UNIVERZITA PARDUBICE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

BAKALÁŘSKÁ PRÁCE

2009 Libor Boháč

Univerzita Pardubice

Fakulta elektrotechniky a informatiky

Demonstrace datových struktur a třídících algoritmů pomocí Java appletů

Libor Boháč

Bakalářská práce

2009

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Katedra informačních technologií
Akademický rok: **2008/2009**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení: **Libor BOHÁČ**

Studijní program: **B2646 Informační technologie**

Studijní obor: **Informační technologie**

Název tématu: **Demonstrace datových struktur a třídících algoritmů pomocí Java apletů**

Zásady pro vypracování:

Cílem práce je vytvořit webovou aplikaci vhodnou pro demonstraci a výuku vybraných datových struktur a třídících algoritmů pomocí Java apletů. Teoretická část: Popis a rozbor používaných datových struktur. Popis a porovnání používaných třídích algoritmů včetně jejich složitosti. Programování Java apletů. Implementační část: Vytvoření demonstračně výukové webové aplikace, která bude pomocí apletů interaktivně demonstrovat činnost vybraných datových struktur a třídících algoritmů. Naprogramované datové struktury budou konfrontovány s implementovanými datovými kontejnery v Javě.

Rozsah grafických prací:

Rozsah pracovní zprávy:

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

Lewis, Denenberg: Data Structures and Their Algorithms, Addison-Wesley 1997. Cormen a kol.: Introduction to algorithms, MIT Press, Cambridge, 2001. Wróblewski: Algoritmy - datové struktury a programovací techniky, Computer Press 2004 http://www.cse.yorku.ca/áaaw/

Vedoucí bakalářské práce: **Ing. Zdeněk Šilar**
Katedra informačních technologií

Datum zadání bakalářské práce: **15. ledna 2009**

Termín odevzdání bakalářské práce: **15. května 2009**

doc. Ing. Simeon Karamazov, Dr.
děkan

L.S.

Ing. Lukáš Čegan
vedoucí katedry

V Pardubicích dne 31. března 2009

Prohlášení

Prohlašuji:

Tuto práci jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 14. 5. 2009

……………………………

Libor Boháč

Poděkování

Tímto děkuji svému vedoucímu práce, Ing. Zdeňku Šilarovi, za rady, které vedly ke zdárnému splnění zadání bakalářské práce.

Abstrakt

Cílem této práce je vytvoření výukové aplikace, jejímž úkolem je graficky demonstrovat činnost vybraných datových struktur a třídících algoritmů. V první části jsou popsány teoretické základy jednotlivých datových struktur a třídících algoritmů. Dále je zde pojednáno o tvorbě Java appletů a možnostech kreslení v jazyku Java. Na závěr je nastíněno řešení výukové aplikace.

Klíčová slova

Datové struktury, třídící algoritmy, Java, applet

Title

Demonstration of data structures and sorting algorithms by the help of Java applets

Abstract

The aim of my project is the creation of the educational application. The main task of the application is graphical representation of the activities of selected data structures and sorting algorithms. The first section describes the theoretical foundation of individual data structures and the sorting of algorithms. There is a treatise on the creation of Java applets and possibilities of drawing in Java. Finally, it is outlined a solution for educational applications.

Keywords

Data structures, sorting algorithms, Java, applet

Obsah

Úvod....10
1 Vymezení pojmů....11
1.1 Abstraktní datový typ (ADT)....11
1.2 Abstraktní datová struktura (ADS)....11
1.3 Složitost datových struktur a algoritmů....11
1.3.1 Časová složitost....11
1.3.2 Paměťová složitost....12
2 Datové struktury....13
2.1 Pole....13
2.1.1 Statické pole....13
2.1.2 Dynamické pole....13
2.2 Lineární seznamy....14
2.2.1 Jednosměrný seznam....14
2.3 Zásobník....15
2.4 Fronta....16
2.5 Halda a prioritní fronta....16
2.6 Množina....17
2.7 Strom....18
2.7.1 Binární strom....19
2.7.2 K-cestný strom....19
2.8 Tabulka....20
2.9 Graf....21
3 Třídicí algoritmy....22
3.1 InsertSort (třídění vkládáním)....22
3.2 SelectSort (třídění výběrem)....22
3.3 BubbleSort (třídění výměnou)....23
3.4 QuickSort (třídění rozdělováním)....23
3.5 MergeSort (třídění sléváním)....24
3.6 RadixSort (číslicové třídění)....25
4 Demonstrační applet....27
4.1 Java Applety....27
4.1.1 Struktura appletu....28
4.1.2 Začlenění appletu do HTML stránky....29
4.2 Kreslení v appletu....29
4.2.1 Kreslící metody....31
4.2.2 Double buffering....34
4.3 Vlákna....34
4.3.1 UML diagram....36
4.3.2 Použité třídy....37
4.4 Vlastní řešení....40
4.4.1 Chybné vykreslování grafiky....40
4.4.2 Zajištění citlivého uživatelského rozhraní....40
4.4.3 Problém určení plnosti vyhledávacího stromu....42
Závěr....43
Seznam použité literatury....44

Seznam obrázků

Obrázek 1: Zásobník ... 15
Obrázek 2: Fronta ... 16
Obrázek 3: Halda ... 17
Obrázek 4: Množina a multimnožina ... 18
Obrázek 5: Strom ... 19
Obrázek 6: Transformace K-cestného stromu na binární ... 20
Obrázek 7: Typy grafů ... 21
Obrázek 8: Třídění vkládáním ... 22
Obrázek 9: Třídění výběrem ... 23
Obrázek 10: Bublinkové třídění ... 23
Obrázek 11: Algoritmus quicksort ... 24
Obrázek 12: Mergesort - třídění sléváním ... 25
Obrázek 13: RadixSort ... 26
Obrázek 14 - ukázka drawArc() ... 32
Obrázek 15 - UML diagram ... 36
Obrázek 16: Přehled použitých tříd ... 37
Obrázek 17 - Rozvržení grafického rozhraní ... 39
Obrázek 18 - Zablokované GUI ... 41
Obrázek 19 - Problém určení grafického znázornění BVS ... 42

Úvod

Při výuce datových struktur nebo třídících algoritmů je potřeba, aby si student dovedl představit, jak daná datová struktura či třídící algoritmus pracuje. Pokud má k dispozici pouze zdrojové kódy, je velmi obtížné si danou problematiku představit. Některé algoritmy pracují rekurzivně nebo v cyklu. Představit si rekurzivně pracující algoritmus může být pro některé studenty značně náročné. Za pomoci slovního popisu a grafické animace, by mělo být pochopení této problematiky snazší.

Vlastní práce je zaměřena na tvorbu grafického znázornění vybraných datových struktur a třídících algoritmů. Aplikace je implementována prostřednictvím programovacího jazyka Java.

První tři kapitoly pojednávají o teoretických základech z oblasti datových struktur a třídících algoritmů. Dále jsou zde popsány jejich implementační možnosti a časové složitosti.

V následující kapitole je popsána samotná tvorba appletu. Nejprve je pojednáno o tvorbě Java appletu, kreslení v Javě a vícevláknové zpracování aplikace. Dále je nastíněn návrh aplikace. Nakonec jsou zmíněny problémy které jsem řešil v průběhu tvorby aplikace.

1 Vymezení pojmů

1.1 Abstraktní datový typ (ADT)

Je-li k datovému typu umožněn přístup pouze přes definované rozhraní, jedná se o abstraktní datový typ. ATD před uživatelem skrývá svoji vnitřní strukturu a poskytuje mu rozhraní, pomocí kterého je možno s ATD pracovat. Pokud se implementace ADT změní, rozhraní je stále stejné. To je výhodné v případě různě výkonných implementací stejného ADT, kde potom nezávisle na implementaci můžeme přistupovat k ADT pomocí stejného rozhraní.

1.2 Abstraktní datová struktura (ADS)

Abstraktní datová struktura je konkrétní instancí ADT. Je možné vytvořit více ADS z jednoho ADT.

1.3 Složitost datových struktur a algoritmů

Datové struktury a algoritmy mohou být různě výkonné a náročné. Potřebu jejich porovnávání řeší právě složitost. Programátor se na základě složitosti rozhodne o použití vhodného algoritmu (datové struktury), který řeší daný problém efektivně.

1.3.1 Časová složitost

Nejčastějším kritériem porovnání je tzv. časová (výpočetní) složitost. Časová složitost je založena na době potřebné k vykonání algoritmu. Doba běhu algoritmu se mění v závislosti na velikosti vstupní množiny dat.

Informaci o časové složitosti algoritmu využijeme v případě, že požadujeme rychlé odezvy programu na požadavky uživatele. Uživatelé jsou velmi netrpěliví, takže by měl program pracovat s minimální odezvou.

V „real-time" systémech je malá časová složitost důležitá z důvodu bezpečnosti, kvality či rychlosti reakcí na změnu vstupů. Pokud by systém nezareagoval včas, mohl by způsobit škody na zdraví nebo majetku.

Co se týče časové složitosti datových struktur, budou nás zajímat zejména operace vkládání, odebírání a vyhledávání prvků. Příkladem mohou být rozsáhlé databáze, se kterými pracují tisíce lidí najednou. Nebylo by žádoucí, kdyby najednou několik uživatelů vzneslo požadavek na vyhledávání dat, čímž by se databázový server výrazně zpomalil.

Složitosti datových struktur a algoritmů mohou dosahovat různých hodnot. Mezi nejčastější patří:

- O(1) – konstantní
- O(log(n)) - logaritmická
- O(n) - lineární
- $O(n^2)$ - kvadratická

1.3.2 Paměťová složitost

Dalším typem je paměťová složitost. Ta podává informaci o velikosti paměti potřebné pro dokončení algoritmu. V dnešní době je velikost operační paměti na většině počítačů dostačující. Z tohoto důvodu se kritérium paměťové složitosti používá méně častěji.

2 Datové struktury

V této kapitole si popíšeme principy jednotlivých datových struktur. Datové struktury jsou výkonnými pomocníky pro logickou organizaci dat. Správná volba datové struktury nám zajistí rychlost a jednoduchost programu. Dále si popíšeme jednotlivé datové struktury.

2.1 Pole

Pole je abstraktní datový typ obsažený snad ve všech programovacích jazycích. Nejčastěji bývá pole realizováno jako souvislý blok prvků v paměti počítače. Přístup k prvkům pole je zajištěn pomocí indexů, které určují pořadí prvku v poli. Jelikož jsou prvky bezprostředně za sebou, můžeme vypočítat adresu prvku na základě znalosti adresy začátku pole, velikosti jednoho prvku a indexu prvku.

Při vytváření pole se může stát, že v paměti není souvislé místo pro alokaci pole. V tomto případě je nutné provést před alokací pole defragmentaci paměti. Časové složitosti operací vlož a odeber jsou $O(1)$.

2.1.1 Statické pole

Vytvořením statického pole se alokuje potřebná paměťová oblast. Pokud bychom za běhu programu potřebovali rozměr pole změnit, neuspěli bychom. Při vytváření statického pole je nutné vhodně zvolit rozsah pole, aby nedošlo k jeho přetečení.

2.1.2 Dynamické pole

Některé programovací jazyky umožňují vytvářet takzvané „Dynamické pole“. Výhodou dynamického pole je, že je možné za běhu programu měnit jeho velikost. Změna velikosti pole je poměrně výpočetně náročná. Pokud by mělo docházet ke změně velikosti častěji, bylo by vhodné zvětšovat pole s větší rezervou. Například: Pokud bychom často zvětšovali pole o 5 prvků, bylo by efektivní jednou dopředu alokovat např. 100 prvků, než provádět alokaci dvacetkrát po pěti prvcích.

2.2 Lineární seznamy

Lineární seznam je dynamická datová struktura, která umožňuje seskupovat libovolný počet prvků různých datových typů. Počet prvků je omezen pouze velikostí dostupné paměti. Lineární seznamy mají oproti poli výhodu v tom, že nepotřebují dopředu alokovat souvislou oblast paměti. Při vkládání nového prvku do seznamu se alokuje paměťová oblast pouze pro vkládaný prvek a tím dochází k lepšímu využití fragmentované paměti bez nutnosti defragmentace. Přístup k prvkům seznamu je sekvenční, což je další odlišnost od pole.

2.2.1 Jednosměrný seznam

Základním stavebním prvkem jednosměrného seznamu je záznam nebo třída, jenž obsahuje referenci na další prvek. Je výhodné uchovávat adresy prvního a posledního prvku pro operace nad začátkem a koncem seznamu. Dále je vhodné uchovávat referenci na „aktuální(vybraný)“ prvek. Jelikož je seznam jednosměrný, nelze procházet seznam z konce na začátek. Přesun na začátek zajistí metoda `zpristupni_prvni()`. Poslední prvek má následníka nastaveného na NULL, toho lze využít při prohlídce seznamu v cyklu.

Rozšířením jednosměrného seznamu vzniká **obousměrný seznam**, který se skládá z prvků obsahujících referenci jak na následníka, tak i na předchůdce. Díky tomu se lze v seznamu pohybovat směrem dopředu i dozadu.

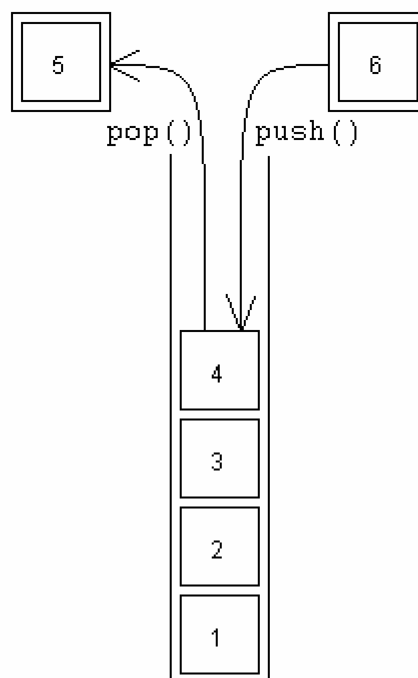
Lineární seznam je možné implementovat na **homogenním souvislém poli** nebo v **dynamické paměti**. Každá z těchto implementací má své výhody i nevýhody. Pokud bychom chtěli implementovat seznam, který nebude mít stálý rozměr, použijeme implementaci v dynamické paměti. V opačném případě použijeme jako implementující typ pole. Některé programovací jazyky sice umí rozměr alokovaného pole měnit, ale tato operace je poměrně výpočetně náročná.

Další možné modifikace jsou lineární seznamy **s hlavou** nebo **bez hlavy** a **cyklický** či **necyklický**. Cyklický lineární seznam má poslední prvek svázán s prvním. Pro obousměrný lineární seznam platí, že je mimo posledního prvku s prvním svázán i první s posledním.

Pro oba typy implementace dosahují pro operace vlož a odeber složitosti *O(1)*. Poněkud hůře je na tom operace prohlídka, která je závislá na počtu prvků a dosahuje lineární složitosti *O(n)* .

2.3 Zásobník

Zásobník je znám také pod akronymem LIFO (Last In First Out) v překladu „Poslední nejdříve". Zásobník funguje na stejném principu jako zásobník pistole, kde poslední vložený náboj bude vystřelen jako první. Naposled vložený prvek je označován jako vrchol zásobníku. Mezi klíčové operace patří metoda `push()`, která slouží pro vkládání prvků na vrchol zásobníku. Pro odebírání prvku z vrcholu zásobníku se používá metoda `pop()`. Poslední z klíčových operací je metoda `top()`, která zpřístupní vrchol zásobníku, ale neodebere jej. Grafické znázornění operací `push()` a `pop()` je na obrázku 1.



Obrázek 1: Zásobník

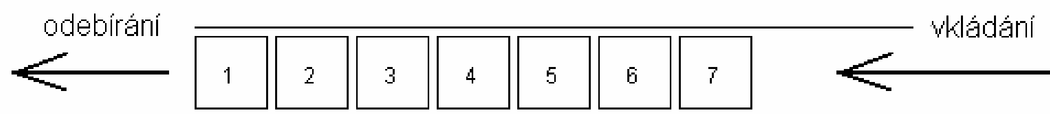
Jako implementující typ zásobníku můžeme využít lineární seznam, který se používáním specifických operací bude chovat jako zásobník. Bude se jednat zejména o operace `vlozNaKonec()`, `odeberZKonce()` a `zpristupniPosledni()`. Další možností je implementace zásobníku na poli. Tato implementace má však stejné

nevýhody jako implementace seznamu na poli. V některých případech je však konstantní velikost zásobníku žádoucí.

Operace `push()`, `pop()` a `top()` nezávisle na implementaci dosahují konstantní složitosti *O(1)*.

2.4 Fronta

Fronta je další hojně využívanou datovou strukturou. Její princip je shodný s frontou u přepážky v bance nebo frontou u pokladny. Akronymem pro frontu je FIFO (First In First Out). V překladu by se to dalo přirovnat k známému pořekadlu: „Kdo dřív přijde, ten dřív mele".



Obrázek 2: Fronta

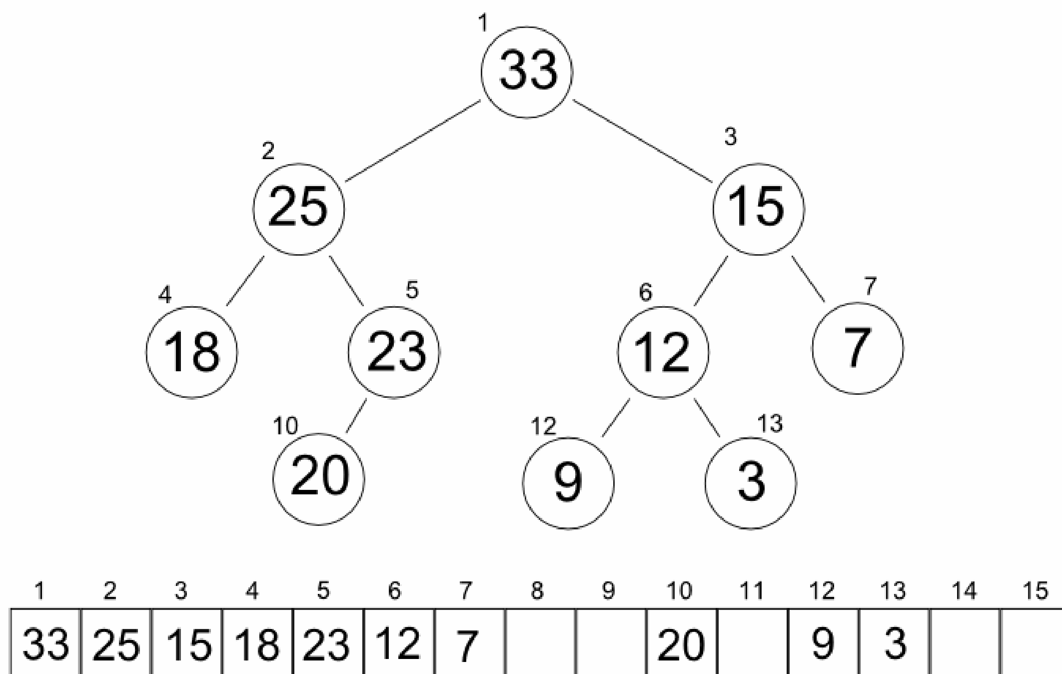
Jak je znázorněno na obrázku 2, vkládání je prováděno vždy na konec fronty pomocí metody `vlozNaKonec()` a odebírání ze začátku fronty metodou `odeberZeZacatku()`.

I fronta, stejně jako zásobník, se implementuje pomocí lineárního seznamu nebo pole. Při implementaci pomocí seznamu jsou využívány metody `VlozNaKonec()` a `OdeberZeZactku()`. Operace vkládání a odebírání dosahují konstantní složitosti *O(1)*.

2.5 Halda a prioritní fronta

Abstraktní datový typ prioritní fronta je fronta, kde je pořadí prvků dáno jejich prioritou. Tou může být např. čas jejich vložení do struktury. Poté bude podle typu implementace z fronty odebrán nejmladší (nejstarší) prvek. Prioritní frontu lze implementovat mnoha způsoby, mezi které patří různé implementace na poli a lineárním seznamu [1]. Nejvýkonnější implementací je však implementace na haldě, kde operace `vloz()` a `odeber()` dosahují logaritmické složitosti $O(\log_2 n)$. Halda je binární strom, pro který platí, že kořen je větší (menší), než jsou všechny ostatní uzly

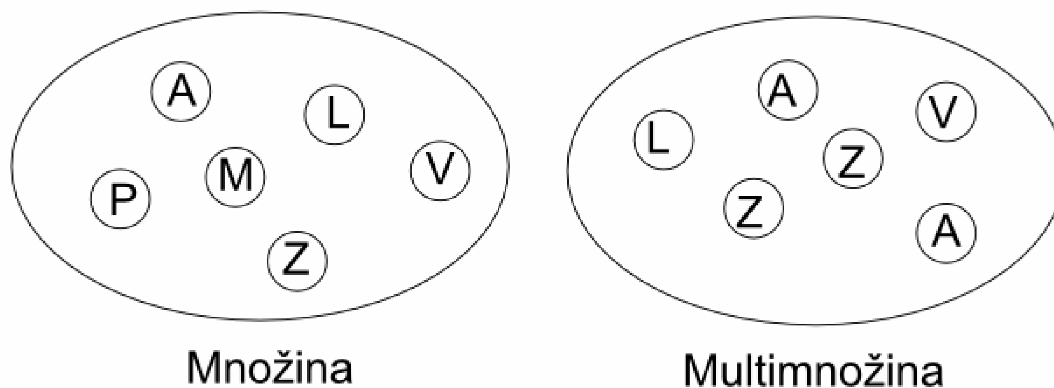
stromu. Jednotlivé uzly jsou větší (menší) jak jejich potomci. Další podmínkou je, že se listy liší maximálně o jednu úroveň. Halda se nejjednodušeji implementuje na poli, kde platí pravidlo, že synové prvku `i` jsou v poli umístěny na indexech `[2*i]` a `[2*i+1]`. Otec prvku `i` je umístěn na pozici `[i/2]`. Lze vyvodit, že se kořen v poli musí nacházet na indexu č.1. [2]. Na obrázku 3 je znázorněn princip haldy a její paměťová reprezentace v poli. Jednou z modifikací haldy je tzv. levostranná halda, kde jsou prvky vkládány od nejlevější větve směrem k nejpravější větvi. Díky tomu nebudou v poli vznikat mezery jako na uvedeném obrázku.



Obrázek 3: Halda

2.6 Množina

Množina obsahuje prvky stejného datového typu tzv. bázový typ. Uspořádání prvků v množině nemá žádné pevné pořadí. Jednotlivé prvky se mohou v množině vyskytovat nejvýše jednou. Jak je znázorněno na obrázku 3, je možný výskyt několika prvků se stejnou hodnotou, v takovémto případě se jedná o tzv. multimnožinu. Mezi množinami lze provádět operace sjednocení, průnik, rozdíl, podmnožina a rovnost.



Obrázek 4: Množina a multimnožina

Množina se implementuje za pomoci tabulky ADT.

2.7 Strom

Strom je hierarchickou datovou strukturou, která v informatice nabízí široké uplatnění. Základním stavebním kamenem stromu je uzel, který se podle umístění ve stromu může dělit na (viz. Obrázek 5):

- Kořen

  Uzel stromu, který se v hierarchické struktuře vyskytuje nejvýše. Z toho vyplývá, že kořen nemá žádné rodiče. Strom má pouze jeden kořen.

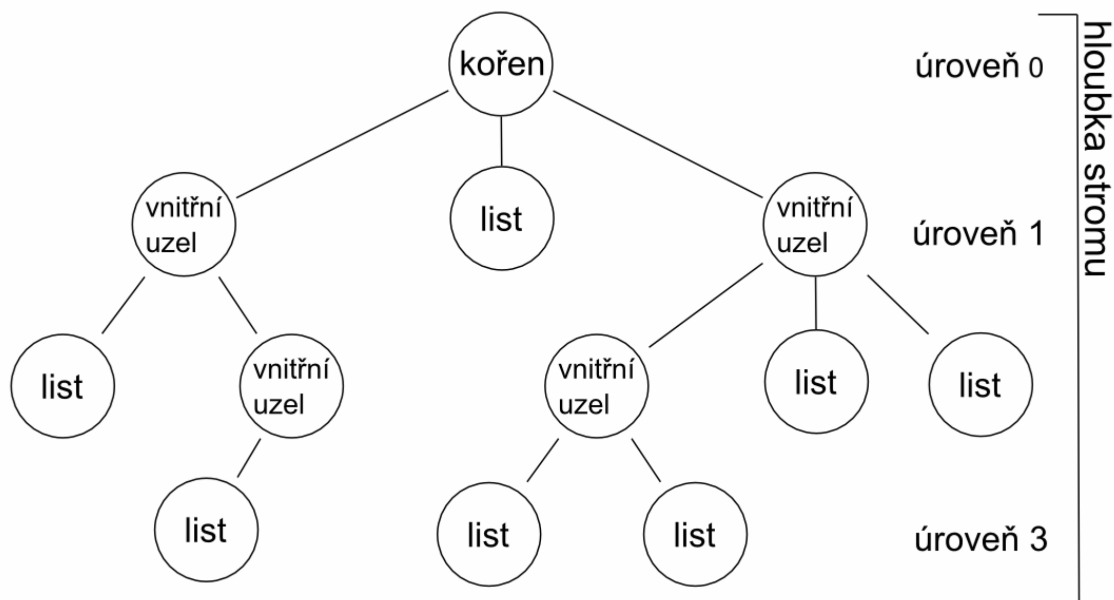
- Vnitřní uzel

  Je to uzel, který má rodiče a zároveň i potomka(y).

- List

  Jedná se o uzel, který nemá žádné potomky. Pokud má strom jen jeden uzel, je zároveň kořenem i listem.

Stromy mohou být utříděné a neutříděné. Utříděné stromy jsou ty, které mají potomky každého uzlu uspořádány. Tím je možné říci např. „Tento prvek je třetím potomkem tohoto otce“.



Obrázek 5: Strom

Pokud každý prvek stromu obsahuje referenci pouze na jediný prvek, jedná se o strom **unární.** Tímto prvkem však musí být otec daného prvku. Pokud by prvek obsahoval referenci na jeho potomka, jednalo by se o jednosměrný seznam.

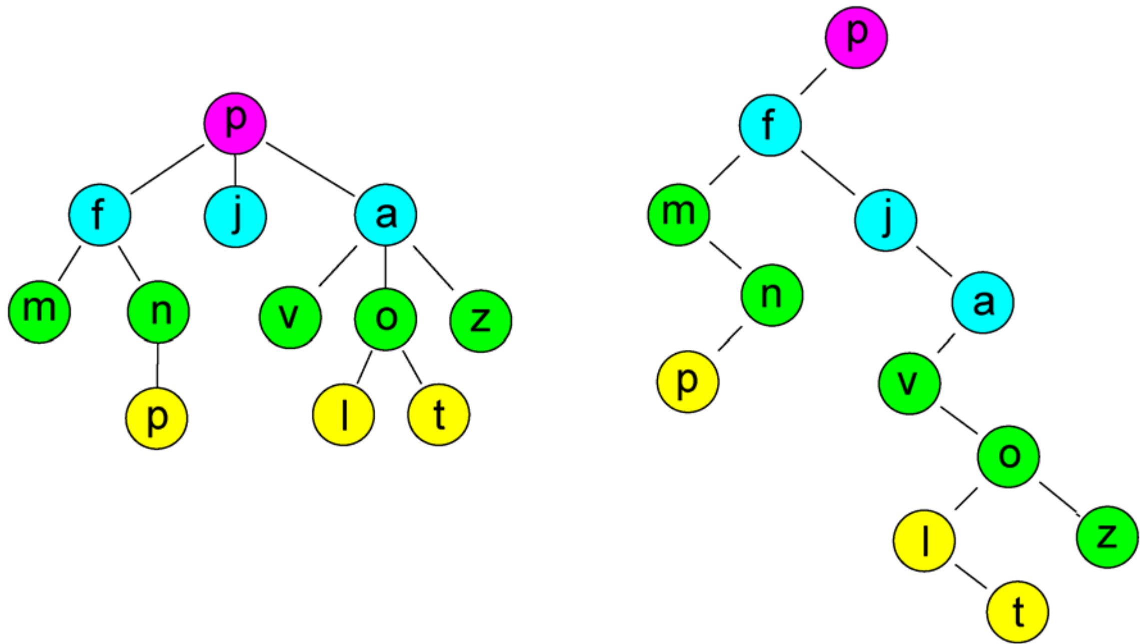
2.7.1 Binární strom

Binární strom patří mezi utříděné stromy. Každý prvek má maximálně dva syny a je rozlišován levý a pravý syn. Existují různé modifikace binárního stromu jako např. binární vyhledávací strom, levostranná halda, atd. K procházení binárního stromu slouží prohlídky preorder, inorder a postorder.

2.7.2 K-cestný strom

Může-li mít uzel stromu libovolný počet potomků, pak se jedná o K-cestný strom. Konstantu „K“ určuje počet synů daného uzlu, který má v rámci celého stromu synů nejvíce. Prohlídku stromu je možno provést jako prohlídku do šířky nebo do hloubky. První jmenovaná využívá jako pomocnou strukturu frontu a druhá zásobník.

Implementace je realizována transformací na binární strom. Transformace proběhne za následujících pravidel: levá reference binárního uzlu obsahuje referenci na svého prvního syna a pravá na následujícího bratra. Více na obrázku 6.



Obrázek 6: Transformace K-cestného stromu na binární

Pokud známe hodnotu „K“ můžeme strom implementovat pomocí uzlů obsahujících pole synů o rozměru „K“

2.8 Tabulka

Zřejmě každý z nás se v životě setkal s daty, která byla uspořádána do tabulky. Ty poskytují rychlý a přehledný přístup k informacím. Tabulka je uspořádaná množina, kde uspořádání určují klíčové hodnoty prvků. Příkladem může být tabulka na novém řidičském průkazu, kde jsou klíčovými hodnotami jednotlivé podskupiny řidičského oprávnění a daty je datum (den, měsíc a rok) složení závěrečné zkoušky. Takováto tabulka se nazývá **statická**, jelikož do ní nelze přidávat další řádky. **Dynamická** tabulka umožňuje přidávat další záznamy např. seznam objednaných pacientů u lékaře, kde klíčovým prvkem může být jejich rodné číslo.

Pro realizaci tabulky se nabízí početné množství implementací. Mezi nejjednodušší patří implementace na utříděném (neutříděném) poli nebo seznamu. Volba utříděnosti má vliv na časovou složitost klíčových operací.

Další možnou implementací je tabulka na binárním vyhledávacím stromu (BVS) nebo na implicitní kosočtvercové vyhledávací síti (KVS). BVS je binární strom, kde

levý potomek má klíčovou hodnotu menší jak jeho otec a pravý potomek ji má naopak větší jak jeho otec

Hashovací tabulka (pole - seznam) je jednou z nejvýkonnějších implementací tabulky. Adresa prvku v paměti je vypočítána přímo z jeho klíče pomocí hashovací funkce. Pokud je dvěma nebo více prvkům vypočítána stejná adresa, tvoří tyto prvky tzv. kolizní skupinu. Kolizní záznamy jsou zřetězeny pomocí seznamu.

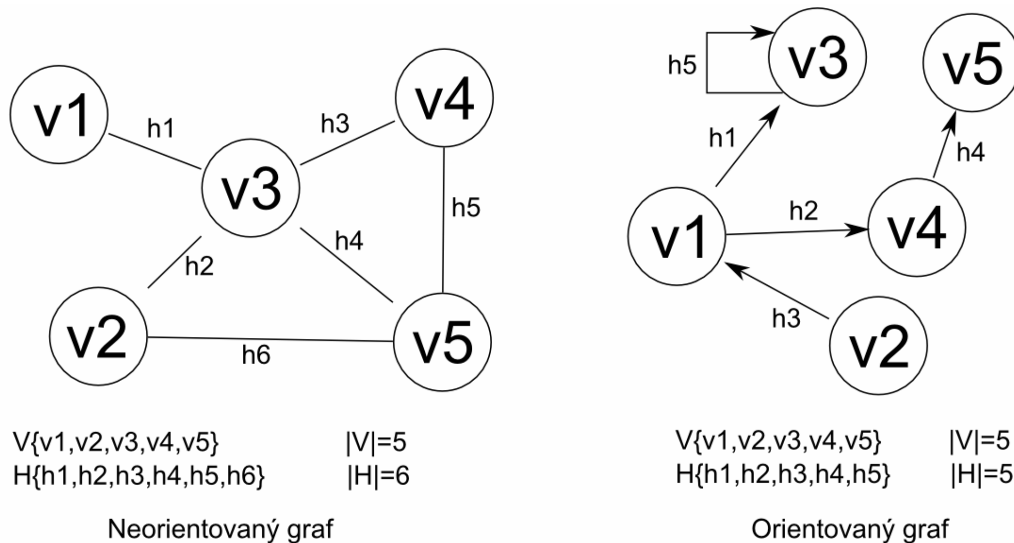
Implementace	**Vloz()**	**Odeber()**	**Najdi()**
Pole - utříděné	$O(n)$	$O(n)$	$O(log_2 n)$
Pole – neutříděné	$O(1)$	$O(n)$	$O(n)$
Seznam – utříděný	$O(n)$	$O(n)$	$O(n)$
Seznam – neutříděný	$O(1)$	$O(n)$	$O(n)$
BVS	$O(log_2 n)$	$O(log_2 n)$	$O(log_2 n)$
KVS	$O(n^{1/2})$	$O(n^{1/2})$	$O(n^{1/2})$
Hashovací tabulka	$O(1)$	$O(1)$	$O(1)$

Tabulka 1: Složitosti ADT tabulka

2.9 Graf

Graf je datová struktura skládající se z dvou odlišných tříd prvků, kterými jsou vrcholy a hrany.

Formálně můžeme zapsat, že existuje graf G=(V,H), kde V je množina vrcholů a H je množina hran grafu. Hrana je reprezentována jako dvojice vrcholů, se kterými je hrana incidentní. Orientovaná hrana je taková, kde platí, že hrana (u,v)≠(v,u). Mohutnost množiny vrcholů se označuje |V| a mohutnost množiny hran |H|.



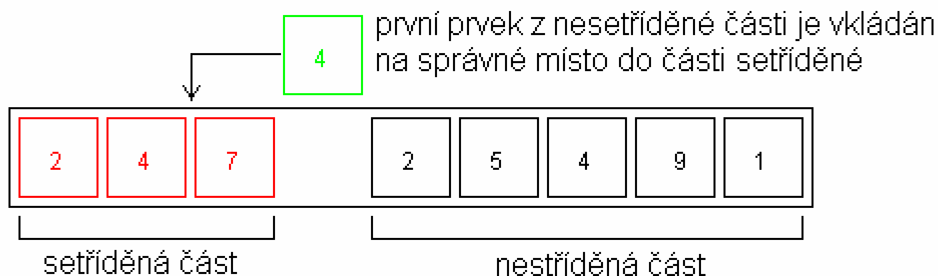
Obrázek 7: Typy grafů

3 Třídicí algoritmy

Představme si telefonní seznam, kde jsou záznamy seřazeny podle data připojení uživatele k telefonní síti. Pokud bychom hledali uživatele a znali pouze jeho příjmení, mohli bychom ho v seznamu hledat celý den. Z hlediska praktičnosti by bylo vhodnější data seřadit podle jiného klíče. V případě našeho seznamu by tímto klíčem bylo příjmení uživatele. Seřazení záznamů v seznamu bychom provedli pomocí třídícího algoritmu. Právě o třídících algoritmech pojednává tato kapitola [2].

3.1 InsertSort (třídění vkládáním)

S tříděním vkládáním se zřejmě setkal každý z nás. Třídění vkládáním je hojně používáno mezi hráči karetních her. Pokud hráč dostane karty tak, že žádné z nich nejsou setříděné, setřídí je následovně. První kartu si označí za setříděnou a prochází zbytek nesetříděné části. Každou kartu z nesetříděné části zařadí na správné místo v setříděné části. Po vložení do setříděné části se zvýší počet setříděných karet o jednu a počet nesetříděných se o jednu zmenší. Až dojde na konec, jsou karty setříděny. Složitost tohoto algoritmu je $O(n^2)$. Více na obrázku 8.

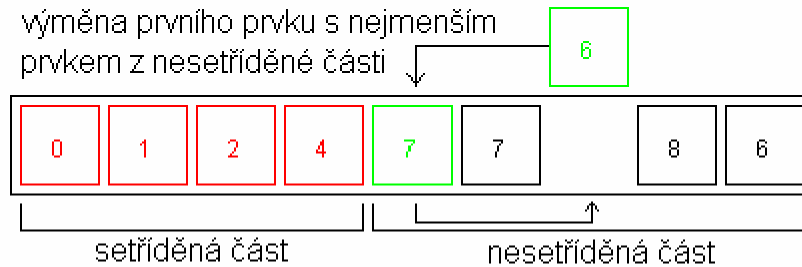


Obrázek 8: Třídění vkládáním

3.2 SelectSort (třídění výběrem)

Mějme pole nesetříděných prvků. Jejich setřídění bude provedeno takto. Na začátku si celé pole označíme za nesetříděné. Zapamatujeme si hodnotu prvního prvku v nesetříděné části. Poté procházíme nesetříděnou část a porovnáváme, zda-li se v ní nenachází prvek s menší hodnotou, než má první prvek z nesetříděné části pole. Pokud se v poli takový prvek nachází, je s prvním prvkem vyměněn. Nezávisle na tom, zda proběhne výměna, bude první prvek z nesetříděné části označen za setříděný. Tím se

zmenší rozměr nesetříděné části. Algoritmus se opakuje stále dokola, dokud není celé pole setříděno. Stejně jako u předcházejícího algoritmu je složitost $O(n^2)$. Algoritmus je zobrazen na obrázku 9.



Obrázek 9: Třídění výběrem

3.3 BubbleSort (třídění výměnou)

Jak je z názvu patrné bublinkové třídění je založeno na „probublávání“ prvků na jejich správnou pozici. Algoritmus pracuje následovně: Porovnáme prvek N s prvkem N+1. Pokud je prvek N větší jak N+1, tak tyto prvky vyměníme. Jestliže tomu tak není, pokračujeme dále inkrementací hodnoty N a znovu porovnáme. Až dojdeme na konec pole, pokračujeme znovu od začátku. Pole je setříděno tehdy, pokud nedojde k žádné výměně prvků. Na obrázku 10 je znázorněno vylepšené bublinkové třídění, které neporovnává již seřazené prvky. Tento algoritmu dosahuje složitosti $O(n^2)$.

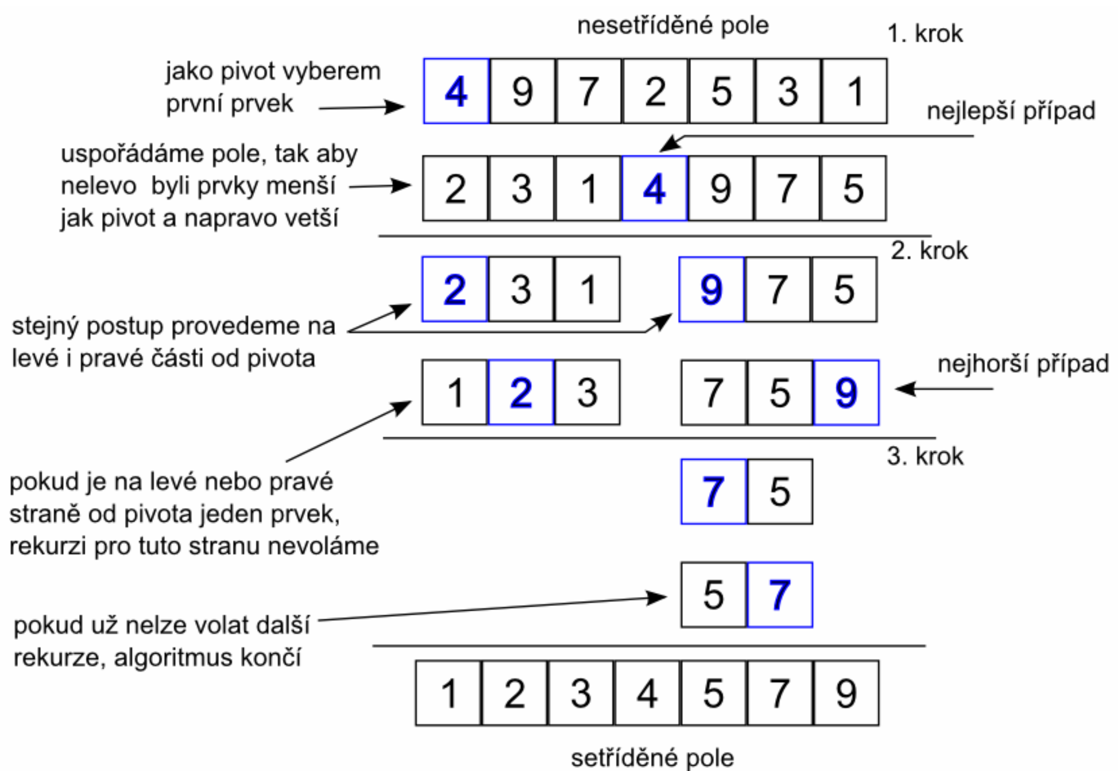
Obrázek 10: Bublinkové třídění

Výhodou těchto tří uvedených algoritmů je, že nepožadují dodatečnou pomocnou paměť. Takovéto algoritmy se označují jako „in site“, pracující na místě. Další uvedené algoritmy pracují rekurzivně nebo využívají pomocné struktury.

3.4 QuickSort (třídění rozdělováním)

QuickSort je jedním z nejrychlejších algoritmů třídění. Jedná se o algoritmus založený na metodě „rozděl a panuj“. Základem algoritmu je volba pivota. Pivot je

prvek, který po seřazení pole rozdělí pole na přibližně stejně velké části. Nalevo od pivota se nacházejí prvky menší jak pivot a napravo se nacházejí prvky větší. Nejlepší volbou pivota by byl medián tříděných hodnot. Výpočet mediánu by ale algoritmu na rychlosti nepřidal. Rychlejší, ale ne vždy vhodnou volbou pivota, je volba prvního, posledního nebo prostředního prvku. V nejlepším případě pivot rozděluje pole na poloviny, v tom případě dosahuje algoritmus složitosti $O(n \log_2 n)$. V nejhorším případě, kdy na jedné straně od pivotu nejsou žádné prvky, je složitost pro tento algoritmus $O(n^2)$. Tato vlastnost řadí quicksort mezi tzv. nestabilní algoritmy. To znamená, že je složitost algoritmu závislá na tříděné množině dat. Kroky algoritmu jsou zobrazeny na obrázku 11.



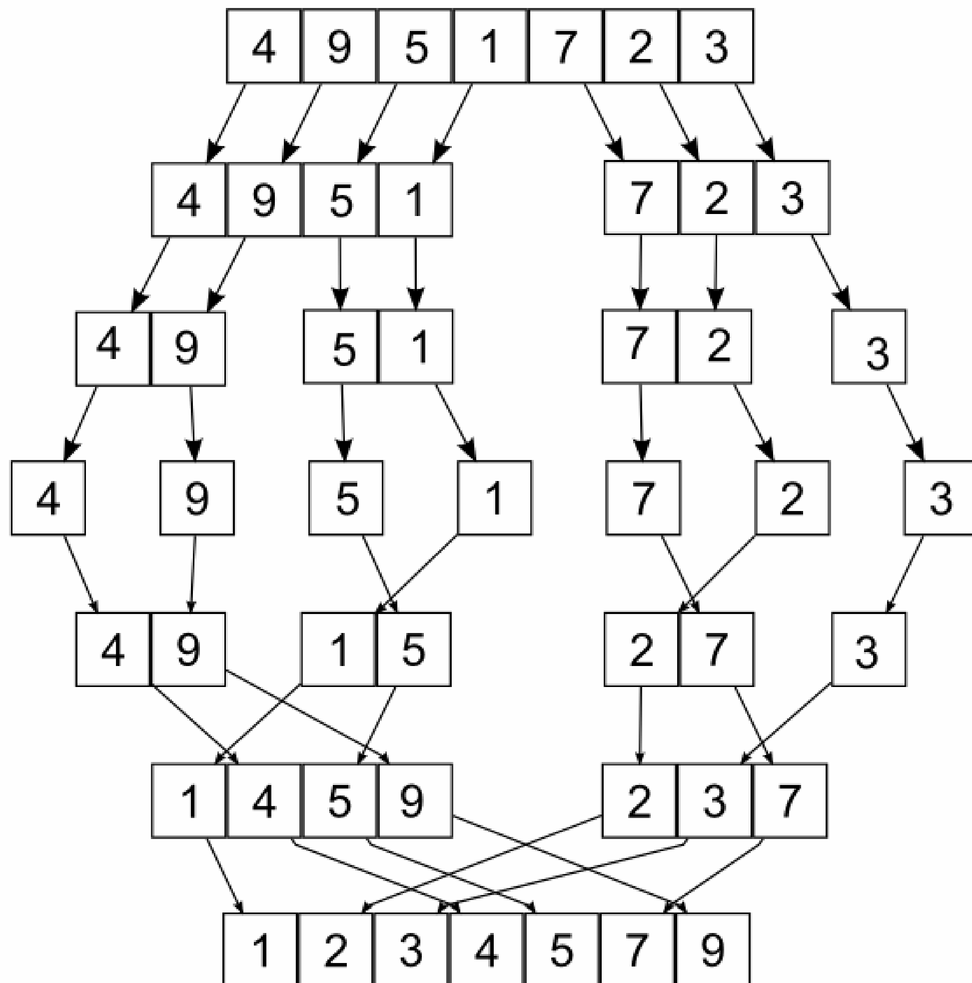
Obrázek 11: Algoritmus quicksort

3.5 MergeSort (třídění sléváním)

Autorem tohoto algoritmu je matematik John von Neumann, který je mimo jiné znám jako autor Von Neumannovi koncepce počítače. Tento algoritmus se řadí mezi stabilní algoritmy. Činnost algoritmu by se dala shrnout do třech základních kroků:

- rozdělení dat na přibližně stejně velké podmnožiny
- seřazení jednotlivých podmnožin
- sloučení podmnožin do jedné množiny.

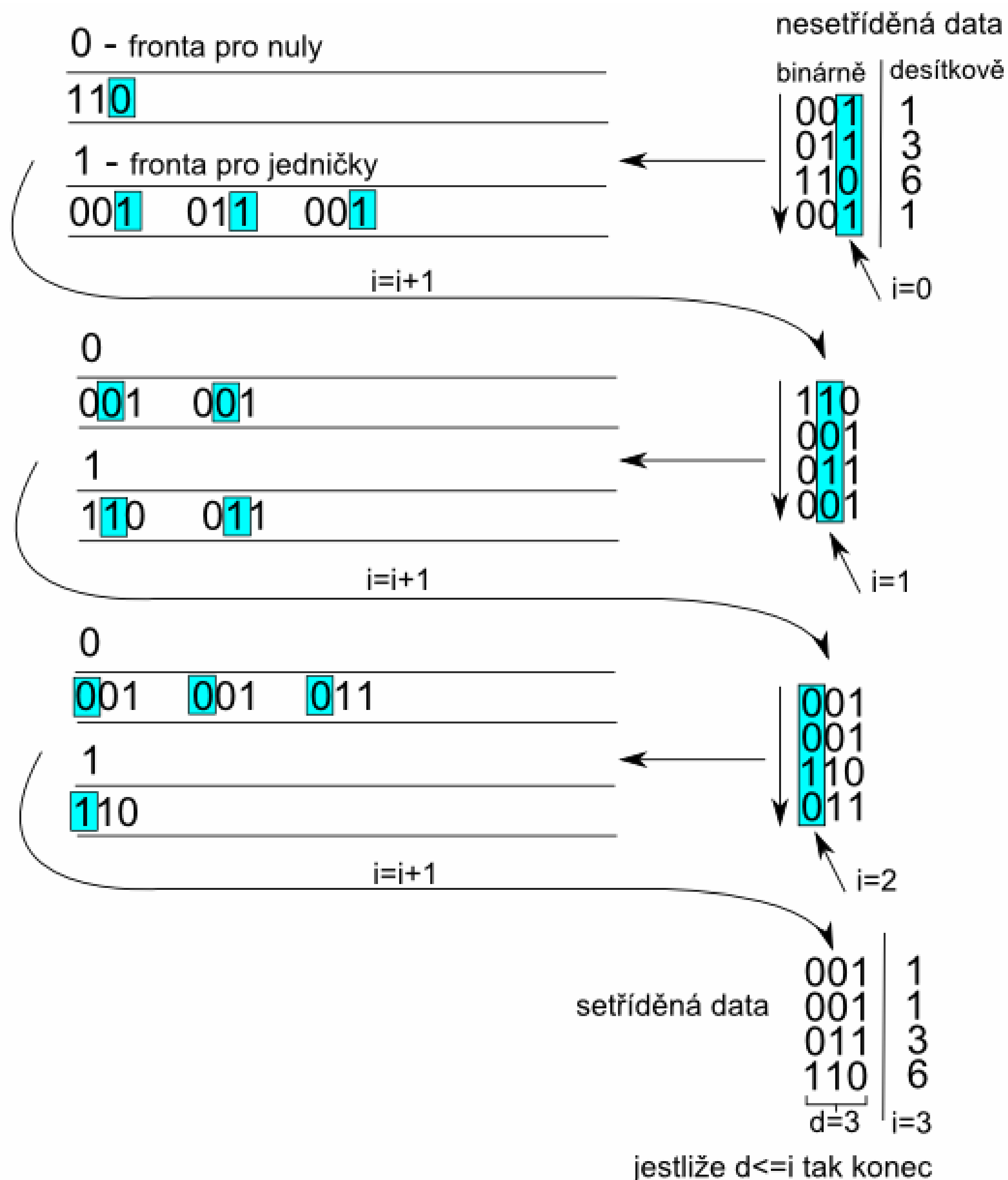
Tento algoritmus je realizován pomocí rekurze. Tříděná množina dat je rekurzivně dělena na poloviny tak dlouho, dokud nevzniknou jednotlivé jednoprvkové množiny, které se postupně začnou slévat do utříděné množiny prvků. Na obrázku 12 jsou znázorněny jednotlivé kroky algoritmu.



Obrázek 12: Mergesort - třídění sléváním

3.6 RadixSort (číslicové třídění)

Tento třídící algoritmus je někdy nazýván jako přihrádkové třídění. Radixsort se řadí mezi stabilní třídící algoritmy se složitostí *O(n)*. Pomocí tohoto algoritmu můžeme třídit nezáporná čísla ze všech číselných soustav. Kroky algoritmu jsou zobrazeny na obrázku 13.



Obrázek 13: RadixSort

Princip algoritmu je založen na procházení všech tříděných čísel a jejich zařazování do jednotlivých front podle čísel na jednotlivých řádech. Počet front je dán základem číselné soustavy tříděných čísel. Budeme-li třídit čísla v desítkové soustavě, použijeme deset front. Pro dvojkovou soustavu využijeme dvě fronty. Algoritmus bude probíhat v **d** krocích, kde **d** znamená počet řádů tříděného čísla. V každém kroku budou čísla zařazovány do front podle čísla umístěného na řádu **i**. Řád **i** označuje aktuální řád, který je každým krokem navyšován. Pokud se aktuální řád **i** rovná počtu řádů **d,** algoritmus skončí. Je-li množina tříděných dat taková, že existují čísla, která nemají stejný řád, pak jsou tato čísla doplněna zleva nulami.

4 Demonstrační applet

Úkolem mojí bakalářské práce bylo vytvořit applet, pomocí kterého bude možno graficky demonstrovat funkčnost vybraných datových struktur a třídících algoritmů. V této kapitole si popíšeme jak tento applet pracuje a jaké problémy spojené s tvorbou appletu bylo nutné řešit.

4.1 Java Applety

Java applet je malá aplikace napsaná v jazyce Java a umístěna na HTML stánku. Java applety tedy rozšiřují možnosti HTML stránek. Pomocí Java appletu můžeme webové stránky oživit grafikou nebo nějakou malou aplikací. Z hlediska bezpečnosti nemohou applety využívat všech možností, které Java poskytuje [3][4]. Applety nemohou:

- číst některé systémové proměnné
- zapisovat do souborů v počítači, kde je applet spuštěn
- navazovat síťové spojení na jiný než domovský server.

Tato omezení mohou být na různých JDK odlišné. Dále lze omezení nastavovat pomocí webového prohlížeče.

Applety na rozdíl od jiných aplikací neobsahují metodu `main()[4]`. Vstupním bodem do programu je totiž webový prohlížeč. Applet obsahuje metody, které webový prohlížeč volá při určitých událostech. Tyto metody je potřeba překrýt a přizpůsobit jejich chování našim požadavkům. Mezi pět nejčastěji překrývaných metod patří:

- `init()` – tuto metodu volá prohlížeč při prvním spuštění appletu. Nejčastěji se zde provádí inicializace objektů a proměnných používaných v appletu.
- `start()` – metoda `start()` je volána když prohlížeč znovu spouští applet, který byl pozastaven metodou `stop()`. Například pokud byl prohlížeč minimalizován a znovu maximalizován.

- `paint()` – tato metoda je volána nastane-li událost, která vyžaduje překreslení appletu. Například je-li přes applet přetaženo jiné okno nebo je okno prohlížeče přesunuto na jinou pozici.
- `stop()` – pokud je okno prohlížeče minimalizováno nebo probíhá jeho zavírání je zavolána metoda `stop()`.
- `destroy()` – metoda `destroy()` je volána bezprostředně před zavřením okna webového prohlížeče.

Překrýt stačí pouze první tři jmenované metody. Tedy metodu `init()`, `paint()` a `start()`.

4.1.1 Struktura appletu

Applet se značně podobá aplikaci, s tím rozdílem, že applet musí dědit po třídě `JApplet`, která je umístěna v balíčku `javax.swing`[4]. Tělo appletu, který nic nedělá, je následující:

```
import javax.swing.JApplet;
public class Pokus extends JApplet{
}
```

Aby applet poskytoval nějaký výstup, musíme překrýt výše uvedené metody. Pro vykreslení obdélníku bude stačit překrýt metodu `paint()`. Následný kód bude vypadat takto:

```
import java.awt.Graphics;
import javax.swing.JApplet;
public class Pokus extends JApplet{
  public void paint(Graphics g){
      g.drawRect(10,10,50,70);
  }
}
```

Tento applet po zavolání metody `paint()` vykreslí obdélník o zadaných rozměrech. Kreslení bude popsáno dále.

Kompilace appletu probíhá stejně jako kompilace jakékoliv jiné aplikace napsané v Javě. Pokud nepoužijeme kompilátor vývojového prostředí, který celý proces kompilace zautomatizuje, můžeme zvolit cestu příkazové řádky. Zdrojový kód uložíme

s extenzí `.java`. Zdrojový kód uložený v souboru `Pokus.java` zkompilujeme pomocí příkazu `javac Pokus.java` [5]. Program javac (Java compiler) se nachází ve složce bin ve složce s JDK. Výstupem kompilátoru bude soubor `Pokus.class`, který předáme HTML stránce.

4.1.2 Začlenění appletu do HTML stránky

HTML stránka bude používat soubor *.class, v našem případě Pokus.class, což jsou zkompilované zdrojové kódy appletu. Applet umístíme na stránku pomocí párového tagu `<applet>`, který napíšeme mezi tag `<html>` a `</html>` má následující syntaxi

```
<applet code="Pokus.class" width="100" height="100">
</applet>
```

Tag `<applet>` má tři základní atributy, kterými jsou:

- `code` – slouží pro určení souboru, který obsahuje požadovaný applet
- `width` – definuje šířku okna, ve kterém se bude applet zobrazovat
- `height` – udává výšku okna pro zobrazení appletu.

4.2 Kreslení v appletu

Kreslení se provádí pomocí metody `void paint(Graphics g)`. Tuto metodu musíme překrýt ve své třídě. Parametr `g` třídy `Graphics` je grafický kontext, na kterém se provádí kreslení. Třída Graphics nám poskytuje metody potřebné pro kreslení. Tyto metody budou popsány v následující podkapitole.

Dalšími užitečnými metodami pro kreslení jsou metody `repaint()` a `update()`. Metoda `repaint()` je několikanásobně přetížená a slouží k překreslení kreslící plochy. Pokud tuto metodu zavoláme bez parametrů, dojde k překreslení celého kreslícího plátna. Další možností je metodu zavolat s následujícím parametrem:

```
void repaint(long nejpozdeji);
```

Metodu `repaint(long nejpozdeji)` zavoláme v případě, že potřebujeme překreslit kreslící plátno, ale není důležité plátno překreslit okamžitě. Parametr tedy

udává dobu (v milisekundách), během které by se mělo plátno překreslit. Tuto metodu je výhodné použít programujeme-li náročnější aplikace na výpočet a grafické zobrazování informací není příliš důležité. Pokud bude čas procesoru plně využit, bude metoda `repaint()` čekat na jeho uvolnění maximálně však po dobu zadanou jako parametr. Další z překrytých metod je metoda:

```
void repaint(int x, int y, int sirka, int vyska);
```

Parametry této přetížené metody určují výřez kreslící plochy, ve kterém dojde k překreslení. Použití této metody má za následek zrychlení programu. Význam parametrů této metody je shodný s parametry metody `drawRect()`.

Metodu `repaint()` bychom neměli v našich třídách překrývat. Úkolem této metody je zavolat metodu `update(Graphics g)`.

Metoda `update()` nejprve překreslí kreslící plátno barvou pozadí, čímž vymaže obsah plátna. Dále zavolá metodu `paint()`, která zajistí vykreslení nového obrazce. Ne vždy je žádoucí, aby byla před kreslením vymazána kreslící plocha. Toho dosáhneme překrytím metody `update()`.

```
public void update(Graphics g){
   paint(g);
}
```

Tímto přetížením obejdeme fázi mazání kreslící plochy a můžeme malovat přes již nakreslené grafické objekty. Na závěr této podkapitoly bych chtěl uvést, že není nutné kreslit pouze v metodě `paint()`, ale kdekoliv, kde lze získat grafický kontext pomocí metody `getGraphics()`. Z vlastní zkušenosti to ale nedoporučuji, protože kreslení mimo metodu `paint()` může způsobit řadu problémů. Například při přetažení jiného okna přes kreslící plátno by se vymazal obsah plátna. Pokud použijeme kreslící metodu v metodě `paint()`, tak k tomu nedojde, protože při přetažení okna přes kreslící plátno bude vyvolána událost, která zavolá metodu `paint()` a plátno znovu překreslí. Pokud budeme mít v appletu ovládací prvky, jako třeba tlačítka, bude nutné zavolat metodu `paint()` předka pomocí `super.paint(g)`. Zavolání metody předka zajistí překreslení ovládacích prvků.

4.2.1 Kreslící metody

V této podkapitole budou popsány vybrané kreslící metody třídy Graphics[6]. Potomkem třídy Graphics je třída Graphics2D, která ji rozšiřuje o další Metody[7]. Tato třída zde však popsána nebude, protože pro naše potřeby postačuje třída Graphics. Popíšeme si zde metody potřebné pro kreslení základních grafických primitiv [8].

- **Vymazání plátna**

K vymazání obsahu kreslícího plátna můžeme použít metodu:

```
clearRect(int x1, int y1, int width, int height);
```

Tato metoda nakreslí obdélník vyplněný barvou pozadí. Parametry `x` a `y` udávají souřadnice levého horního rohu mazaného obdélníku. `Width` a `height` určují šířku a výšku mazaného obdélníku.

- **Úsečka**

```
drawLine(int x1, int y1, int x2, int y2);
```

Metoda drawLine slouží k nakreslení čáry mezi dvěma body. První bod se nachází na souřadnicích x1 a y1 a druhý bod na x2 a y2. Kreslení bude provedeno barvou popředí grafického kontextu.

- **Obdélník**

```
drawRect(int x1, int y1, int width, int height);
```

Tato metoda vykreslí obdélník bez výplně. Parametry mají stejný význam jako u metody `clearRect()`. Levá hrana je na pozici `x` a pravá na pozici `x+width`. Horní hrana je na pozici `y` a spodní na `y+height`. Pokud bychom chtěli vyplnit obsah obdélníku barvou, použijeme tuto metodu:

```
Fillrect(int x1, int y1, int width, int height);
```

Pro levou hranu se použije souřadnice `x` a pro pravou `x+width-1`. Podobný vzorec platí i pro výšku. Dále je nutno očekávat, že nakreslený obdélník nebude mít nakresleny okraje.

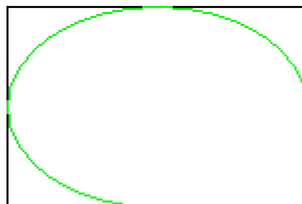
- **Kruhová a eliptická výseč**

```
drawArc(int x, int y, int width, int height, int startAngle, int
arcAngle);
```

Pomocí této metody můžeme nakreslit kruhový nebo eliptický oblouk vepsaný zadanému obdélníku. První čtyři parametry jsou stejné jako u obdélníku a určují obdélník, kterému bude kružnice vepsána. Parametr `startAngle` označuje počáteční úhel, od kterého se začne vykreslovat. Na obrázku 14 je použit úhel 0, který se nachází uprostřed levé hrany obdélníku. Posledním parametrem je parametr `arcAngle`. Tento parametr říká, kolik stupňů od úhlu zadaného parametrem `startAngle` se má vykreslit. Na obrázku je hodnota parametru `arcAngle` 270. Poslední dva parametry lze zadávat jako záporné hodnoty. V tomto případě potom bude směr vkreslování opačný. Výsledné rozměry kružnice nebo elipsy budou pro šířku `width+1` pixelů a pro výšku `height+1` pixelů. Uvedený obrázek byl nakreslen pomocí těchto příkazů:

```
g.setColor(Color.black);
g.drawRect(50,150,150,100);
g.setColor(Color.green);
g.drawArc(50,150,150,100,0,270);
```

Obdélník zde byl vykreslen pouze pro názornou ukázku.



Obrázek 14 - ukázka drawArc()

Pro vyplnění kruhové výseče slouží metoda `fillArc()` se stejnými parametry jako má `drawArc()`. Vyplněna bude oblast ohraničená kruhovou výsečí a úsečkami vedených z počátečního a koncového bodu do středu výseče.

- **Kruh a elipsa**

```
drawOval(int x1, int y1, int width, int height);
```

Tato metoda je podobná metodě `drawArc()` , ale neobsahuje poslední dva parametry. Z toho vyplývá, že metoda slouží pouze pro kreslení elips a kružnic

vepsaných obdélníku zadaného čtyřmi parametry. Pro nakreslení oválu vyplněného barvou můžeme použít metodu `fillOval()`.

- **Mnohoúhelník**

```
drawPolygon(int[] xPoints, int[] yPoints, int nPoints);
```

Jak název napovídá, tato metoda je určena pro kreslení mnohoúhelníků. Souřadnice bodů `x` a `y` jsou uloženy v polích, které jsou do metody předávané jako parametry `xPoints`. a `yPoints`. Poslední parametr `nPoints` udává počet bodů, ze kterých se mnohoúhelník skládá.

- **Text**

```
drawString(String str, int x, int y);
```

Někdy budeme potřebovat vepsat na kreslící plátno text. K tomu slouží výše uvedená metoda. Prvním parametrem je text, který se má vypsat. Druhý a třetí parametr určují souřadnice, kde bude vypisovaný text umístěn.

- **Obrázek**

```
drawImage(Image img, int x, int y, ImageObserver observer);
```

Touto metodou bude na kreslící plátno vykreslen obrázek třídy Image, který je do metody předán jako první parametr. Parametr `x` a `y` jsou souřadnice na kreslícím plátně, kde bude vykreslen levý horní roh obrázku. Poslední parametr `observer` rozhraní `ImageObserver` je objekt přijímající zprávy o stavu kresleného obrázku[9]. Tato metoda je několikanásobně přetížena, její další varianta je následující:

```
drawImage(Image mg,int dx1, int dy1, int dx2, int dy2,
int sx1,int sy1, int sx2, int sy2, ImageObserver observer);
```

Pokud byl předcházející metodě `drawImage()` předán obrázek větší jak kreslící plátno, byl vykreslen pouze jeho levý horní roh. Tento problém řeší tato metoda, která může kreslený obrázek zmenšit nebo zvětšit. První a poslední parametr je shodný s předchozí metodou, takže je nebudu popisovat. Zajímat nás budou parametry souřadnic. Parametry `dx1` až `dy2` jsou souřadnicemi levého horního a pravého dolního rohu výřezu kreslícího plátna, na který se kreslí obrázek. Parametry `sx1` až `sy2` jsou potom souřadnicemi zdroje(s=source), tj. souřadnice vykreslovaného obrázku. Pokud

bude zadaný obdélník kreslícího plátna menší jak rozměry obrázku, dojde ke zmenšení obrázku. V opačném případě se obrázek zvětší.

4.2.2 Double buffering

Při programování grafické aplikace můžeme narazit na nepříjemné blikání a pomalé vykreslování grafiky. Tento problém řeší tzv. double buffering [10]. Tato vykreslovací technika spočívá v kreslení obrázku do paměti počítače a následném zkopírování na obrazovku. Obrázek se vytvoří pomocí metody `createImage()`, která vrací objekt typu `Image`.

Kreslení provádíme na grafický kontext objektu třídy `Image`, který získáme metodou `getGraphics()`. Poté, co je kreslení do bufferu dokončeno, přeneseme obrázek na obrazovku pomocí metody `drawImage()` popsané dříve.

4.3 Vlákna

Zřejmě každý programátor, který programoval grafické uživatelské prostředí(GUI) se při zpracovávání výpočetně náročné úlohy setkal se zamrznutím GUI. Došlo k tomu z důvodu, že všechen procesorový čas přidělený procesu aplikace byl využit na výpočet náročné úlohy, kterou může být například vykreslování animace. Tento problém řeší vlákna. Za pomocí vláken můžeme proces aplikace rozdělit na jednotlivé podprocesy (vlákna), které zdánlivě běží paralelně. Paralelismus se zajistí cyklickým přidělováním procesorového času jednotlivým vláknům. Navenek se toto střídání jeví jako jejich paralelní běh [3]. V Java aplikacích se o přidělování času procesoru jednotlivým vláknům stará Java Virtual Machine(JVM).

Vlákna je možné v jazyku Java implementovat pomocí třídy `Thread` nebo rozhraní `Runalble`. Při programování vícevláknové aplikace musíme dědit z této třídy. V odvozené třídě budeme nuceni překrýt metodu `run()`, do které napíšeme kód, jenž se má paralelně zpracovat. Takto může vypadat třída podporující vlákna:

```
class void Pocitadlo extends Thread{

public Pocitadlo(){
}
```

```
public void run(){
  for (int i = 0; i < 10; i++) {
    System.out.println(i);
  }
}
}
```

Po vytvoření instance dané třídy se vlákno spustí metodou `start()`, kterou naše třída zdědila od třídy `Thread`.

```
Pocitadlo pocitadlo = new Pocitadlo();
Pocitadlo.start();
```

Metoda `start()` ještě před spuštěním metody `run()` provede jisté „administrativní“ úkony spojené vytvořením nového vlákna a poté zavolá metodu `run()` v nově vytvořeném vláknu. Pokud bychom zavolali přímo metodu `run()`, tak by se sice tato metoda provedla, ale nebyla by spuštěna jako nové vlákno.

Další možností vytvoření vlákna je pomocí rozhraní `Runnable`. Této možnosti využijeme v případě, že naše třída již po nějaké třídě dědí. Jak je známo, tak Java nepodporuje vícenásobnou dědičnost, tak bychom s děděním po třídě `Thread` neuspěli. Nyní si ukážeme, jak vytvořit třídu pracující s vlákny pomocí rozhraní `Runnable`:

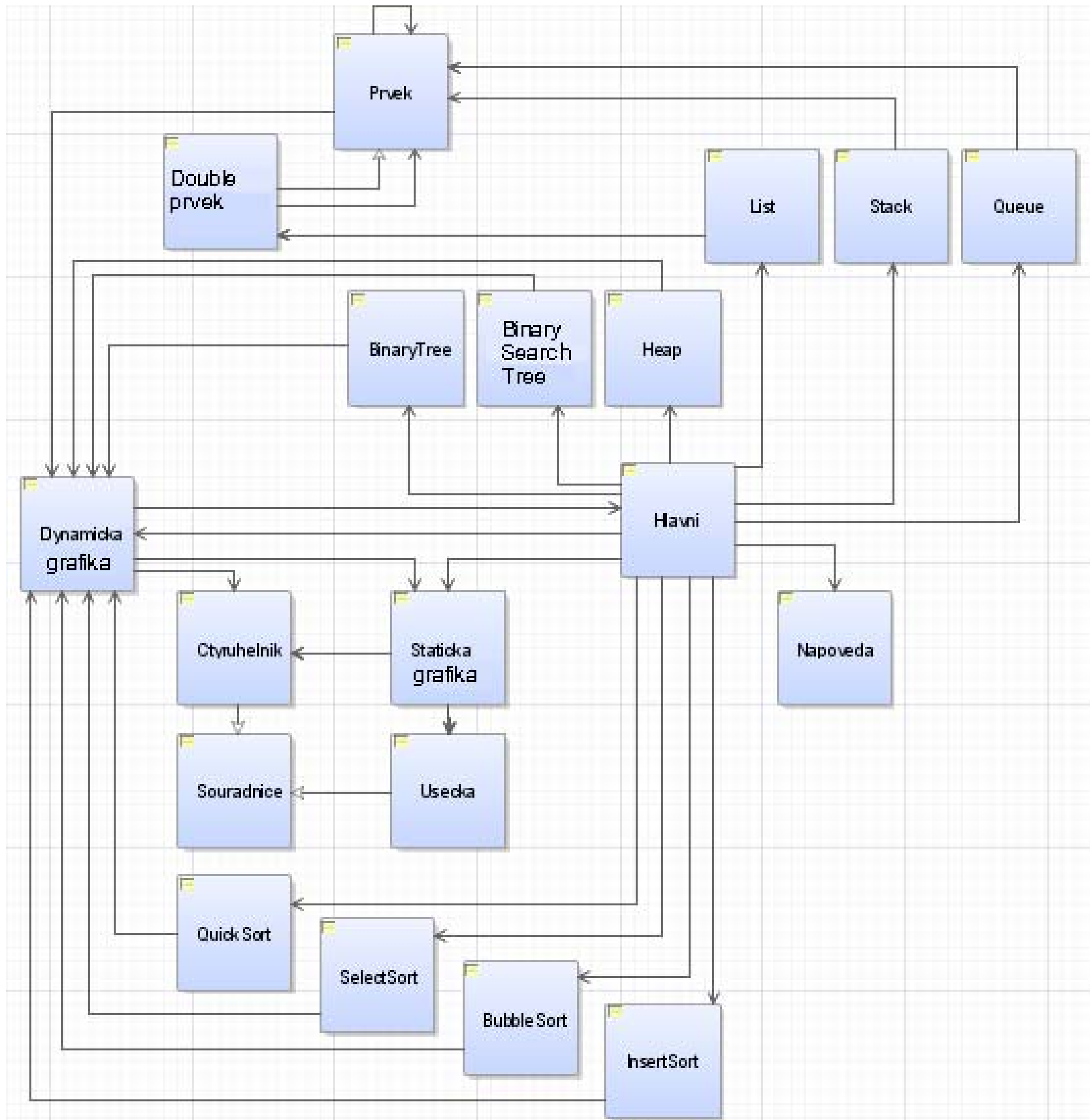
```
class void Pocitadlo implements Runnable{
  public Pocitadlo(){
  }
  public void run(){
    for (int i = 0; i < 10; i++) {
      System.out.println(i);
    }
  }
}
```

Vytvoření instance a spuštění vlákna této třídy bude mírně odlišné od předcházejícího postupu:

```
Pocitadlo pocitadlo = new Pocitadlo();
Thread vlakno = new Thread(pocitadlo);
vlakno.start();
```

Jelikož rozhraní Runnable neposkytuje metodu `start()`, musíme vytvořit objekt třídy `Thread`, kterému pomocí konstruktoru předáme instanci třídy, implementující rozhraní `Runnable`. Následně vlákno spustíme metodou `start()`.

4.3.1 UML diagram

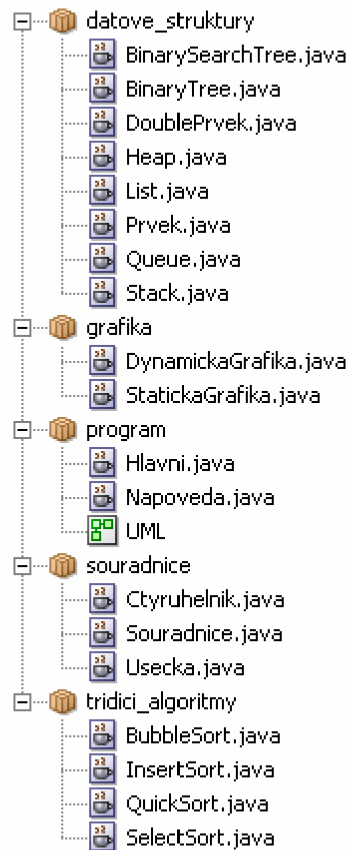


Obrázek 15 - UML diagram

Na obrázku číslo 15 je zobrazen UML diagram aplikace. Z obrázku je patrné, že většina tříd je asociována s třídou `DynamickaGrafika`, která je zodpovědná za zobrazování animací. Třída `StatickaGrafika` slouží k vykreslení nepohyblivé grafiky, jako jsou rámečky a ostatní pomocná grafika potřebná k lepšímu znázornění animace.

4.3.2 Použité třídy

Na obrázku 16 je zobrazena struktura aplikace. Jednotlivé třídy jsou logicky členěny do balíčků. Dále si popíšeme jednotlivé balíčky a třídy v nich obsažené.



Obrázek 16: Přehled použitých tříd

Prvním balíček nese název `datové_struktury`. Tento balíček obsahuje třídy implementující datové struktury využívané appletem. Třídy `Queue` a `Stack` implementují datové struktury fronta a zásobník. Obě tyto třídy využívají jako základní stavební kamen třídu `Prvek`. Třída `List` implementující lineární seznam naopak používá třídu `DoublePrvek`. Třídy BinaryTree, Heap a BinarySearchTree jsou implementací binárního stromu, levostranné haldy a binárního vyhledávacího stromu. Tyto dvě datové struktury jsou implementovány za pomoci statického pole. Realizace v dynamické paměti by pozbývala svých výhod, jelikož je znám maximální počet prvků u obou struktur.

Datové struktury zásobník a fronta lze v jazyku Java naprogramovat pomocí kolekce `LinkedList`, která je reprezentací seznamu. Požadované chování

nasimulujeme použitím odpovídajících metod. Pro zásobník `addLast()`a `removeLast()` nebo pro frontu `addLast()` a `removeFirst()`.

Dalším balíčkem je balíček `grafika`. Obsahuje třídy odpovídající za vykreslování animace. Třída `DynamickaGrafika` slouží k zobrazování pohyblivých grafických objektů. Obsahuje příslušné metody zajištující pohyb grafických prvků. Třída `StaticksGrafika` má na starosti vykreslování nepohyblivých objektů jako je např. ohraničení grafického znázornění datových struktur, třídicích algoritmu, atd.

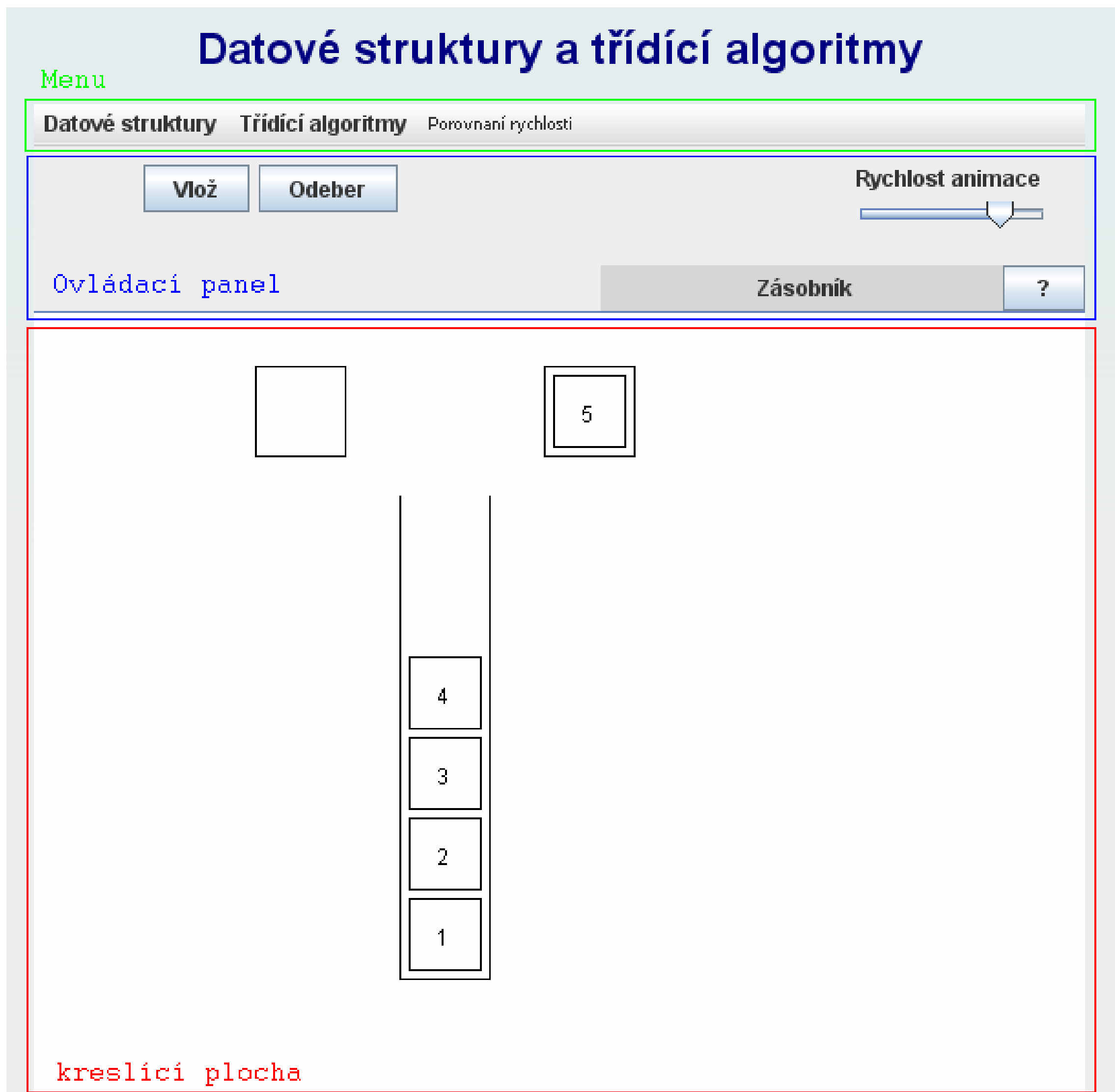
Balíček `program` obsahuje třídu `Hlavni`, která dědí po třídě `JApplet.` V této třídě je implementováno GUI pro ovládání aplikace. Pomocí této třídy jsou propojeny datové struktury s grafikou a spouštěny animace třídících algoritmů. Druhou třídou v tomto balíčku je třída `Napoveda`. Ta dědí po třídě `JDialog` a slouží pro zobrazení nápovědy v novém okně.

Dalším důležitým balíčkem je balíček `souradnice`. V tomto balíčku jsou třídy, jejichž instance slouží k uchovávání souřadnic grafických objektů. Z těchto objektů jsou získávány informace pro výpočet trajektorie animovaného grafického prvku.

Posledním balíčkem jsou `tridici_algoritmy`. Zde jsou implementovány jednotlivé třídicí algoritmy. Implementovány jsou algoritmy bubblesort, insertsort, selectsort a quicksort. Výkonná část těchto algoritmů je provázána s třídou `DynamickaGrafika` z balíčku `grafika`, pomocí které jsou vykreslovány jednotlivé kroky algoritmů.

Na obrázku 16 je zobrazen applet s částí HTML stránky. Na obrázku je zvýrazněno rozvržení aplikace. Pro výběr jednotlivých datových struktur a třídících algoritmů je v aplikaci použito menu. Menu je rozděleno na tři části. První částí je položka menu s názvem „Datové struktury“. Pod tímto menu se nacházejí tyto položky: „Fronta, Zásobník, Seznam, Binární strom, Levostranná halda“. Položky z tohoto menu slouží k grafické demonstraci funkčnosti vyjmenovaných datových struktur. Další částí menu je položka „Třídící algoritmy“, kde je možné vybrat jednotlivé třídící algoritmy. Poslední částí menu je položka „Porovnání rychlosti“, kde se porovnává rychlost vyladěného třídicího algoritmu quicksort používaného v jazyku Java, jako algoritmus

pro třídění polí s algoritmem prezentovaným na internetu [11]. Měření probíhá tak, že se vytvoří dvě stejně velké pole a obě se naplní stejnými náhodně vygenerovanými daty. Poté se před a po provedením jednotlivých algoritmů uloží aktuální čas v nanosekundách. Na základě těchto časů se vypočítají délky trvání jednotlivých algoritmů.



Obrázek 17 - Rozvržení grafického rozhraní

Jak je vidět na obrázku 17, ovládací panel obsahuje tlačítka a jiné ovládací prvky potřebné pro ovládání animace. Animaci lze zrychlovat a zpomalovat pomocí posuvníku umístěného v pravém horním rohu ovládacího panelu. Dalšími možnostmi jsou pozastavení a opětovné spuštění animace. Této funkce je možno využít pro pozastavení animace a následnému popsání fáze v jaké se algoritmus nachází.

4.4 Vlastní řešení

Při vývoji grafické aplikace se můžeme setkat s různými problémy, které je pro správný chod aplikace nutno vyřešit. Dále si popíšeme s jakými problémy jsem se setkal a jak jsem je řešil. Jedná se zejména o problémy s uživatelským rozhraním a vykreslováním grafiky.

4.4.1 Chybné vykreslování grafiky

Při vývoji této aplikace jsem se setkal s více chybami vykreslování. První chybou bylo vymazání kreslícího plátna, pokud přes něj bylo taženo jiné okno. Druhou chybou bylo trhané vykreslování a problikávání animace, kdy za sebou animovaný objekt zanechával stopu.

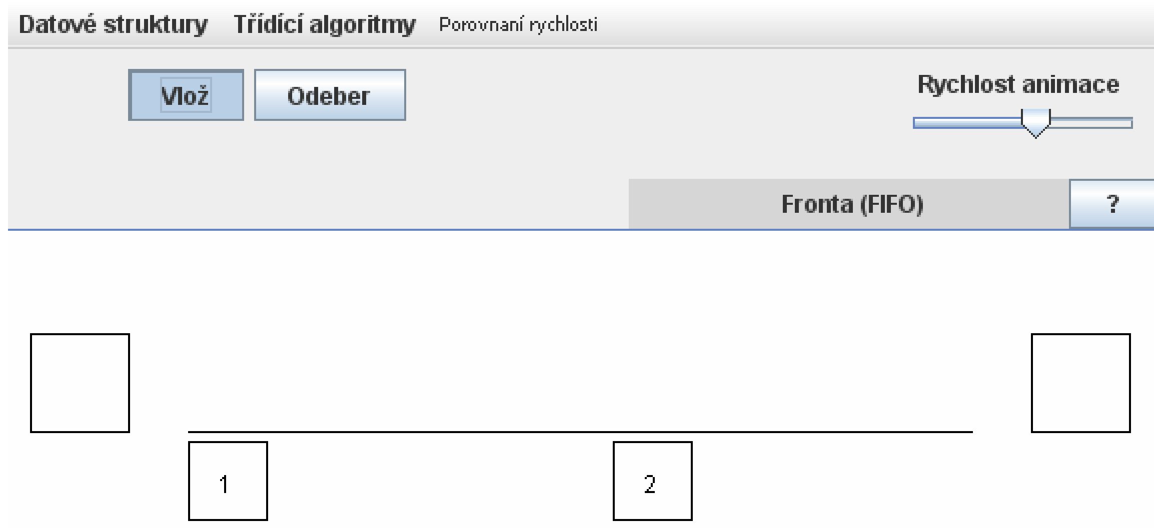
Všechny tyto chyby jsem vyřešil použitím metody zvané „double buffering“, která byla popsána v kapitole 4.2. Přemazávání kreslícího plátna bylo odstraněno vykreslením bufferu v metodě `paint()`, která je automaticky zavolána pokud je potřeba applet překreslit. Vyvolání metody `paint()` způsobí například ono zmiňované okno, které je taženo přes applet. Pokud do metody `paint()` vložíme pouze vykreslení bufferu, vznikne nový problém, kterým je nepřekreslení ovládacích prvků. Tento problém lze vyřešit tak, že do metody `paint()` přidáme příkaz na překreslení předka, a tím dojde překreslení ovládacích prvků. Tímto příkazem je příkaz:

```
super.paint(g);
```

4.4.2 Zajištění citlivého uživatelského rozhraní

Vykreslování animace může plně využít procesorový čas přidělený aplikaci. Důsledkem toho nezbude procesorový čas na obsloužení grafického uživatelského rozhraní (GUI). Po dobu animace není možno s uživatelským rozhraním pracovat. Při delších animacích by to bylo zcela nepřípustné. Jak je vidět na obrázku 18 aplikace má v průběhu animace stále stisknuté tlačítko Vlož. To je způsobeno tím, že po stisku tlačítka byla spuštěna animace, která konzumuje veškerý procesorový čas přidělený aplikaci. Tím nemůže dojít k přepnutí tlačítka do správného tvaru. Tlačítko bude přepnuto až po dokončení animace.

Tento problém vyřeší vícevláknové zpracování aplikace. Aplikace napsaná v jazyku Java má vždy alespoň jedno vlákno a tím je vlákno samotné aplikace. Tímto vláknem může být např. vlákno obsluhující GUI.



Obrázek 18 - Zablokované GUI

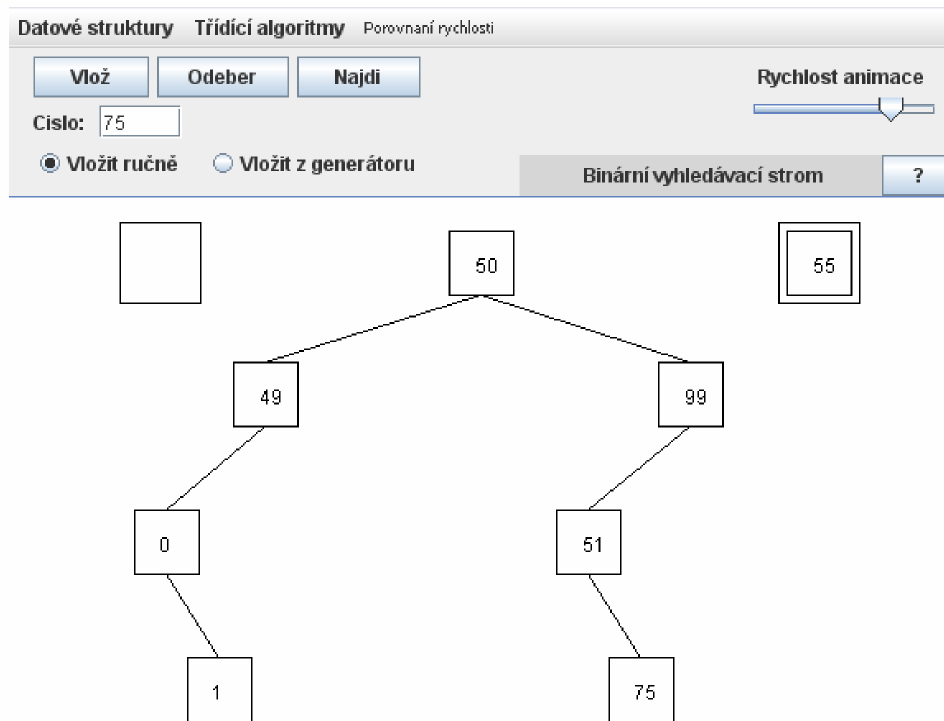
Všechny animace třídění jsem umístil do vlákna paralelního k vláknu aplikace. Tím je zajištěna možnost ovládat applet během animace. Aby mohla animace běžet jako vlákno, musí její třída dědit po třídě `Thread` nebo musí implementovat rozhraní `Runnable`. V obou případech je nutné překrýt metodu `run()`, do které napíšeme kód, který chceme zpracovat v novém vlákně. Pro třídící třídy jsem vybral variantu implementace rozhranní `Runnable` [12].

Implementaci rozhraní `Runnable` jsem vybral z důvodu, že Java nepodporuje vícenásobnou dědičnost. Pokud bych použil variantu dědění po třídě `Thread`, tak by v případě potřeby, nebylo možné dědit po jiné třídě.

Pokud běží vlákno animace paralelně k vláknu GUI, tak by bylo vhodné aby šla animace pozastavit. K tomu jsem využil třídy `DynamickaGrafika`, jejíž instance jsou využívány jako tříděné prvky. Do této třídy jsem přidal atribut `pozastavit` typu `boolean`. Pokud má být instance třídy `DynamickaGafika` překreslena, tak se překreslí pouze v případě, že je atribut `pozastavit` nastaven na `false`. V opačném případě se v cyklu čeká dokud nebude atribut nastaven na `false`.

4.4.3 Problém určení plnosti vyhledávacího stromu

Z důvodu grafického vyjádření obsahuje binární vyhledávací strom (BVS) pouze čtyři úrovně. Důsledkem toho se strom může strom obsahovat pouze patnáct prvků. V některých případech může dojít k zaplnění stromu dříve, než je vloženo všech patnáct prvků. Na obrázku 20 je zobrazen stav stromu, kdy do něho nelze vkládat další prvky. Problém nastává, když potřebujeme zjistit zdali lze do stromu vkládat další prvky.



Obrázek 19 - Problém určení grafického znázornění BVS

Pro zjištění plnosti stromu je nutno zjistit, zda-li jsou ve stromu prvky, které nesplňují jakoukoli z uvedených podmínek:

- Prvek se nesmí rovnat 0, jinak nebude možno vkládat levé potomky.
- Dále se nesmí rovnat 99, poté by nebylo možné vkládat pravé potomky.
- Rozdíl hodnoty prvku a hodnoty jeho otce se nesmí rovnat -1, pokud ano je pravá strana prvku nepřístupná. Pokud se rozdíl bude rovnat 1, nebude možné vkládat levé potomky prvku.
- Rozdíl hodnot prvku a jeho prapředka se nesmí rovnat 1 nebo -1. Pokud ano platí stejné pravidlo jako v předcházející podmínce.

Pokud nějaký prvek stromu nesplní alespoň jednu z těchto podmínek, strom ještě není plný. Prvky v poslední úrovni kontrolovány nejsou, neboť ji nelze vkládat žádné potomky.

Závěr

Mým úkolem bylo vytvořit applet, který by měl pomocí animace ulehčit výklad vybraných datových struktur a třídících algoritmů.

Aplikaci jsem se snažil navrhnout tak, aby ji bylo možné dále rozšiřovat. Grafické znázornění jsem nevztahoval ke zdrojovému kódu algoritmu, kde by byl v určité části animace zobrazován obsah proměnných a část zpracovávaného zdrojového kódu. Zaměřil jsem se zejména na to, aby byla animace co nejednodušší a nejsrozumitelnější. Applet by měl sloužit pro doplnění výkladu vyučujícího. Vyučující má možnost měnit rychlost animace a pozastavovat její průběh pro důkladné vysvětlení algoritmu.

Jako nejsložitější považuji řešení animace binárního vyhledávacího stromu. Obtížné bylo určit, zda-li strom již není plný, respektive zda do něho lze vložit další prvek. Dalším problémem bylo vykreslení reorganizace stromu po odebrání prvku ze stromu. Viditelnou chybou bylo špatné vykreslování animace, které jsem odstranil použitím vykreslovací metody zvané „double buffering". Dále bylo nutné řešit vícevláknové zpracování aplikace, aby v průběhu animace nezamrzalo grafické uživatelské rozhraní.

Seznam použité literatury

[1] *Java algoritmy* [online]. [2004] [cit. 2009-04-24]. Dostupný z WWW: <http://javaalgoritmy.wz.cz/pfronta.htm>.

[2] WRóBLEWSKI, Piotr. *Algoritmy : Datové struktury a programovací techniky*. Překlad Marek Michalek, Bogdan Kiszka. 1. vyd. Brno : Computer Press, 2004. 351 s. ISBN 80-251-0343-9.

[3] ECKEL, Bruce. *Myslíme v jazyku Java : knihovna zkušeného programátora*. Přeložil Bogdan Kiszka. 1. vyd. Praha : Grada Publishing, 2001. 472 s. ISBN 80-247-0027-1.

[4] KEOGH, James. *Java bez předchozích znalostí : Průvodce pro samouky*. Přeložil Ivo Fořt. 1. vyd. Brno : CP Books, 2005. 275 s. ISBN 80-251-0839-2.

[5] *CZECH Info Center : Prvni Aplet* [online]. c1994-2006 [cit. 2009-05-12]. Dostupný z WWW: <http://www.czechinfocenter.com/e.mag/java/java2.html>.

[6] Sun Microsystem. *Graphics (Java 2 Platform SE v1.4.2)* [online]. c2003 [cit. 2009-04-21]. Dostupný z WWW: <http://java.sun.com/j2se/1.4.2/docs/api/java/awt/Graphics.html>.

[7] Sun Microsystems. *Graphics2D (Java 2 Platform SE v1.4.2)* [online]. c2003 [cit. 2009-04-21]. Dostupný z WWW: <http://java.sun.com/j2se/1.4.2/docs/api/java/awt/Graphics2D.html>.

[8] HEROUT, Pavel. *Java : grafické uživatelské prostředí a čeština*. 1. vyd. České Budějovice : KOPP, 2006. 316 s. ISBN 80-7232-237-0.

[9] Sun Microsystems. *ImageObserver (Java 2 Platform SE v1.4.2)* [online]. c2003 [cit. 2009-04-28]. Dostupný z WWW: <http://java.sun.com/j2se/1.4.2/docs/api/java/awt/image/ImageObserver.html>.

[10] BRACKEEN, David, BARKER, Bret, VANHELSUWÉ, Laurence. *Vývoj her v jazyku Java.* Voráček Karel. 1. vyd. Praha : Grada Publishing, 2004. 711 s. ISBN 80-247-0874-4.

[11] BERAN, Radek. *Třídící algoritmy* [online]. c2003-2005 [cit. 2009-05-04]. Dostupný z WWW: <http://www.beranr.webzdarma.cz/algoritmy/trideni.html#quicksort_rek>.

[12] Sun Microsystems. *Runnable (Java 2 Platform SE v1.4.2)* [online]. c2003 [cit. 2009-04-29]. Dostupný z WWW: <http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Runnable.html>.