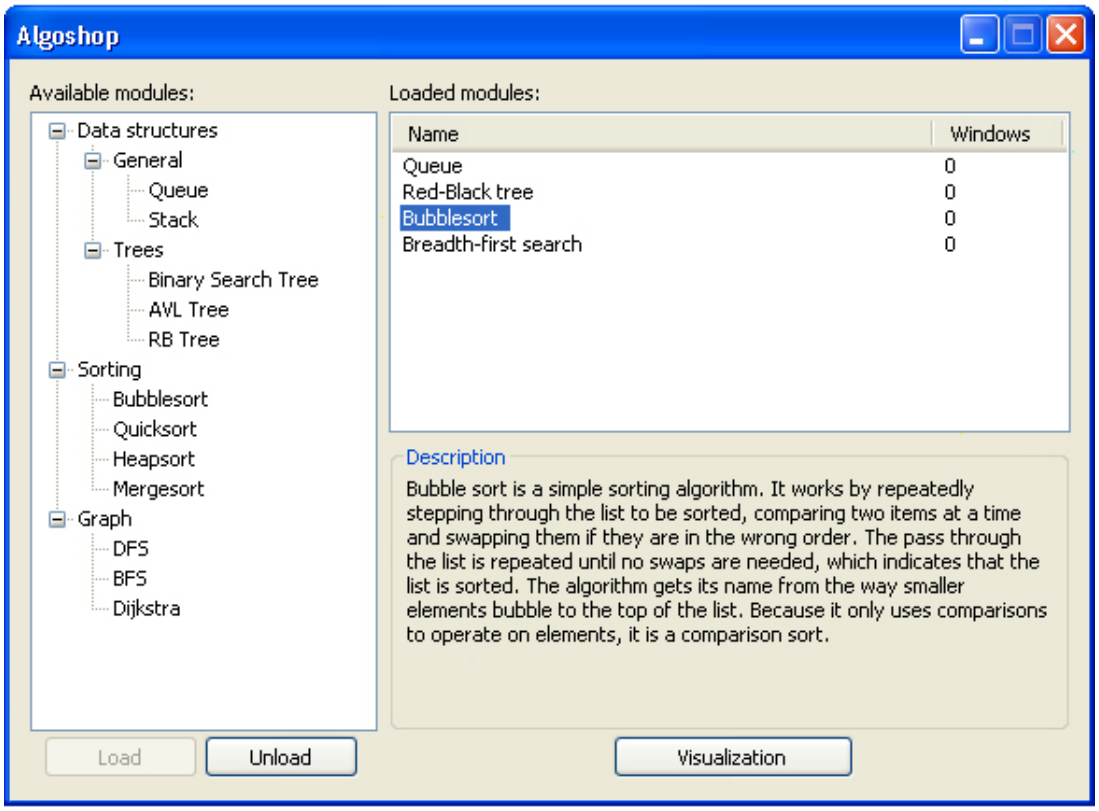
Ovládanie je umiestnené v spodnej časti dialógu a obsahuje 3 tlačidla:

- Načítaj/Load
- Uvoľni/Unload
- Vizualizácia/Visualization

Tlačidla *Load* a *Unload* sa vzťahujú k časti *Dostupné moduly*. *Load* načíta modul označený v strome dostupných modulov do pamäti a pridá ho do zoznamu načítaných modulov. *Unload* naopak modul [1] uvoľní a odstráni zo zoznamu načítaných modulov.

Tlačidlo *Visualization* sa vzťahuje k časti *Načítané moduly*. Jeho stlačenie otvorí dialóg modulu vyznačeného v tejto časti.

[1] Tlačidlo *Unload* sa taktiež vzťahuje k modulu označenému v zozname dostupných modulov.



Obrázok A.1: Dialóg hlavnej aplikácie s načítanými modulmi *Queue*, *Red-Black tree*, *Bubblesort* a *Breadth-first search* s výberom a opisom modulu *Bubblesort*.

A.2 Moduly všeobecne

Každý modul obsahuje *ovládací dialóg* a *vizualizačné okno*. *Ovládací dialóg* sa líši približne podľa triedy algoritmov, zatiaľčo *vizualizačné okno* je všade rovnaké pre zachovanie čo najväčšej konzistencie zobrazovania algoritmu.

Kamera vo vizualizačnom okne sa dá ovládať buď pomocou klávesnice alebo myši.

- Klávesnica:
  - *Šípka doľava* – pohyb kamerou doľava.
  - *Šípka doprava* – pohyb kamerou doprava.
  - *Šípka nahor* – pohyb kamerou nahor.
  - *Šípka nadol* – pohyb kamerou nadol.
  - *Page Up* – pohyb kamerou dopredu.
  - *Page Down* – pohyb kamerou dozadu.
- Myš – Stlačenie ľavého tlačidla a pohyb myšou.

Obrázok A.2: Titulok vizualizačného okna - Zobrazujú sa *Red-Black tree* vo fáze *Empty tree*.

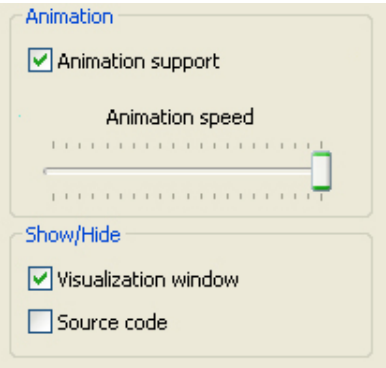
Napriek tomu, že sa ovládacie dialógy líšia, majú aj mnoho spoločných častí. Každý jeden sa skladá z troch častí:

- Ovládanie/Controls
- Animácia/Animation
- (Ukáž/Schovaj)/(Show/Hide)

Práve v časti *Controls* sú najväčšie rozdiely. Na druhú stranu časť *Animation* je všade rovnaká a v časti *Show/Hide* pribúda v niektorých moduloch jedna položka. Rozoberiem teda spoločné prvky ovládacích dialógov.

Časť *Animation* obsahuje jedno zaškrtávacie políčko s názvom *Animation support*, pomocou ktorého sa zapína a vypína animácia pri vizualizácií. Táto časť ďalej obsahuje posuvník, ktorý určuje rýchlosť animácie.

Časť *Show/Hide* vždy obsahuje zaškrtávacie políčko s názvom *Visualization window*, pomocou ktorého môžeme zobraziť/schovať vizualizačné okno. V niektorých moduloch pribudne v tejto časti aj ďalšie zaškrtávacie políčko s názvom *Source dialog*, pomocou ktorého môžeme zobraziť/schovať dialóg s pseudokódom popisujúcim algoritmus.

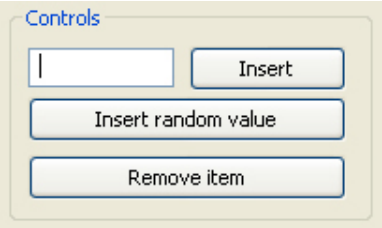


Obrázok A.3: Spoločné prvky ovládacieho dialógu.

A.3 Základné dátové štruktúry

Ovládacia časť dialógu (obrázok A.4) obsahuje 4 komponenty:

- Políčko na vyplnenie vstupných dát.
- Tlačidlo s nápisom *Insert*, ktoré vloží dáta z políčka do štruktúry.
- Tlačidlo s nápisom *Insert random value*, ktoré vloží náhodne vygenerovanú hodnotu.
- Tlačidlo s nápisom *Remove item*, ktoré odstráni prvok.

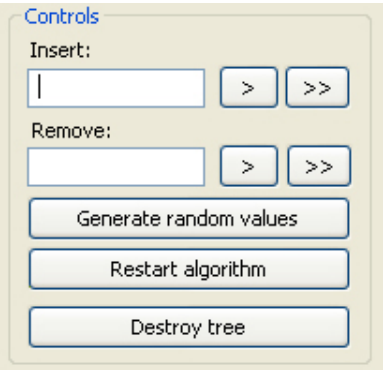


Obrázok A.4: Ovládanie základných dátových štruktúr.

A.4 Binárne stromy

Ovládanie (obrázok A.5) je pre všetky stromy rovnaké a obsahuje:

- Časť *Insert* a *Remove*:
  - Políčko na vyplnenie vstupných hodnôt.
  - Tlačidlo s väčšítkom, ktoré vykoná krok algoritmu s použitím hodnoty v políčku.
  - Tlačidlo s dvoma väčšítkami, ktoré spustí algoritmus na hodnotu v políčku bez zastavenia.
- Tlačidlo s nápisom *Generate random values*, ktoré vygeneruje náhodné hodnoty pre políčka v častiach *Insert* a *Remove*.
- Tlačidlo s nápisom *Restart aglorithm*, ktoré reštartuje algoritmus.
- Tlačidlo s nápisom *Destroy tree*, ktoré zničí strom a reštartuje algoritmus.

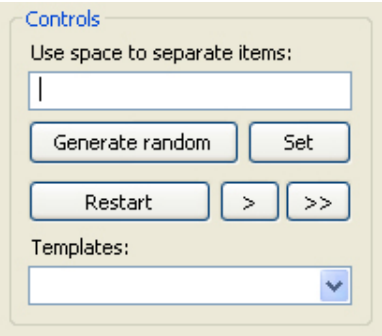


Obrázok A.5: Ovládanie stromových algoritmov.

A.5 Triediace algoritmy

Ovládanie (obrázok A.6) je pre všetky triediace algoritmy rovnaké a obsahuje:

- Políčko na manuálne zadanie zoznamu na triedenie. Jednotlivé položky zoznamu sa oddeľujú pomocou medzier.
- Tlačidlo s nápisom *Generate random*, ktoré vygeneruje náhodné hodnoty do aktuálneho zoznamu.
- Tlačidlo s nápisom *Set*, ktoré použije obsah políčka na vytvorenie zoznamu
- Tlačidlo s nápisom *Restart algorithm*, ktoré reštartuje algoritmus.
- Tlačidlo s väčšítkom, ktoré vykoná krok algoritmu.
- Tlačidlo s dvoma väčšítkami, ktoré spustí algoritmus.
- Zoznam so šablónami niektorých špecifických zoznamov.

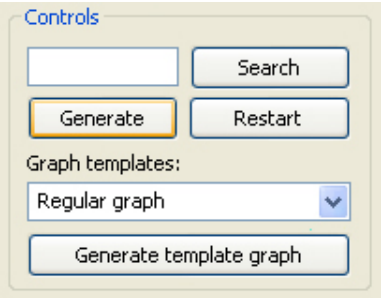


Obrázok A.6: Ovládanie triediacich algoritmov.

A.6 Prehľadávanie grafov

Algoritmy pre prehľadávanie grafov majú rovnakú ovládaciu časť (obrázok A.7):

- Políčko na vyplnenie vstupných údajov. Hodnota, ktorá má byť v grafe nájdená.
- Tlačidlo s nápisom *Search*, ktoré spustí krok algoritmu pre hľadanie v grafe.
- Tlačidlo s nápisom *Generate*, ktoré vyplní políčko náhodnou hodnotou z grafu.
- Tlačidlo s nápisom *Restart*, ktoré reštartuje algoritmus.
- Zoznam so šablónami typov grafov. Zmena v tomto zozname len prepne šablónu. Na vygenerovanie grafu z aktuálnej šablóny treba použiť ďalšie tlačidlo.
- Tlačidlo s nápisom *Generate template graph*, ktoré vygeneruje náhodný graf za použitia šablóny.

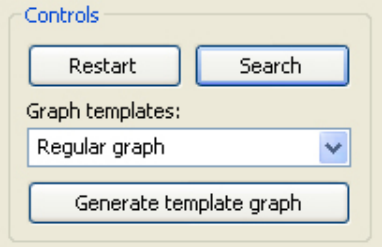


Obrázok A.7: Ovládanie algoritmov na prehľadávanie grafu.

A.7 Dijkstrov algoritmus

Ovládacia časť (obrázok A.8) pre tento algoritmus sa jemne líši od ovládacej časti algoritmov na prehľadávanie grafov:

- Tlačidlo s nápisom *Restart*, ktoré reštartuje algoritmus.
- Tlačidlo s nápisom *Search*, ktoré vykoná krok algoritmu.
- Zoznam so šablónami typov grafov. Zmena v tomto zozname len prepne šablónu. Na vygenerovanie grafu z aktuálnej šablóny treba použiť ďalšie tlačidlo.
- Tlačidlo s nápisom *Generate template graph*, ktoré vygeneruje náhodný graf za použitia šablóny.



Obrázok A.8: Ovládanie Dijkstrovho algoritmu.