Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Výukový program pro předmět „KIV/PT – Programovací techniky“

Plzeň 2013

Michal Dékány

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 10. května 2013

Michal Dékány

Abstrakt

Cílem práce je návrh a implementace programu, který bude sloužit k výuce algoritmů a datových struktur vyučovaných v předmětu KIV/PT – Programovací techniky. V první a druhé části jsou podrobně popsány vlastnosti vyučovaných datových struktur a algoritmů, které byly zahrnuty do programu. Třetí část se zabývá návrhem grafického uživatelského rozhraní pro tento program. V poslední části je popsána implementace navrženého grafického uživatelského rozhraní.

Abstract

The goal of this thesis is to design and implement the program, which will serve to teaching algorithms and data structures which are taught in the subject KIV/PT – Programming Techniques. In the first and second part are described data structures and algorithms which are included in the program. In the third part is described the design of the graphical user interface for this program and in the last section is described the implementation of designed graphical user interface.

Poděkování

Rád bych poděkoval vedoucímu bakalářské práce Ing. Pavlu Mautnerovi, Ph.D. a Ing. Petru Kratochvílovi za poskytnutí odborných rad, věcné připomínky a ochotu během zpracování této práce.

Obsah

1 Úvod **1**

2 Vyučované datové struktury **2**

2.1 Abstraktní datový typ 2
2.1.1 Implementace ADT 3
2.2 Zásobník 3
2.2.1 ADT Zásobník 4
2.2.2 Implementace zásobníku polem 5
2.2.3 Implementace zásobníku spojovým seznamem 6
2.2.4 Implementace zásobníku v Java Core API 7
2.3 Fronta 7
2.3.1 ADT Fronta 8
2.3.2 Implementace fronty polem 8
2.3.3 Implementace fronty spojovým seznamem 10
2.4 Obousměrná fronta 10
2.4.1 ADT Obousměrná fronta 11
2.4.2 Implementace obousměrné fronty 11
2.4.3 Implementace zásobníku a fronty 12
2.5 Spojový seznam 12
2.5.1 Jednosměrně zřetězený spojový seznam 13
2.5.2 Obousměrně zřetězený spojový seznam 15
2.6 Vektor 16
2.6.1 ADT Vektor 17
2.6.2 Implementace vektoru 17
2.7 Tabulka 18
2.7.1 ADT Tabulka 19
2.7.2 Tabulky s přímým přístupem 19
2.7.3 Tabulky se sekvenčním přístupem 20
2.7.4 Tabulky s rozptýlenými položkami 20
2.7.5 Tabulky s otevřeným rozptýlením 21
2.7.6 Tabulky s vnitřním zřetězením 22

2.7.7 Tabulky s vnějším zřetězením 23
2.7.8 Tabulky s lineárním zřetězením 24

3 Vyučované algoritmy 25
3.1 Vyhledávání řetězců 26
3.1.1 Základní terminologie 27
3.1.2 Algoritmus hrubé síly 27
3.1.3 Algoritmus Boyer-Moore 28
3.1.4 Algoritmus Knuth-Morris-Pratt 31
3.1.5 Algoritmus Rabin-Karp 33
3.2 Hledání podobnosti řetězců 35
3.2.1 Nalezení nejdelší společné podsekvence znaků 35
3.2.2 Nalezení nejkratšího společného „nadřetězce“ 37
3.3 Komprese dat 38
3.3.1 Run Length Encoding 38
3.3.2 Lempel-Ziv-Welch 39

4 Návrh GUI 42
4.1 Návrh menu 42
4.1.1 Lišta menu 43
4.1.2 Menu v samostatném okně 44
4.1.3 Podokno úloh 45
4.2 Návrh zobrazení práce algoritmu 45
4.2.1 Panel pro obrázek 46
4.3 Návrh zobrazení stavu datové struktury 47
4.3.1 Navržené komponenty pro zobrazení stavu 47
4.4 Návrh zobrazení zdrojového kódu 49
4.4.1 Umístění textového pole se zdrojovým kódem 49
4.4.2 Nástroje pro práci se zdrojovým kódem 50
4.4.3 Navržené okno pro zobrazení zdrojového kódu 51
4.5 Navržené hlavní okno GUI 52

5 Implementace GUI 53
5.1 Implementace modulů 53
5.1.1 Implementace algoritmů a datových struktur 53
5.2 Menu 54
5.2.1 Implementace menu 55
5.2.2 Implementace menu do hlavního okna 56
5.2.3 Úpravy menu 58
5.3 Zdrojový kód algoritmu či dat. struktury 59
5.3.1 Zvýraznění právě vykonávané řádky 60

5.3.2 První verze textového pole pro zdrojový kód 61
5.3.3 Druhá verze textového pole pro zdrojový kód 63
5.4 Akce na pozadí 63
5.5 Knihovna JEP 65

6 Závěr **66**

Literatura **69**

A Panel pro kompresi dat **70**

B Zvýrazňování vykonávané řádky **74**

C XSD šablona pro menu **76**

D XML soubor pro menu **78**

E Uživatelská příručka **81**

1 Úvod

Na Katedře informatiky a výpočetní techniky Fakulty aplikovaných věd Západočeské univerzity je vyučován předmět Programovací techniky. Cílem tohoto předmětu je poskytnout studentům informace o datových strukturách a algoritmech, které jsou v informatice považovány za základní.

Znalost těchto algoritmů a datových struktur je důležitá pro vypracování semestrální práce předmětu a pro úspěšné absolvování zkoušky. Tyto algoritmy a datové struktury jsou také velice důležité pro ostatní předměty studia, ve kterých se předpokládá jejich znalost za samozřejmost, a proto je nutné mít o nich dobrý přehled.

Bohužel pro studenty bývá pochopení některých algoritmů či datových struktur obtížné. Pro tyto studenty jsou k dispozici na webových stránkách předmětu Java applety, které by měly pomoci studentovi s jejich pochopením. Tyto Java applety ovšem slouží pouze jako ukázka stromových datových struktur. Studenti proto musí často vyhledávat informace o jednotlivých algoritmech a datových strukturách na internetu a doufat, že informace ve vyhledaných zdrojích jsou správné.

Proto vznikla tato práce, jejíž cílem je návrh a implementace grafického uživatelského rozhraní, které bude sloužit jako pomůcka při výuce algoritmů a datových struktur.

V práci jsou podrobně rozebrány vyučované algoritmy a datové struktury, které byly zahrnuty do programu. Dále práce obsahuje postup návrhu a popis implementace grafického uživatelského rozhraní.

2 Vyučované datové struktury

V předmětu Programovací techniky jsou vyučovány datové struktury:

- **Zásobník**
- **Fronta**
- **Obousměrná fronta**
- **Vektor**
- **Seznam**
- **Strom**
- **Tabulka**
- **Skip-List**

Do programu nebyla zahrnuta datová struktura strom, protože jsou pro ni na stránkách předmětu k dispozici odkazy na ukázkové Java applety. Datové struktury seznam a skip-list byly vynechány z důvodu vyšší náročnosti zahrnutí do programu. Zahrnutí vynechaných datových struktur je plánované v dalších verzích programu.

2.1 Abstraktní datový typ

Podle [6] je datový typ množinou hodnot a souborem operací s těmito hodnotami. Součástí Javy a ostatních programovacích jazyků jsou *primitivní datové typy* (viz příklad 2.1), se kterými bychom mohli vytvořit všechny naše programy, ale mnohem vhodnější je psát programy na vyšší úrovni abstrakce, tzn. využívat *abstraktní datové typy*.

Příklad 2.1: Datový typ `int` patří v Javě mezi primitivní datové typy. Hodnotami tohoto datového typu `int` jsou celá čísla mezi -2^{31} a $2^{31} - 1$. Mezi operace s tímto datovým typem patří $+$, $*$, $-$, $/$, $\%$, ...

Abstraktní datový typ (zkr. ADT) reprezentuje model složitějšího datového typu (často se jedná o množinu datových typů), jehož struktura je skryta před jeho uživatelem a jedinou možností, jak s hodnotami ADT pracovat, je prostřednictvím nabízených operací, které jsou přesně specifikovány nezávisle na konkrétní implementaci.

2.1.1 Implementace ADT

V [8] je uvedeno, že implementace ADT v Javě a většině dalších programovacích jazycích je rozdělena do dvou kroků.

Prvním krokem je definice *API*, nebo jednodušeji *rozhraní* (angl. interface), které popíše jména a způsob použití metod (operací) podporovaných v daném ADT. V rozhraní také můžeme určit, s jakými datovými typy bude ADT pracovat. V případě, že je určen datový typ `Object`, může ADT pracovat s datovými typy, které nemusí být homogenní (tj. stejné).

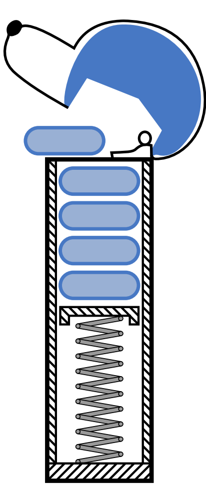
Abychom mohli začít ADT používat, musíme v druhém kroku vytvořit třídu, která bude implementovat všechny metody z rozhraní pro toto ADT. Díky tomu můžeme využívat ADT bez ohledu na způsobu implementace.

2.2 Zásobník

Zásobník (angl. stack) je v [5] definován jako datový typ, který se používá k přechodnému uchovávání prvků. Pro ukládání a vybírání prvku platí pravidlo, že se vždy vybírá prvek, který byl uložen jako poslední. Pro tuto uvedenou vlastnost je zásobník často označován jako *LIFO* (z angl. „last-in first-out“).

Podle [8] si princip zásobníku můžeme představit jako dávkovač bonbonů PEZ®, ve kterém jsou bonbony uloženy v boxu s pružinou (viz obrázek 2.1). Při odklopení víčka lze odebrat pouze horní bonbon (poslední vložený) a ostatní bonbony se díky pružině posunou nahoru.

Ve spojitosti s prvky zásobníku jsou spojeny pojmy *dno zásobníku* a *vrchol zásobníku* (angl. stack pointer). Dno zásobníku označuje první vložený prvek a zároveň poslední, který bude vybrán. Vrcholem zásobníku je poslední vložený prvek v zásobníku.



Obrázek 2.1: Nákres dávkovače bonbonů PEZ® – fyzická implementace zásobníku (PEZ® je registrovaná obchodní značka PEZ Candy, Inc.)

Dále [8] označuje zásobník jako základní datovou strukturu, která má využití v mnoha aplikacích včetně následujících příkladů.

Příklad 2.2: Internetový prohlížeč si uchovává adresy navštívených webových stránek do zásobníku. Pokaždé, když uživatel navštíví novou stránku, je její internetová adresa vložena do zásobníku adres. Prohlížeč umožňuje uživateli vrátit se na předchozí navštívenou stránku vyjmutím poslední vložené adresy ze zásobníku (tlačítko „zpět“).

Příklad 2.3: Textové editory obvykle umožňují zrušit poslední provedené změny v dokumentu. Tato operace může být zajištěna uchováváním změn textu dokumentu v zásobníku.

2.2.1 ADT Zásobník

Zásobník (označován S) je ADT, který poskytuje dvě základní metody:

- `push(o)` – vloží objekt `o` na vrchol zásobníku. Pokud je zásobník plný, dojde k chybě.

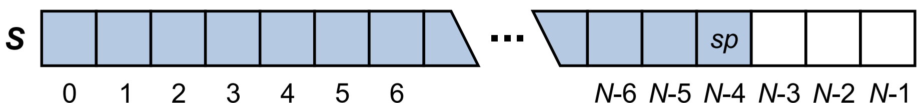
- `pop()` – vyjme objekt z vrcholu zásobníku. Pokud je zásobník prázdný, dojde k chybě.

Dále může obsahovat následující podpůrné metody:

- `size()` – vrací počet prvků v zásobníku.
- `isEmpty()` – vrací `true` pokud je zásobník prázdný, jinak `false`.
- `top()` – vrací objekt z vrcholu zásobníku bez jeho odstranění. Pokud je zásobník prázdný, dojde k chybě.

2.2.2 Implementace zásobníku polem

Podle [8] je nejjednodušší implementací zásobníku pole objektů (viz obrázek 2.2). Jelikož pole v Javě a většině dalších programovacích jazycích musí být vytvořeno s pevně danou délkou, je tedy tato délka (označení N) důležitou částí implementace, protože určuje maximální počet objektů v zásobníku (např. $N = 1000$ objektů). Tyto objekty budou uloženy v poli pro zásobník, které označíme S. Poslední důležitou proměnnou zásobníku je *ukazatel na vrchol zásobníku*, který označíme jako *sp*.



Obrázek 2.2: Implementace zásobníku S přímým polem

Způsob vkládání objektů:

- *přímý* – nové objekty se vkládají ve směru rostoucích adres (viz obrázek 2.2), tzn. že vrchol zásobníku roste a dno zásobníku je na indexu 0 (pole v Javě začínají indexem 0);
- *zpětný* – nové objekty se vkládají ve směru klesajících adres, tzn. že vrchol zásobníku klesá a dno zásobníku je na indexu $N - 1$.

Druhy ukazatelů na vrchol zásobníku:

1. ukazující na první volnou položku;
2. ukazující na poslední obsazenou položku.

Každá metoda zásobníku provádí konstantní počet aritmetických operací, porovnání a přiřazení, tzn. že má časovou složitost $O(1)$. Využití paměti implementace zásobníku polem je $O(N)$, kde N je velikost pole. Z toho vyplývá, že je nezávislá na počtu vložených objektů v zásobníku.

Podle [8] je implementace zásobníku polem velice jednoduchá a efektivní. Je široce využívána v různých počítačových aplikacích. Nicméně tato implementace má jednu nevýhodu, kterou je velikost pole N (kapacita zásobníku).

Budeme uvažovat implementaci zásobníku s výše zmiňovanou výchozí velikostí pole $N = 1000$ s možností zvolit vlastní velikost zásobníku. V případě, že aplikace využívající zásobník využívá méně místa, potom dochází k plýtvání pamětí. Na druhou stranu tato aplikace může potřebovat více místa a v tom případě by tato implementace zásobníku selhala při vložení $(N + 1)$ objektu.

Z těchto důvodů nemusí být tato implementace ideální. Naštěstí existují jiné druhy implementace, které jsou pouze omezeny pamětí přidělenou zvoleným programovacím jazykem a které využívají paměť úměrně s počtem uložených objektů v zásobníku (viz kapitola 2.2.3). Implementace zásobníku polem je nejvíce vhodná v případě, kdy máme dobrý odhad na počet položek, které budeme vkládat. Zásobníky hrají důležitou roli v řadě počítačových aplikací, takže je vhodné mít rychlou a paměť šetřící implementaci zásobníku.

2.2.3 Implementace zásobníku spojovým seznamem

Podle [8] lze zásobník také implementovat jednosměrně zřetězeným spojovým seznamem (viz kapitola 2.5.1). Je vhodné použít referenci na začátek spojového seznamu jako ukazatel na vrchol zásobníku, protože budeme vkládat a mazat objekty na začátku seznamu. Je také vhodné, abychom v zásobníku měli proměnnou, která bude sledovat aktuální počet vložených prvků.

Metody této implementace mají časové složitosti, které jsou stejné jako u implementace polem (viz kapitola 2.2.2). Paměťová náročnost této imple-

mentace vychází z vlastností spojového seznamu (viz kapitola 2.5) a je rovna $O(n)$, kde n je aktuální počet objektů v zásobníku. Maximální počet objektů v zásobníku je omezen pamětí počítače popř. pamětí přidělenou JVM, ale také lze tento počet ručně omezit. Pokud bychom přesáhli maximální možnou paměťovou kapacitu, potom je v Javě vyhozena výjimka `OutOfMemoryError`.

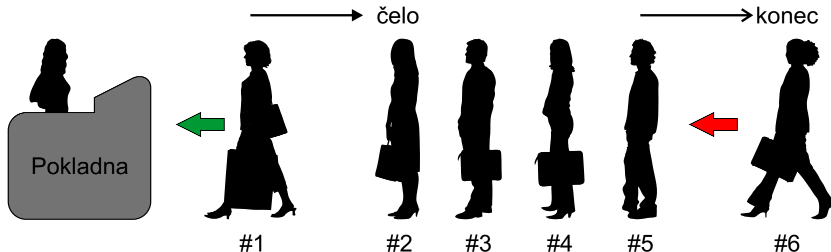
2.2.4 Implementace zásobníku v Java Core API

Z důvodu důležitosti zásobníku je Javě v balíčku `java.util` knihovní třída `Stack`. Instance této třídy umožňuje uchovávat libovolné objekty a poskytuje metody `push(o)`, `pop()`, `peek()` (ekvivalent k `top()`), `size()` a `empty()` (ekvivalent k `isEmpty()`). Pokud jsou metody `pop()` a `peek()` volány nad prázdným zásobníkem, vyhazují výjimku `StackEmptyException`.

2.3 Fronta

Fronta (angl. queue) je podle [5, 8] další základní datovou strukturou, která je příbuzná zásobníku (viz kapitola 2.2). Fronta je datový typ, který se používá k dočasnému uložení prvků, kde se prvek, který se ukládá dříve, také dříve vybere. Fronta je často označována jako *FIFO* (z angl. „first-in first-out“).

Princip fronty lze jednoduše vysvětlit na řadě lidí, kteří čekají u pokladny na odbavení (viz obrázek 2.3). Lidé vstupují do fronty na její konec a odbaveni mohou být lidé z čela fronty. Dalším možným příkladem je profesor na vysoké škole, kterému žáci odevzdávají semestrální práce. Tyto práce jsou profesorem opravovány v pořadí, ve kterém je obdržel.



Obrázek 2.3: Ukázka principu fronty na řadě čekajících lidí na obsloužení u pokladny – fyzická implementace fronty

Stejně jako zásobník i fronta má využití v mnoha aplikacích včetně následujícího příkladu.

Příklad 2.4: Součástí počítačové sítě často bývá sdílená tiskárna, kterou může využívat více uživatelů najednou. Když uživatel pošle požadavek k tisku (např. dokument), jeho tisková úloha je v tiskárně vložena do tiskové fronty. Když tato úloha dosáhne čela tiskové fronty, tiskárna ji provede. To zajišťuje, že pouze jeden uživatel může v daný moment tisknout. Tento princip také používají tiskárny, které využívá pouze jeden uživatel, v případě, že zašle více požadavků za sebou (tisk více dokumentů). Tyto požadavky jsou vyřizovány v pořadí, ve kterém je uživatel zaslal do tiskárny.

2.3.1 ADT Fronta

Fronta (označována Q) je ADT, který poskytuje dvě základní metody:

- `enqueue(o)` – vloží objekt `o` na konec fronty. Pokud je fronta plná, dojde k chybě.
- `dequeue()` – vyjme objekt z čela fronty. Pokud je fronta prázdná, dojde k chybě.

Dále může obsahovat stejně jako ADT zásobník (viz kapitola 2.2.1) následující podpůrné metody:

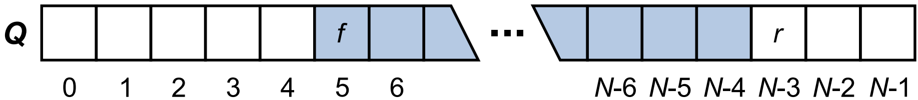
- `size()` – vrací počet prvků ve frontě.
- `isEmpty()` – vrací `true` pokud je fronta prázdná, jinak `false`.
- `front()` – vrací objekt z čela fronty bez jeho odstranění. Pokud je fronta prázdná, dojde k chybě.

2.3.2 Implementace fronty polem

Podle [8] je implementace fronty polem objektů je velice podobná jako u zásobníku (viz kapitola 2.2.2) s tím rozdílem, že vybíráme prvky ze začátku fronty a nové prvky vkládáme na konec. Pokud bychom zvolili $Q[0]$ jako čelo

fronty, potom by princip vkládání byl stejný jako u implementace zásobníku přímým polem, ale při vybírání prvku (metoda `dequeue`) bychom museli posunout všechny prvky ve frontě směrem k jejímu čelu. To znamená, že by výběr prvku měl časovou složitost $O(n)$.

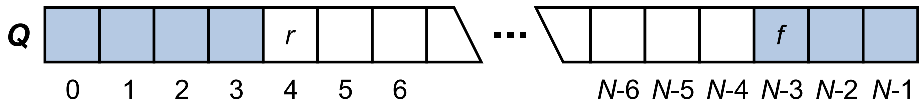
Abychom zabránili nutnosti posouvání prvků ve frontě, musíme definovat proměnnou f, která reprezentuje ukazatel na čelo fronty. Dále musíme definovat proměnnou r pro konec fronty, která bude obsahovat index, do kterého se bude vkládat nový prvek. Pokud se bude hodnota f rovnat hodnotě r, potom je fronta prázdná. Ukázku této implementace si lze prohlédnout na obrázku 2.4.



Obrázek 2.4: Implementace fronty Q polem

Při vytvoření fronty zvolíme $f = r = 0$, což znamená, že je fronta prázdná. V případě, že vložíme nový prvek do fronty, potom jednoduše navýšíme hodnotu r. V případě vyjmutí prvku z fronty navýšíme hodnotu f. Tento přístup přináší riziko přetečení indexu pole, kdy navýšením f nebo r získáme hodnotu větší nebo rovno N (velikost pole).

Abychom předešli přetečení indexů, musíme k poli fronty přistupovat jako ke *kruhovému poli* (viz obrázek 2.5), u kterého následuje po prvku $Q[N-1]$ prvek $Q[0]$. To je velice jednoduché, protože při každém navýšení hodnoty f nebo r vypočteme zbytek po celočíselném dělení velikostí pole (matematický operátor *modulo*), který vložíme do dané proměnné (např. $f = (f+1) \mod N$).



Obrázek 2.5: Ukázka řešení přetečení konce fronty r využitím kruhového pole

Nevýhodou použití kruhového pole je, že po vložení N prvků do fronty (fronta bude plná) se bude hodnota f rovnat hodnotě r, což nyní znamená, že

fronta je prázdná nebo plná. Naštěstí existuje několik způsobů řešení tohoto problému. Prvním z nich je zavedením nové proměnné, která bude sledovat aktuální počet vložených prvků. Dalším způsobem řešení je ponechání jedné volné buňky v poli, tzn. že maximální počet prvků fronty bude $N - 1$.

Každá metoda fronty provádí konstantní počet aritmetických operací, porovnání a přiřazení, tzn. že má časovou složitost $O(1)$. Využití paměti implementace fronty polem je $O(N)$, kde N je velikost pole. Z toho vyplývá, že je nezávislá na počtu vložených objektů ve frontě.

2.3.3 Implementace fronty spojovým seznamem

Podle [8] lze frontu také implementovat jednosměrně zřetězeným spojovým seznamem (viz kapitola 2.5.1). Z vlastností tohoto spojového seznamu vyplývá, že budeme vkládat nové prvky na konec seznamu a mazat prvky ze začátku. Proto musíme ve frontě uchovávat reference na začátek i konec spojového seznamu. Také je vhodné, abychom ve frontě měli proměnnou, která bude sledovat aktuální počet vložených prvků.

Metody této implementace mají časové složitosti, které jsou stejné jako u implementace polem (viz kapitola 2.3.2). Paměťová náročnost této implementace vychází z vlastností spojového seznamu (viz kapitola 2.5) a je rovna $O(n)$, kde n je aktuální počet objektů ve frontě. Maximální počet objektů ve frontě je omezen pamětí počítače popř. pamětí přidělenou JVM, ale také lze tento počet ručně omezit. Pokud bychom přesáhli maximální možnou paměťovou kapacitu, potom je v Javě vyhozena výjimka `OutOfMemoryError`.

2.4 Obousměrná fronta

V [3] je uvedeno, že *obousměrná fronta* (angl. *double-ended queue* nebo *deque*) je datová struktura podobná frontě (viz kapitola 2.3), ale umožňuje vkládání a výběr prvků na obou koncích. Této vlastnosti se využívá v aplikacích, ve kterých používáme zároveň zásobník (viz kapitola 2.2) i frontu. Obousměrnou frontou lze implementovat některé další datové struktury.

2.4.1 ADT Obousměrná fronta

Obousměrná fronta (označována D) je ADT, který spojuje vlastnosti ADT zásobníku (viz kapitola 2.2.1) a fronty (viz kapitola 2.3.1) a poskytuje následující základní metody:

- `insertFirst(o)` – vloží objekt `o` na začátek fronty.
- `insertLast(o)` – vloží objekt `o` na konec fronty.
- `removeFirst()` – vyjme objekt z čela fronty. Pokud je fronta prázdná, dojde k chybě.
- `removeLast()` – vyjme objekt z konce fronty. Pokud je fronta prázdná, dojde k chybě.

Dále může obsahovat následující podpůrné metody:

- `first()` – vrací objekt z čela fronty bez jeho odstranění. Pokud je fronta prázdná, dojde k chybě.
- `last()` – vrací objekt z konce fronty bez jeho odstranění. Pokud je fronta prázdná, dojde k chybě.
- `size()` – vrací počet prvků ve frontě.
- `isEmpty()` – vrací `true` pokud je fronta prázdná, jinak `false`.

2.4.2 Implementace obousměrné fronty

Obousměrnou frontu lze implementovat obousměrně zřetězeným spojovým seznamem (viz kapitola 2.5.2). Ve frontě musíme uchovávat odkazy na počáteční a koncový uzel spojového seznamu, abychom mohli vkládat a mazat prvky ze začátku i konce fronty. Také je vhodné, abychom ve frontě měli proměnnou, která bude sledovat aktuální počet vložených prvků.

Každá metoda obousměrné fronty provádí konstantní počet aritmetických operací, porovnání a přiřazení, tzn. že má časovou složitost $O(1)$. Paměťová náročnost této implementace vychází z vlastností obousměrně zřetězeného

spojového seznamu a je rovna $O(n+2)$, kde n je aktuální počet objektů ve frontě. Maximální počet objektů ve frontě je omezen pamětí počítače popř. pamětí přidělenou JVM, ale také lze tento počet ručně omezit. Pokud bychom přesáhli maximální možnou paměťovou kapacitu, potom je v Javě vyhozena výjimka `OutOfMemoryError`.

2.4.3 Implementace zásobníku a fronty

Jak již bylo uvedeno v definici obousměrné fronty, lze tuto frontu využít jako zásobník i frontu. Tabulky 2.1 a 2.2 obsahují metody obousměrné fronty korespondující s metodami zásobníku a fronty.

Metody zásobníku	**Metody obousměrné fronty**
size()	size()
isEmpty()	isEmpty()
top()	last()
push(o)	insertLast(o)
pop()	removeLast()

Tabulka 2.1: Metody zásobníku odpovídající metodám obousměrné fronty

Metody fronty	**Metody obousměrné fronty**
size()	size()
isEmpty()	isEmpty()
front()	first()
enqueue(o)	insertLast(o)
dequeue()	removeFirst()

Tabulka 2.2: Metody fronty odpovídající metodám obousměrné fronty

2.5 Spojový seznam

V [8] je uvedeno, že *spojový seznam* (angl. linked list) je dynamická datová struktura, která slouží k uchování předem neznámého počtu prvků. Základní stavební jednotkou spojového seznamu je *uzel* (angl. node), který vždy obsahuje objekt k uchování (angl. element) a ukazatel na další uzel v seznamu,

který určuje pozici daného uzlu. Některé implementace uzlů mohou obsahovat ukazatele na více uzlů v seznamu.

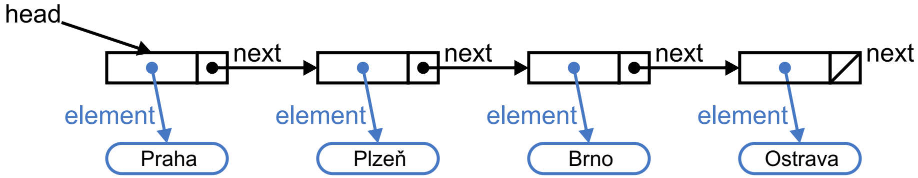
Stejně jako pole i spojový seznam udržuje prvky v určitém pořadí, které je určeno ukazateli na ostatní uzly v seznamu. Na rozdíl od pole, spojový seznam nemusí mít předem danou velikost a jeho velikost je určena počtem uzlů. Velikost spojového seznamu je pouze omezena pamětí počítače, popř. pamětí přidělenou JVM. Využití paměti spojovým seznamem je $O(n)$, kde n je počet uzlů v seznamu. Každý uzel využívá $O(1)$ paměti pro uložení odkazu na objekt a ukazatelů na další uzly v seznamu.

Spojový seznam může být implementován jako samostatné ADT, ale převážně se využívá k implementaci ostatních ADT (např. zásobník – viz 2.2, fronta – viz 2.3 atd.).

2.5.1 Jednosměrně zřetězený spojový seznam

Podle [8] je nejjednodušší forma spojového seznamu tvořena uzly, které tvoří lineární uspořádání. Každý uzel obsahuje objekt k uchování a odkaz na další uzel v řadě (angl. next) (viz obrázek 2.6).

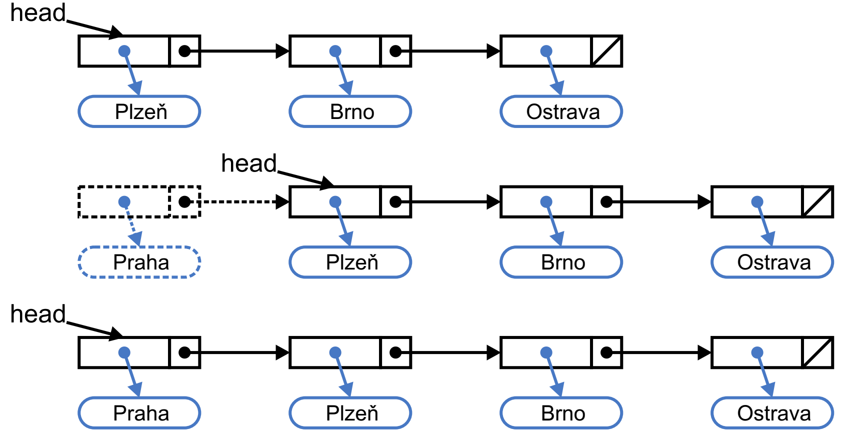
První uzel spojového seznamu se nazývá *hlava* či *začátek* (angl. head) a poslední uzel se nazývá *ocas* či *konec* (angl. tail). Konec spojového seznamu lze snadno poznat podle ukazatele na další uzel, který je `null`. Tento spojový seznam lze procházet pouze jedním směrem, a proto se často označuje jako *jednosměrně zřetězený seznam* (angl. singly linked list).



Obrázek 2.6: Ukázka jednosměrně zřetězeného spojového seznamu

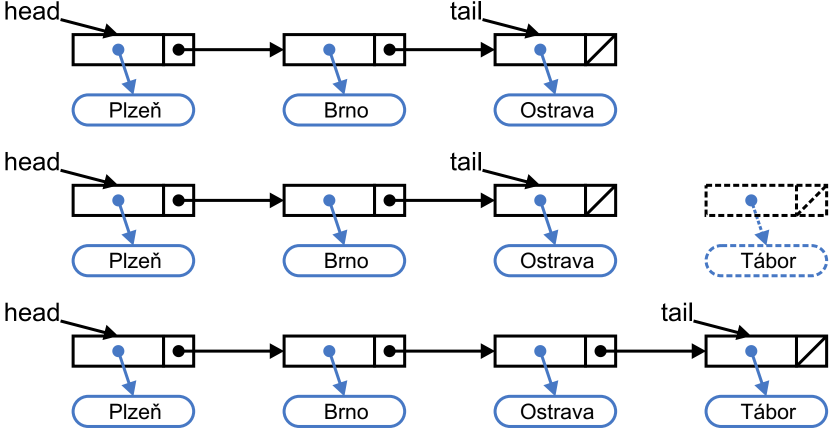
U jednosměrně zřetězeného spojového seznamu lze snadno vkládat či odstraňovat uzly ze začátku seznamu s časovou složitostí $O(1)$ (viz obrázek 2.7). Při vložení nového uzlu nastavíme ukazatel na začátek seznamu a pak nastavíme nový uzel jako referenci na začátek seznamu. Tento postup funguje i pro prázdný spojový seznam, protože reference na začátek seznamu ukazuje

na `null`. Pokud seznam není prázdný, potom můžeme odstranit uzel z jeho začátku. Při odstranění nastavíme ukazatel na další prvek jako referenci na začátek seznamu.



Obrázek 2.7: Vložení nového uzlu na začátek spojového seznamu

V případě, že máme referenci na konec seznamu, potom můžeme vkládat nové uzly na jeho konec s časovou složitostí $O(1)$ (viz obrázek 2.8). Novému uzlu nastavíme `null` jako ukazatel na další uzel. Uzlu na konci seznamu nastavíme ukazatel na nový uzel. Potom nastavíme referenci konce seznamu na tento nový uzel.



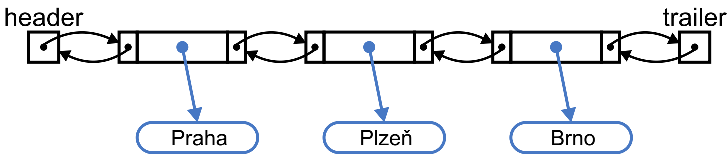
Obrázek 2.8: Vložení nového uzlu na konec spojového seznamu

Z jednosměrně zřetězeného seznamu nelze odstranit uzel z jeho konce s časovou složitostí $O(1)$. Při odstranění koncového uzlu musíme projít celý

spojový seznam a najít uzel před koncem seznamu, kterému nastavíme `null` jako ukazatel na další uzel a také nastavíme tento uzel jako referenci na konec seznamu. Z toho vyplývá, že odstranění posledního uzlu seznamu lze provést s časovou složitostí $O(n)$.

2.5.2 Obousměrně zřetězený spojový seznam

V [8] je uvedeno, že v některých případech vyžadujeme vkládání a mazání libovolných uzlů spojového seznamu, potom potřebujeme komplexnější strukturu než jednosměrně zřetězený spojový seznam (viz kapitola 2.5.1). Proto vznikl spojový seznam, který umožňuje velké množství operací – *obousměrně zřetězený spojový seznam* (angl. doubly linked list) (viz obrázek 2.9). Jak už vyplývá z názvu, tento seznam lze procházet oběma směry.



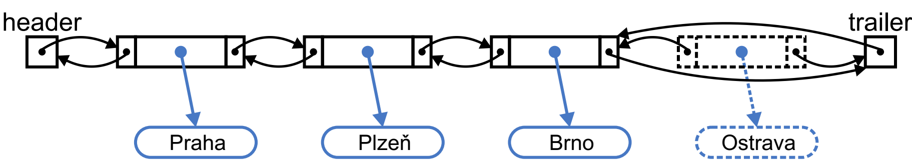
Obrázek 2.9: Ukázka obousměrně zřetězeného spojového seznamu

Každý uzel obousměrně zřetězeného seznamu obsahuje objekt k uchování (angl. element), odkaz na další uzel v seznamu (angl. next) a odkaz na předchozí uzel v seznamu (angl. prev).

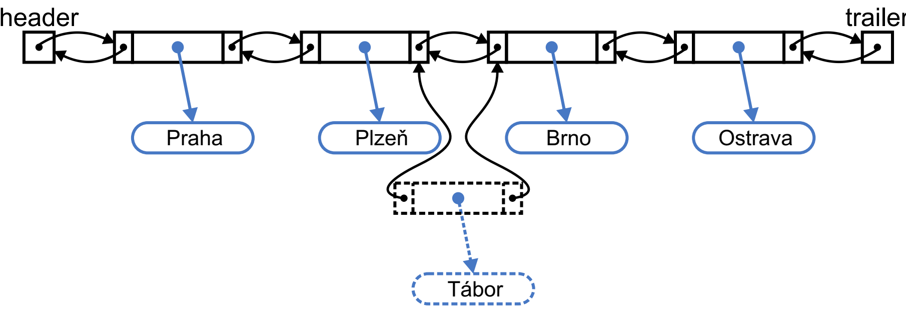
Pro jednoduchou implementaci tohoto spojového seznamu je vhodné přidat speciální uzly (nazývané *hlídky*) na oba konce seznamu. Prvním z nich je *hlavička* (angl. header), která je vložena před začátkem seznamu. Druhým speciálním uzlem je tzv. *koncový uzel* (angl. trailer), který je vložený za koncem seznamu. Tyto „fiktivní“ (angl. dummy) uzly neuchovávají žádné objekty a slouží pouze k ohraničení spojového seznamu. Hlavička má ukazatel na předchozí prvek ukazující na `null` a ukazatel na další prvek ukazuje na začátek seznamu. Obdobně koncový uzel má ukazatel na další prvek ukazující na `null` a ukazatel na předchozí prvek ukazuje na konec seznamu. V prázdném seznamu hlavička ukazuje na koncový uzel a naopak.

Vložení nebo odstranění uzlů na obou koncích obousměrně zřetězeného spojového seznamu má díky odkazům na předchozí uzly časovou složitost

$O(1)$, protože pro odstranění posledního uzlu nemusíme procházet celým seznamem k nalezení předposledního uzlu. Vložení nebo odstranění nekoncových uzlů vyžaduje procházení seznamu, a proto má časovou složitost $O(n)$. Ukázky vložení a odstranění uzlů z obousměrně zřetězeného spojového seznamu si lze prohlédnout na obrázcích 2.10 a 2.11.



Obrázek 2.10: Ukázka smazání posledního uzlu z obousměrně zřetězeného spojového seznamu



Obrázek 2.11: Ukázka vložení nového uzlu do obousměrně zřetězeného spojového seznamu

2.6 Vektor

V [8] je *vektor* definován jako lineární posloupnost, ve které se můžeme odkazovat na prvky s libovolnou pozicí r (z angl. rank) z intervalu $\langle 0, n-1 \rangle$, kde n je počet vložených prvků. První prvek vektoru má index 0 a poslední prvek má index $n-1$, což je stejný způsob indexace jako u polí v Javě nebo v jiném programovacím jazyce. Pozice prvku se může během práce s vektorem průběžně měnit. Například pokud vložíme nový prvek na začátek vektoru, pozice ostatních prvků se zvýší o jednu.

2.6.1 ADT Vektor

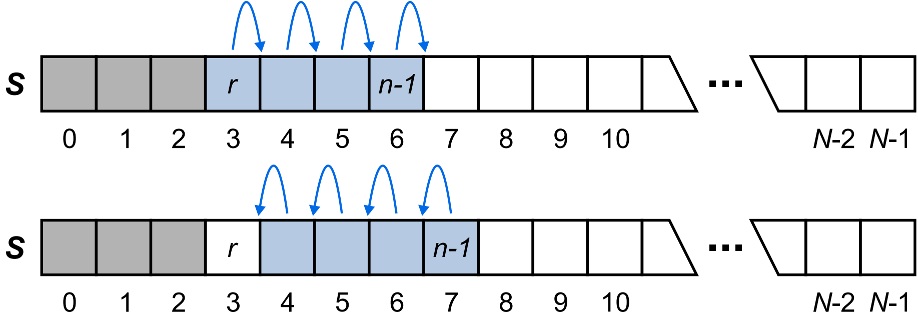
Vektor (označován S) je ADT, který poskytuje mimo obvyklé metody `size()` a `isEmpty()` následující metody:

- `elemAtRank(r)` – vrací prvek na pozici `r`.
- `replaceAtRank(r, o)` – zamění prvek na pozici `r` prvkem `o` a vrátí původní prvek.
- `insertAtRank(r, e)` – vloží nový prvek `o` na pozici `r`.
- `removeAtRank(r)` – odstraní a vrátí prvek na pozici `r`.

V těchto metodách dojde k chybě v případě, že $r \notin \langle 0, n-1 \rangle$, kde n je počet prvků ve vektoru.

2.6.2 Implementace vektoru

Podle [8] je zřejmou volbou pro implementaci vektoru pole (označíme A), ve kterém bude každá buňka $A[r]$ uchovávat prvek s pozicí r. Díky tomu můžeme snadno implementovat metody `elemenAtRank` a `replaceAtRank`, ve kterých se bude pracovat pouze s buňkou $A[r]$. Implementace metod `insertAtRank` a `removeAtRank` je složitější, protože musíme přesouvat prvky ve vektoru nahoru nebo dolů k zachování souvislosti obsazených buněk (viz obrázek 2.12).



Obrázek 2.12: Ukázka posunů nahoru při vložení nového prvku a posunů dolů při smazání prvku na pozici r

Hlavní slabinou této implementace je pevná velikost pole N, protože může dojít k plýtvání pamětí v případě malého počtu prvků n. Dále může dojít k přetečení vektoru v případě, že se snažíme vložit větší počet prvků, než je velikost pole. Řešením této slabiny je využití *pole proměnné délky* (angl. extendable array), u kterého v případě zaplnění zvětšíme jeho velikost.

Jelikož v Javě a většině dalších programovacích jazycích nemůžeme zvětšit velikost pole, protože je pevně dána, musíme v případě zaplnění pole (využití v metodě `insertAtRank`) provést následující kroky:

1. Vytvoření nového pole B o velikosti $2N$.
2. Zkopírování všech prvků z pole A do pole B.
3. Nahrazení reference na pole A referencí na pole B ($A = B$).

Časová složitost metod `insertAtRank` a `removeAtRank` je kvůli potřebným posunům a případnému růstu pole $O(n)$. Ostatní metody vektoru provádí konstantní počet aritmetických operací, porovnání a přiřazení, tzn. že mají časovou složitost $O(1)$.

2.7 Tabulka

Datový typ *tabulka* je v [3] definován jako datová struktura, která umožňuje vkládat a vybírat prvky podle identifikačního klíče k. Každá položka v tabulce má mimo identifikačního klíče také adresní klíč A_k tj. „adresa položky“, který je nejčastěji odvozen z klíče k zobrazením $h : k \rightarrow A_k$.

Zobrazení $h(k) \rightarrow A_k$ nemusí být prosté, tzn. že může platit $h(k_1) = h(k_2)$ pro položky s různými klíči k_1 a k_2. Tyto položky jsou nazývány *synonyma*.

U tabulek dále definujeme pojem *plnění tabulky*, které určuje poměr mezi počtem položek v tabulce a předpokládaného počtu položek. Pro výpočet plnění tabulky slouží vzorec:

$$P = \frac{n_s}{n_f} \tag{2.1}$$

kde:

- n_s je skutečný počet položek v tabulce;
- n_f je předpokládaný počet položek v tabulce (odpovídá velikosti tabulky).

2.7.1 ADT Tabulka

Tabulka je ADT, který poskytuje následující metody:

- `search(k)` – vrací prvek s klíčem `k`.
- `insert(k, o)` – vloží prvek `o` s klíčem `k` do tabulky.
- `delete(k)` – odstraní a vrátí prvek s klíčem `k`.
- `size()` – vrací velikost tabulky.

V případě, že metoda `search` nebo `delete` nenaleznou prvek s klíčem `k`, potom zpravidla vrací `null`.

2.7.2 Tabulky s přímým přístupem

V [3] je uvedeno, že u *tabulky s přímým přístupem* je $h(k) \rightarrow A_k$ prostým zobrazením, tzn. že každá položka tabulky má své místo jednoznačně určené hodnotou A_k přímo odvozenou z k. V [2] je uvedeno, že to znamená, že v tabulce se nevyskytují synonyma. Když známe rozsah identifikačního klíče, známe počet položek v tabulce. Potom můžeme implementovat tabulku polem, protože indexy jsou přímo klíče v tabulce.

Výhodou této tabulky je jednoduchá implementace a díky přímému přístupu k položkám mají metody časovou složitost $O(1)$.

Jelikož je velikost tabulky daná rozsahem klíče, pro praktické účely bývá většinou neúnosná. Navíc vzhledem k rozsahu tabulky může vznikat *řídké pole*, ve kterém je kvůli nerovnoměrnému počtu klíčů mnoho prázdných buněk.

2.7.3 Tabulky se sekvenčním přístupem

Podle [3] se v *tabulkách se sekvenčním přístupem* využívá sekvenční vyhledávání, které je nejjednodušším způsobem vyhledávání položek v tabulce, protože nepředpokládá žádnou vnitřní organizaci tabulky. Toto vyhledávání prochází tabulkou po jednotlivých položkách, u kterých je porovnáván jejich klíč s hledaným klíčem. Vyhledávání je ukončeno nalezením hledané položky nebo nalezením konce tabulky.

Do tabulky jsou nové položky vkládány na konec tabulky. Při výběru položky z tabulky je přesunuta poslední položka na uvolněné místo.

Další možností je, že při výběru položky mohou v tabulce vznikat mezery, které jsou označeny jako prázdné (neplatné) položky speciálním klíčem, jehož hodnoty klíče nikdy nemůžou nabýt (např. -1). Potom při vkládání nových položek je vyhledáno první volné místo v tabulce, do kterého je položka umístěna.

Tabulky se sekvenčním přístupem jsou obvykle implementovány polem nebo spojovým seznamem (viz kapitola 2.5).

Podle [5] se sekvenční přístup se používá především tam, kde požadujeme zpracování všech položek v tabulce.

2.7.4 Tabulky s rozptýlenými položkami

V případě velikého rozsahu klíče k se tabulka s přímým přístupem (viz kapitola 2.7.2) stává nepoužitelná, protože její plnění je nepřijatelně nízké a rozsah adresního klíče A_k může být i mimo rozsah implementujícího pole (viz příklad 2.5).

Uvedené nevýhody odpadají, jestliže upustíme od požadavku prostého zobrazení a připustíme výskyt synonym v tabulce. Pro získání adresního klíče A_k k určení pozice v *tabulce s rozptýlenými položkami*, na kterou máme uložit položku s klíčem k, budeme využívat *rozptylovací funkci* (označení h), která je často nazývaná jako *hašovací funkce*. Platí tedy, že $A_k = h(k)$, neboli adresní klíč je roven hodnotě rozptylovací funkce pro identifikační klíč.

Rozptylovací funkce transformuje veliký rozsah identifikačních klíčů na nepoměrně malý rozsah adresních klíčů. Proto musí být pro každé k jedno-

značně definována a měla by vytvářet co nejmenší počet kolizí. Tato funkce obvykle bývá realizována jako výpočet zbytku po celočíselném dělení klíče k velikostí tabulky.

Jak už bylo zmíněno, do tabulky mohou být vkládány synonyma (často nazývána přeplnění), pro která platí $k_1 \neq k_2$ a $h(k_1) = h(k_2)$, u kterých dochází při vložení do tabulky ke kolizi. Řešení ukládání synonymických položek je důležitou částí implementace tabulky s rozptýlenými položkami.

Příklad 2.5: V některých organizacích může být evidence pracovníků tvořena tabulkou a klíčem každého pracovníka může být jeho rodné číslo. Pro ukládání číselných hodnot rodných čísel musí být v Javě použit datový typ `long`, ale maximální velikost pole je omezena maximální velikostí datového typu `int`. Proto musíme použít tabulku s rozptýlenými položkami, do které budeme pracovníky ukládat podle hash hodnoty rodného čísla.

2.7.5 Tabulky s otevřeným rozptýlením

Tabulka s otevřeným rozptýlením je jednoduchý typ tabulky s rozptýlenými položkami (viz kapitola 2.7.4), ve které se synonyma ukládají na volné místo, jehož adresa se vypočte z hodnoty hašovací funkce. Při pokusu uložení synonyma může dojít k vícenásobnému přepočtu adresního klíče položky, proto si označíme funkci pro výpočet nového adresního klíče jako h_i, kde i je pořadové číslo výpočtu.

Jelikož častým způsobem implementace tabulky bývá pole, musíme si při výpočtech adresního klíče dávat pozor na přetečení indexu pole. Proto musíme k poli tabulky přistupovat jako ke kruhovému poli, které bylo využito k implementaci fronty polem (viz kapitola 2.3.2).

Pro výpočet nového adresního klíče se nejčastěji používají tyto metody:

- *lineární zkoušení* – k adresnímu klíči vypočtenému hašovací funkcí se přičítá konstantní číslo s (obvykle 1).

$$h_i(k) = \Big(h(k) + i \cdot s\Big) \pmod{N} \tag{2.2}$$

- *kvadratické zkoušení* – k adresnímu klíči vypočtenému hašovací funkcí se přičítá hodnota pevně daného kvadratického polynomu.

$$h_i(k) = \Big(h(k) + c_1 \cdot i + c_2 \cdot i^2\Big) \pmod N \tag{2.3}$$

- *dvojité hašování* – k adresnímu klíči vypočtenému hašovací funkcí se přičítá hodnota jiné hašovací funkce H.

$$h_i(k) = \Big(h(k) + i \cdot H(k)\Big) \pmod N \tag{2.4}$$

Během vkládání prvků do tabulky je nutné kontrolovat, zda není tabulka plná. V [5] je uvedeno, že existují v zásadě dva způsoby testu plné tabulky. Buď evidujeme aktuální počet obsazených prvků v tabulce, nebo testujeme, zda jsme při hledání nejbližší následující volné pozice opět nenarazili na původní adresní klíč vypočítaný hašovací funkcí.

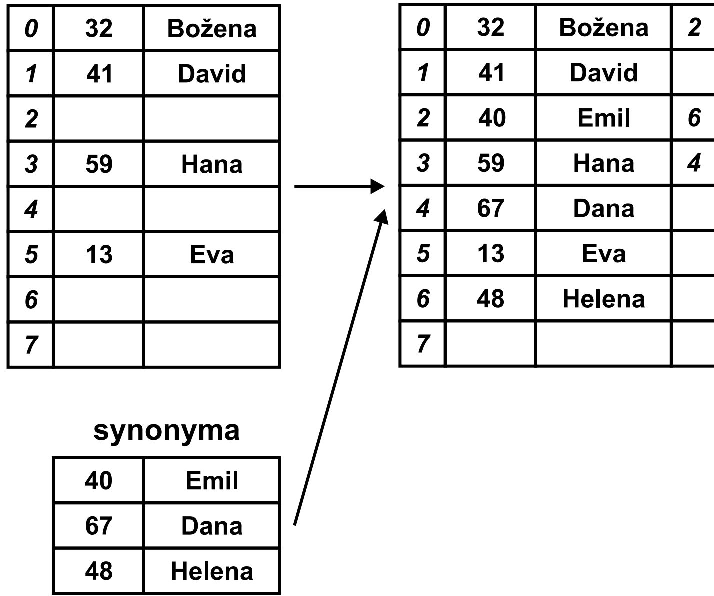
2.7.6 Tabulky s vnitřním zřetězením

Tabulka s vnitřním zřetězením je podle [2] podobná tabulce s otevřeným rozptýlením (viz kapitola 2.7.6). Rozdíl je v tom, že všechna synonyma se stejným adresním klíčem tvoří řetězec, takže při výběru položky není nutné postupovat od adresy A_k sekvenčním způsobem, nýbrž po nalezení řetězu synonym procházíme po tomto řetězu, čímž se zkracuje délka hledání oproti tabulce s otevřeným rozptýlením položek.

Tabulka s vnitřním zřetězením se vytváří ve dvou etapách (viz obrázek 2.13):

1. Zařazují se pouze základní položky a synonyma se zatím nezařazují do tabulky a ukládají se do zóny přeplnění.
2. Do volných míst tabulky se od každé základní položky umístí její synonyma ze zóny přeplnění tak, že tvoří řetěz.

V [5] je uvedeno, že v případě vložení nové položky do naplněné tabulky může nastat situace, kdy vkládaná položka není synonymum a její místo je obsazeno. V tomto případě je nutné pro novou položku místo uvolnit. Vyřazená položka je však nutně součástí některého řetězu synonym a ten se nesmí porušit. Proto je nutné seznam zřetězených synonym realizovat jako *obousměrný*.



Obrázek 2.13: Ukázka vytvoření tabulky s vnitřním zřetězením

2.7.7 Tabulky s vnějším zřetězením

V [3] je uvedeno, že *tabulka s vnějším zřetězením* je rozdělena na dvě části (viz obrázek 2.14) přinášející vylepšení oproti tabulce s vnitřním zřetězením (viz kapitola 2.7.6). První částí je *primární část* (základní tabulka), která obsahuje pouze položky, které nejsou synonyma. Druhou částí je *sekundární část* (zóna zřetězení, přeplnění), ve které jsou uloženy synonyma k položkám v tabulce.

K implementaci zóny zřetězení (popř. celé tabulky) se často využívá vektor (viz kapitola 2.6), protože využívá pole proměnné délky, které se při zaplnění zvětšuje.

Podle [2] je nevýhodou této tabulky nižší plnění, neboť na rozdíl od vnitřního zřetězení zůstanou volná místa v základní tabulce nevyužita. I přes tuto nevýhodu je tabulka s vnějším zřetězením velmi výhodná a používá se často pro implementaci tabulek identifikátorů v překladačích programových jazyků.

primární část

0	75	Božena	5
1	31	David	
2			
3	73	Hana	6
4			

sekundární část

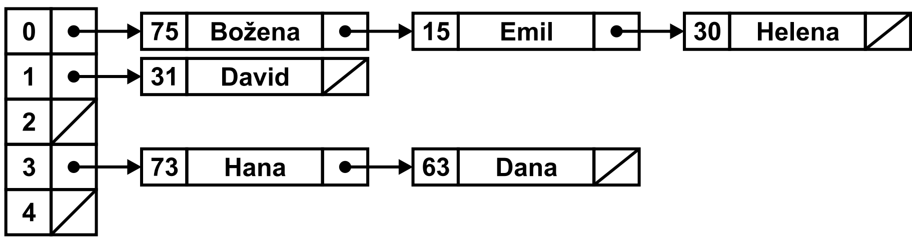
5	15	Emil	7
6	63	Dana	
7	30	Helena	

Obrázek 2.14: Ukázka tabulky s vnějším zřetězením

2.7.8 Tabulky s lineárním zřetězením

Tabulka s lineárním zřetězením je nejjednodušší formou tabulky s rozptýlenými položkami (viz kapitola 2.7.4), protože využívá jednoduchý způsob řešení kolizí (synonymických položek). Tabulka je tvořena polem spojových seznamů (viz kapitola 2.5) a v případě kolize se prvek přidá na začátek (popř. konec) adresovaného spojového seznamu (viz obrázek 2.15).

Tabulky s lineárním zřetězením jsou často používány díky snadné implementaci. Jejich hlavní nevýhodou je náročné sekvenční prohledávání příslušného seznamu při velkém zaplnění tabulky.



Obrázek 2.15: Ukázka tabulky s lineárním zřetězením

3 Vyučované algoritmy

Vyučované algoritmy v předmětu Programovací techniky jsou rozděleny do následujících skupin:

- **Grafové algoritmy**
  - **Hledání nejkratší cesty**
    - Hledání nekratší cesty v acyklickém orientovaném grafu
    - Dijkstrův algoritmus
    - Floyd-Warshallův algoritmus
  - **Hledání minimální kostry grafu**
    - Kruskalův algoritmus
    - Borůvkův algoritmus
    - Primův-Jarníkův algoritmus
- **Algoritmy zpracování textů**
  - **Vyhledávání řetězců**
    - Algoritmus hrubé síly
    - Algoritmus Boyer-Moore
    - Algoritmus Knuth-Moris-Pratt
    - Algoritmus Rabin-Karp
    - Vyhledávání řetězců pomocí Trie
  - **Hledání podobnosti řetězců**
    - Nalezení nejdelší společné podsekvence znaků
    - Nalezení nejkratšího společného „nadřetězce"
    - Porovnávání řetězců (edit-distance)
- **Kompresní algoritmy**
  - Run Length Encoding
  - Lempel-Ziv-Welch
  - Huffmanovo kódování
  - Aritmetické kódování

Do programu nebyly zahrnuty grafové algoritmy, protože jejich implementace vyžaduje komplexnější přístup k vytváření grafů pro tyto algoritmy.

Z algoritmů pro zpracování textů byl vynechán algoritmus pro vyhledávání řetězců pomocí Trie, protože datová struktura Trie patří mezi stromové datové struktury, které budou zahrnuty v další verzi programu. Dále byl z časových důvodů vynechán algoritmus porovnání řetězců (edit-distance).

Z kompresních algoritmů bylo vynecháno Huffmanovo kódování, protože stejně jako datová struktura Trie, i Huffmanovo kódování využívá stromové struktury. Dále bylo z časových důvodů vynecháno Aritmetické kódování.

Všechny vynechané algoritmy budou zahrnuty v dalších verzích programu.

3.1 Vyhledávání řetězců

V [8] je uvedeno, že algoritmus *vyhledávání řetězců* (angl. pattern matching) zjišťuje, zda hledaný řetězec P o délce m se nachází v textu T o délce n. Jak vyplývá z názvu, algoritmus se snaží najít podřetězec (viz kapitola 3.1.1) textu T, který začíná na pozici (indexu) i a který odpovídá P. Pro nalezený index i musí tedy platit, že $T[i] = P[0]$, $T[i+1] = P[1]$, ..., $T[i+m-1] = P[m-1]$; neboli $P = T[i..i+m-1]$. Výsledkem algoritmu může být informace, zda se řetězec P nachází v T, nebo index i (popř. -1 pokud nebyl nalezen). Další možností jsou všechny nalezené indexy i.

Pro zajištění obecnosti algoritmu není vhodné omezit znaky v T a P na určitou známou znakovou sadu (např. ASCII, Unicode), ale na obecnější znakovou sadu (abecedu) označenou Σ. Tato abeceda Σ může být podmnožinou znakové sady ASCII, Unicode aj., ale také se může jednat o libovolnou množinu znaků, která nemusí být konečná. Příkladem může být abeceda $\Sigma = \{0, 1\}$ pro binární čísla. Jelikož praktické využití vyhledávání řetězců pracuje s konečnými abecedami, budeme tedy přepokládat, že abeceda Σ je konečná, tzn. že $|\Sigma| = konst$.

Příklad 3.1: V [1] je uvedeno, že problém nalezení všech výskytů podřetězce v textu se často využívá v textových editorech (funkce hledat), kde T je text dokumentu a P je vyhledávaný řetězec, který byl zadán uživatelem. Text dokumentu může být obsáhlý, a proto je vhodné použít efektivní

algoritmy, protože výrazně snižují dobu odezvy. Vyhledávání řetězců se také používá při práci s konkrétními vzory v sekvencích DNA.

3.1.1 Základní terminologie

Budeme předpokládat, že máme řetězec S o délce m, potom lze podle [3] definovat pojmy:

- *podřetězec* (angl. substring) $S[i..j]$ – část řetězce S mezi indexy i a j, kde $0 <= i <= j < m$;
- *předpona* (angl. prefix) řetězce S – podřetězec $S[0..i]$, kde i je libovolný index mezi 0 a $m-1$;
- *přípona* (angl. suffix) řetězce S – podřetězec $S[i..m-1]$, kde i je libovolný index mezi 0 a $m-1$.

Příklad 3.2: Nechť máme řetězec `S = "andrew"`, potom:

- Podřetězec $S[1..3]$ = `"ndr"`;
- Prefixy S jsou: `"andrew"`, `"andre"`, `"andr"`, `"and"`, `"an"`, `"a"`;
- Suffixy S jsou: `"andrew"`, `"ndrew"`, `"drew"`, `"rew"`, `"ew"`, `"w"`.

3.1.2 Algoritmus hrubé síly

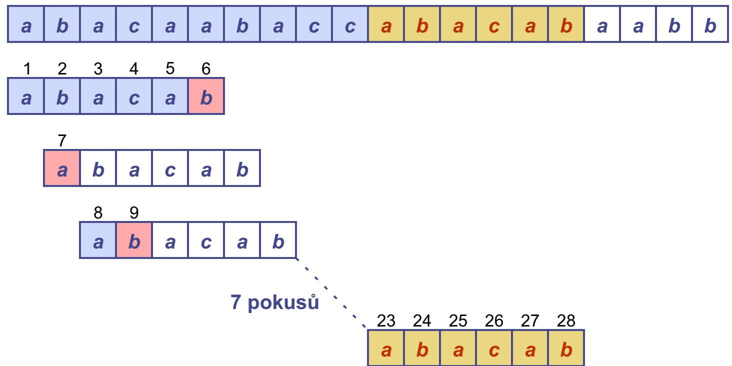
Podle [8] je algoritmus *hrubé síly* (angl. brute force) silná algoritmická technika, která se využívá v případech, kdy chceme něco hledat nebo když chceme optimalizovat některé funkce. Při použití této techniky obvykle vytvoříme výčet všech možných řešení a vybereme z nich to nejlepší. Nevýhodou tohoto algoritmu často bývá velká časová popř. paměťová složitost hledání.

Použití této techniky při vyhledávání řetězců odpovídá algoritmu, který nás pravděpodobně napadne jako první, tj. vyzkoušení všech možných umístění řetězce P v řetězci T.

Časová složitost tohoto algoritmu je $O((n-m+1)m)$, nebo zjednodušeně $O(nm)$.

Příklad 3.3: U algoritmu hrubé síly je nejhorším případem např. vyhledávání řetězce P `= "aaah"` v textu T `= "aaaaaaaaaaaaaaaaaaaaaaaaah"`. Při vyhledávání v T dojde v každé testované pozici k m porovnání znaků, aby algoritmus zjistil, že se na této pozici nenachází hledaný řetězec P.

Příklad 3.4: Na obrázku 3.1 můžeme vidět ukázku vyhledávání řetězce P `= "abacab"` v textu T `= "abacaabaccabacabaabb"`. Z obrázku je vidět, že bude provedeno celkem 11 pokusů, při kterých dojde k 28 porovnání znaků. Modře označené znaky jsou shodné, oproti tomu červeně označené znaky shodné nejsou. Zlatě je označen nalezený řetězec.



Obrázek 3.1: Ukázka vyhledávání řetězce algoritmem hrubé síly

3.1.3 Algoritmus Boyer-Moore

V [8] je uvedeno, že se nám možná může zdát, že je vždy nutné ověřovat každý znak v T k nalezení řetězce P. To však vždy nemusí platit pro tento algoritmus, který se někdy může vyhnout zbytečným porovnáním znaků P se značnou částí znaků z T. Algoritmus *BM* (Boyer-Moore) pracuje s konečnou abecedou Σ, což může být nevýhodou oproti algoritmu hrubé síly (viz kapitola 3.1.2), který nevyžaduje konečnou abecedu. Tento algoritmus nejlépe pracuje s přiměřeně velkou abecedou a s relativně dlouhým P, proto je vhodný pro vyhledávání slov v dokumentech.

Hlavní myšlenkou algoritmu je vylepšit dobu běhu algoritmu hrubé síly přidáním dvou způsobů vyhledávání, které výrazně šetří čas. Během testování umístění řetězce P v textu T využíváme tyto způsoby (metody) vyhledávání:

1. *Zrcadlový přístup k vyhledávání* – porovnávání provádí od konce P směrem na začátek;
2. *Přeskočení skupiny znaků* – v případě, že znak $c = T[i]$ není roven $P[j]$, potom se ověřuje, zda P obsahuje znak c. Pokud neobsahuje, potom se vyhledávání posune o m tak, aby bylo $P[0]$ zarovnáno s $T[i + 1]$, protože se na tomto místě řetězec P nemůže vyskytovat. V opačném případě se v P nalezne poslední výskyt tohoto znaku a vyhledávání se posune doprava tak, aby platilo $T[i] = P[j]$. Pokud posun doprava není možný, potom se posune P o jeden znak k $T[i + 1]$.

Zrcadlový přístup k vyhledávání umožňuje vynechání velké skupiny porovnávání znaků P a T, protože porovnáváním odzadu nalezne nerovnající se znak rychleji. Pokud narazí na nerovnost, potom se využije metoda přeskočení skupiny znaků, která vynechá mnoho zbytečných porovnání posunutím P na správnou pozici vůči T. Čím dříve je využita metoda přeskočení skupiny znaků, tím rychleji dochází k posunutí P na místo, na kterém se nachází podřetězec S, pro který platí $P = S$.

Při implementaci metody přeskočení skupiny znaků musíme definovat funkci `last(`c`)` (viz obrázek 3.2), která vezme znak c z abecedy Σ a definuje, jak moc se může posunout řetězec P v případě, že se znak c nachází v textu na pozici $T[i]$ a zároveň neodpovídá znaku $P[j]$. Tato funkce se počítá pro každý řetězec P před začátkem vyhledávání. Funkce `last(`c`)` vrací pro všechna c, která se nachází v P, pozici (index) posledního výskytu v řetězci P. V opačném případě vrací -1.

Jelikož lze v Javě použít znaky jako indexy pole, potom lze funkci `last(`c`)` implementovat jako pole o délce Σ. Metoda, která vytvoří pole pro tuto funkci, bude mít časovou složitost $O(m+|\Sigma|)$. Takto implementovaná funkce `last(`c`)` obsahuje všechny informace, které jsou potřeba pro správnou funkčnost metody pro přeskočení skupiny znaků.

Časová složitost algoritmu BM pro nejhorší případ je $O(mn + |\Sigma|)$. Konkrétně časová složitost pro výpočet funkce `last(`c`)` je $O(m + |\Sigma|)$ a nejhorší případ vyhledávání řetězce má časovou složitost $O(nm)$, která je stejná jako

u vyhledávání hrubou silou. Příklad nejhoršího případu vyhledávání je znázorněn na obrázku 3.4.

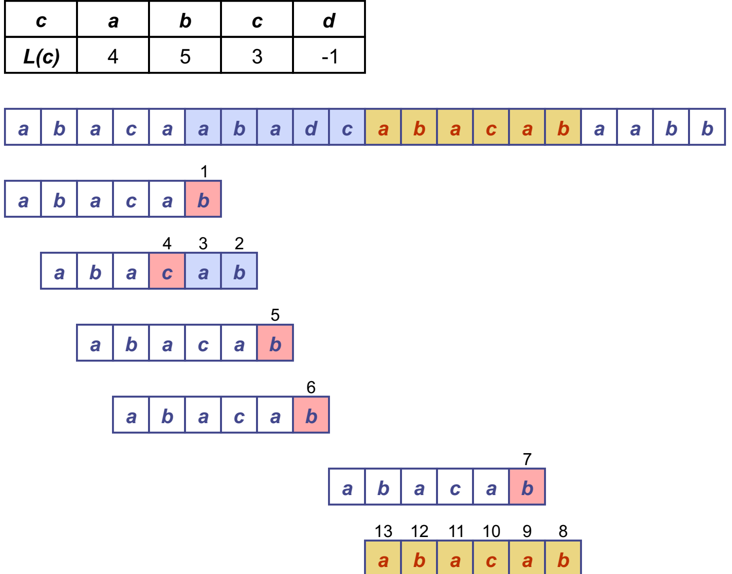
	0	1	2	3	4	5
P	a	b	a	c	a	b

Σ = {a, b, c, d}

c	a	b	c	d
Last(c)	4	5	3	-1

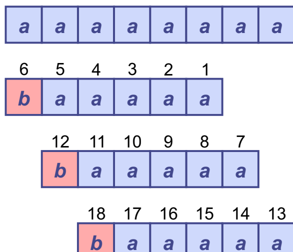
Obrázek 3.2: Ukázka funkce `last(`c`)` pro řetězec S `= "abacab"`.

Příklad 3.5: Na obrázku 3.3 můžeme vidět ukázku vyhledávání řetězce P `= "abacab"` v textu T `= "abacaabadcabacabaabb"`. Z obrázku je vidět, že bude provedeno celkem 6 pokusů, při kterých dojde k 13 porovnání znaků. Modře označené znaky jsou shodné, oproti tomu červeně označené znaky shodné nejsou. Zlatě je označen nalezený řetězec.



Obrázek 3.3: Ukázka vyhledávání řetězce algoritmem Boyer-More

Příklad 3.6: Na obrázku 3.4 je ukázka nejhoršího případu vyhledávání. Vyhledává se řetězec P `= "baaaaa"` v textu T `= "aaaaaaaa"`. Modře označené znaky jsou shodné, oproti tomu červeně označené znaky shodné nejsou. Zlatě je označen nalezený řetězec.



Obrázek 3.4: Ukázka nejhoršího případu vyhledávání algoritmem Boyer-Moore

3.1.4 Algoritmus Knuth-Morris-Pratt

Algoritmus *KMP* (Knuth-Morris-Pratt) vychází podle [8] ze studia časových složitostí algoritmu hrubé síly (viz kapitola 3.1.2) a algoritmu Boyer-Moore (viz kapitola 3.1.3) u konkrétních případů jako je příklad 3.4, u kterých si lze všimnou velké neefektivnosti. Konkrétně, pokud vyhledávací algoritmus provede celou řadu porovnání znaků při testování potencionálních umístění P v textu T a narazí na znak $P[j]$, který se nerovná $T[i]$, potom zahodí všechny informace, které získal provedenými porovnáními a začne znovu s novým umístěním P. Tomu však zabraňuje algoritmus KMP, který díky tomu dosahuje časové složitosti $O(n+m)$, která je dostačující i pro nejhorší případ. V nejhorším případě musí KMP jako ostatní vyhledávací algoritmy vyzkoušet alespoň jednou všechny znaky v textu T a všechny znaky v řetězci P.

Chybová funkce

Hlavní myšlenkou algoritmu KMP je předzpracování řetězce P a výpočtu *chybové funkce* f (angl. failure function) (viz tabulka na obrázku 3.5), která určuje správný posun P tak, aby byla dříve provedená srovnání využita v co největším možném rozsahu. Chybová funkce $f(j)$ definuje délku nejdelšího prefixu (viz kapitola 3.1.1), který je suffixem podřetězce $P[1..j]$ (všimněte si, že nebyl použit podřetězec $P[0..j]$). Pro chybovou funkci musí platit, že $f(0) = 0$. Chybová funkce tedy „kóduje“ opakující se podřetězce uvnitř řetězce P.

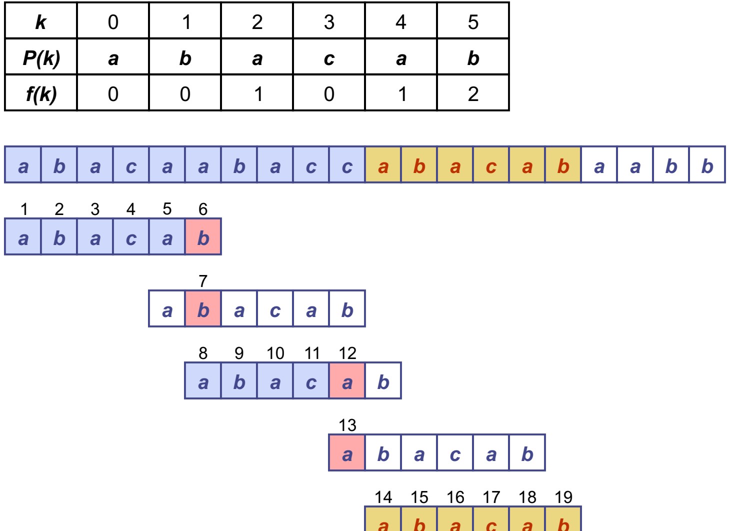
Konstrukce chybové funkce f je velice podobná algoritmu KMP, protože porovnáváme řetězec P sám se sebou stejným způsobem jako v KMP. Pokaždé, když máme dva stejné znaky, potom stanovíme hodnotu $f(i) = j+1$. Za zmínku stojí fakt, že při konstrukci je vždy $i > j$, proto je v případě po-

třeby $f(j-1)$ vždy definováno. Časová složitost konstrukce chybové funkce je $O(m)$.

Princip algoritmu

Algoritmus KMP postupně prochází text T a porovnává ho s řetězcem P. Pokaždé, kdy narazí na shodu znaků, zvýší pozici v řetězci i v textu. Pokud se ale znaky nerovnají a pozice v řetězci P je větší než 0, potom algoritmus využije chybovou funkci k určení nové pozice v P, od které bude pokračovat v porovnávání P a T. Pokud je však pozice v řetězci P rovna 0 (znaky se nerovnají na začátku řetězce P), potom je pouze posunuta pozice v textu T. Tento postup se opakuje, dokud není nalezena pozice P v T, nebo dokud algoritmus nedojde na konec textu T.

Příklad 3.7: Na obrázku 3.5 můžeme vidět ukázku vyhledávání řetězce P = `"abacab"` v textu T = `"abacaabaccabacabaabb"` algoritmem KMP. Z obrázku je vidět, že bude provedeno celkem 5 pokusů, při kterých dojde k 19 porovnání znaků. Modře označené znaky jsou shodné, oproti tomu červeně označené znaky shodné nejsou. Zlatě je označen nalezený řetězec.



k	0	1	2	3	4	5
P(k)	**a**	**b**	**a**	**c**	**a**	**b**
f(k)	0	0	1	0	1	2

Obrázek 3.5: Ukázka vyhledávání řetězce algoritmem KMP

3.1.5 Algoritmus Rabin-Karp

Tento algoritmus podle [4] vychází z myšlenky převedení řetězců na celá čísla, díky kterým mohou být tyto řetězce snadno porovnány.

V Javě jsou řetězce tvořeny polem znaků, kde tyto znaky mohou být interpretovány jako celá čísla. Hodnoty jednotlivých znaků se mohou lišit v závislosti na typu použitého kódování (např. ASCII, Unicode). Z toho vyplývá, že si v Javě můžeme představit řetězec jako pole celých čísel. K převedení pole celých čísel (řetězce) na jedno celé číslo využijeme *hašovací funkci*, kterou označíme h.

Vzhledem k tomu, že řetězce nejsou jednoduchým prvkem, ale jsou to pole znaků, potom jejich porovnání není tak jednoduché jako porovnání dvou čísel. Pokud chceme ověřit, zda řetězec A o délce n je roven řetězci B o délce n, potom bychom museli porovnat všechny znaky z obou řetězců a muselo by platit, že $A[i] = B[i]$ pro všechna $i \in \langle 0, n)$. Z toho vyplývá, že porovnání řetězců o délce n má časovou složitost $O(n)$.

Hašování řetězce o délce n prochází všechny znaky v řetězci, a proto má časovou složitost $O(n)$. Pokud bychom využili jednoduché hašování řetězce k vyhledání řetězce P o délce m v textu T o délce n, potom bychom museli v každé zkoumaném podřetězci $S[i..i+m]$ vypočítat $h(S)$ a porovnat s $h(P)$. V tomto případě by časová složitost hledání byla $O(mn)$.

K vylepšení časové složitosti algoritmu se využívá metoda *rolling hash*, která výrazně zjednodušuje výpočet hodnoty h využitím již vypočtených hodnot. Například pokud máme řetězec `"algoritmus"`, ve kterém budeme vyhledávat řetězec o délce 5, potom první dva zkoumané podřetězce budou `"algor"` a `"lgori"`. Tento způsob hašování zjednoduší výpočet využitím toho, že tyto dva řetězce mají společný podřetězec `"lgor"`, který tvoří valnou část obou řetězců.

Výpočet hodnoty hašovací funkce

V [3] je uvedeno, že pro výpočet hodnoty h pro libovolný řetězec použijeme rovnici:

$$h(S) = \sum_{i=1}^{m} d^{m-i} S[i] = S[m] + dS[m-1] + \ldots + d^{m-1} S[1] \pmod q \quad (3.1)$$

kde:

- S je řetězec, pro který bude vypočtena hodnota h;
- q je libovolné prvočíslo;
- d je velikost použité abecedy Σ.

Rovnici 3.1 lze upravit použitím Hornerovo schéma:

$$h(S) \equiv d\left(\sum_{i=1}^{m-1} d^{m-i-1} S[i]\right) + S[m] \equiv dh_{m-1}(S[1..m-1]) + S[m] \pmod{q} \tag{3.2}$$

Tím získáme rovnici pro výpočet hodnoty h využívající metodu rolling hash pro zkoumané podřetězce v T:

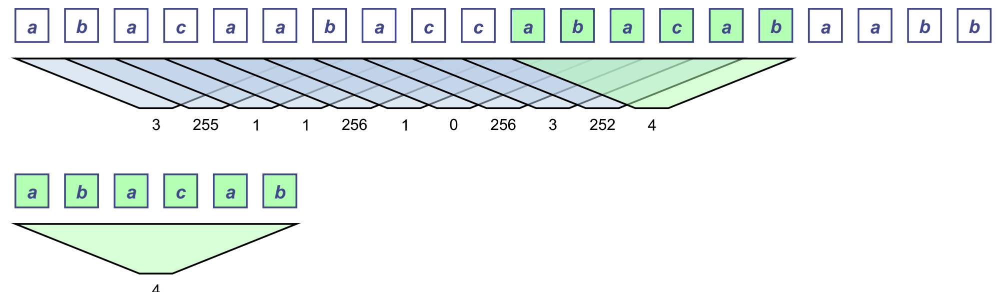
$$h(T[i..i{+}m]) \equiv d\Big(h\big(T[i{-}1..i{-}1{+}m]\big) {-} d^{m-i-1} T[i{-}1]\Big) {+} T[i{+}m] \pmod{q} \tag{3.3}$$

Pro výpočet hodnoty h prvního zkoumaného podřetězce $T[1..m]$ a hledaného řetězce S použijeme rovnici 3.1.

Při nalezení shody mezi hodnotami hašovací funkce zkoumaného podřetězce a hledaného řetězce S se provede kontrola, zda se nejedná o falešnou shodu, tj. různé řetězce, které mají stejnou hodnotu h.

Tento algoritmus se často používá ke kontrole plagiátorství. Díky hašování metodou rolling hash dosahuje časové složitosti $O(n{+}m)$. Pro nejhorší případ má tento algoritmus časovou složitost $O(m(n-m+1))$.

Příklad 3.8: Na obrázku 3.6 můžeme vidět ukázku vyhledávání řetězce P `= "abacab"` v textu T `= "abacaabaccabacabaabb"` algoritmem Rabin-Karp.



Obrázek 3.6: Ukázka vyhledávání řetězce algoritmem Rabin-Karp

3.2 Hledání podobnosti řetězců

Podle [8] je *hledání podobnosti dvou řetězců* X a Y jedním z problémů zpracování textu, který se vyskytuje v genetice nebo softwarovém inženýrství.

V případě genetiky hledáme podobnost mezi dvěma řetězci DNA, například pokud hledáme genetickou příbuznost dvou jedinců nalezením nejdelšího společného řetězce v DNA.

V případě využití v softwarovém inženýrství můžeme například porovnávat dvě verze zdrojového kódu, u kterých budeme hledat provedené změny z jedné verze na druhou.

Určování podobnosti mezi dvěma řetězci je považováno v operačních systémech Unix/Linux za základní funkci, a proto obsahují program `diff`, který porovnává textové soubory.

3.2.1 Nalezení nejdelší společné podsekvence znaků

V [7, 3] je uvedeno, že se problém *nalezení nejdelší společné podsekvence znaků* (angl. longest common subsequence, zkr. LCS) využívá při hledání podobností v textech. Zvláštním využitím tohoto algoritmu je hledání shody mezi řetězci DNA.

Podle [8] existuje několik různých způsobů, kterými můžeme definovat podobnost mezi řetězci $X = x_0x_1x_2 \dots x_{n-1}$ a $Y = y_0y_1y_2 \dots y_{n-1}$. Jedním z nich je *podsekvence znaků* (angl. subsequence), která je pro jakýkoliv řetězec S definována jako libovolný řetězec ve tvaru $s_{i_1}s_{i_2} \dots s_{i_k}$ kde $i_j < i_{j+1}$. To

znamená, že se jedná o řetězec obsahující libovolné znaky z S, u kterých musí být dodrženo pořadí. Například řetězec `AAAG` je podsekvencí řetězce `CGATAATTGAGA`. Je nutné si všimnout, že se podsekvence liší od podřetězce (viz kapitola 3.1.1).

Příklad 3.9: Řetězec `"bcb"` je nejdelší společnou podsekvencí znaků pro řetězce `"abcb"` a `"bdcab"`.

Dále [8] uvádí, že cílem tohoto problému je nalezení řetězce S, který je nejdelší podsekvencí vstupních řetězců X o délce m a Y o délce n, jejíž znaky jsou z abecedy Σ (např. abeceda využívaná v genetice je $\Sigma = \{A, C, G, T\}$).

Příklad 3.10: V [7] je uvedeno, že při odhalování plagiátorství nebo identifikování autorů anonymních děl musíme najít nejdelší fráze, které jsou společné pro zkoumané dokumenty. Vzhledem k tomu, že si můžeme jednotlivé věty dokumentu představit jako řetězce, potom můžeme k řešení využít algoritmus LCS.

Podle [8] je jedním ze způsobu řešení tohoto problému použití algoritmu hrubé síly, který vytvoří výčet všech podsekvencí řetězce X a vybere nejdelší podsekvenci, která je zároveň podsekvencí řetězce Y. Jelikož každý znak v X náleží nebo nenáleží podsekvenci, potom potencionálně existuje 2^m různých podsekvencí řetězce X, z nichž každá vyžaduje n porovnání, aby algoritmus zjistil, že je podsekvencí Y (časová složitost $O(n)$). Z toho vyplývá, že se jedná o algoritmus s exponenciální časovou složitostí $O(2^m n)$, a proto je toto řešení velice neefektivní.

Problém LCS můžeme řešit mnohem rychleji než s exponenciální složitostí s použitím *dynamického programování*, které umožňuje snížit časovou exponenciální časovou složitost na polynomiální.

Nalezení LCS dynamickým programováním:

V [3] je uvedeno, že nejprve nalezneme délku LCS a podél „cesty“, kterou budeme procházet, si budeme nechávat značky, které nám pomohou nalézt nejdelší společnou podsekvenci znaků. Postup pro nalezení LCS je:

1. Vytvoříme si dvourozměrné pole c o velikosti $m \times n$, kde m je délka řetězce X a n je délka řetězce Y.
2. Začneme na pozici $i = j = 0$ (prázdné podsekvence X a Y).

3. Protože X_0 a Y_0 jsou prázdné řetězce, je jejich LCS vždy prázdná (tj. $c[0,0] = 0$).

4. LCS prázdného řetězce a libovolného jiného řetězce je také prázdná, a tak pro každé i a j platí $c[0,j] = c[i,0] = 0$.

5. Když určujeme hodnotu $c[i,j]$, tak uvažujeme dva případy:

   (a) Znak řetězce X se shoduje se znakem Y (tj. $x[i] = y[j]$) a délka LCS(X_i, Y_j) je rovna délce LCS kratších řetězců X_{i-1} a Y_{i-1}, zvětšená o 1;

   (b) Znak řetězce X se neshoduje se znakem Y (tj. $x[i]! = y[j]$), tudíž se délka LCS(X_i, Y_j) nezvětší a zůstává shodná jako předtím (tj. maximum z LCS(X_i, Y_{j-1}) a LCS(X_{i-1}, Y_j)).

3.2.2 Nalezení nejkratšího společného „nadřetězce“

Definice: Nechť máme řetězce X a Y, potom řetězec Z je podle [3] *nadřetězec* (angl. supersequence nebo superstring) řetězců X a Y pokud jsou řetězce X a Y podsekvencí Z.

Příklad 3.11: Řetězce `"abbc"` a `"abcb"` jsou nejkratšími společnými nadřetězci řetězců `"abc"` a `"abb"`.

Problém *nalezení nejkratšího společného nadřetězce* (angl. shortest common superstring, zkr. SCS) je velice podobný LCS (viz kapitola 3.2.1). Stejně jako LCS, i problém SCS budeme řešit dynamickým programováním.

Nalezení SCS dynamickým programováním:

V [3] je uvedeno, že stejně jako u LCS nejprve nalezneme délku SCS a podél „cesty“, kterou budeme procházet, si budeme nechávat značky, které nám pomohou nalézt výsledný nejkratší nadřetězec. Postup pro nalezení SCS je:

1. Vytvoříme si dvourozměrné pole c o velikosti $m \times n$, kde m je délka řetězce X a n je délka řetězce Y.

2. Začneme na pozici $i = j = 0$ (prázdné podsekvence X a Y).

3. Protože X_0 a Y_0 jsou prázdné řetězce, je jejich SCS vždy prázdná (tj. $c[0,0] = 0$).

4. SCS prázdného řetězce a libovolného jiného řetězce je rovna danému řetězci, a tak pro každé i a j je délka $c[i, 0] = i$ a $c[0, j] = j$.

5. Když určujeme hodnotu $c[i, j]$, tak uvažujeme dva případy:

   (a) Znak řetězce X se shoduje se znakem Y (tj. $x[i] = y[j]$) a délka SCS(X_i, Y_j) je rovna délce LCS kratších řetězců X_{i-1} a Y_{i-1}, zvětšená o 1;

   (b) Znak řetězce X se neshoduje se znakem Y (tj. $x[i]! = y[j]$) a délka SCS(X_i, Y_j) je daná minimální hodnotou dvojice SCS(X_i, Y_{j-1}) a SCS(X_{i-1}, Y_j).

3.3 Komprese dat

3.3.1 Run Length Encoding

Podle [6, 3] je tato jednoduchá bezztrátová metoda založena na kompresi opakujících se znaků popř. bitů vstupního souboru. Hlavním principem této metody je zkrátit opakující se informace na dvojici, která je tvořena počtem opakování a opakujícím se symbolem (např. `"aaaa"` na `"4a"`). Tento algoritmus se nejčastěji se používá pro kompresi obrazové informace (obrazový formát BMP) nebo zřídka pro kompresi textů.

Metody komprese:

- *Přímá* – u každého symbolu je uveden počet opakování (např. pro vstupní řetězec `AAAABBC` je výstupem `4A2B1C`). Nevýhodou je, pokud se znaky neopakují často, potom nedochází ke kompresi, ale naopak k prodloužení kódovaného souboru.

- *Pomocí escape sekvencí* – kódují se pouze opakující se sekvence delší než 3 znaky, kratší sekvence se zapisují přímo do výstupního souboru (např. pro vstupní řetězec `AAAABBC` je výstupem `#4ABBC`). Tato metoda neprodlužuje soubor, protože zachovává nekomprimovatelná data v původní podobě. V případě použití této metody je nutné zvolit symbol pro escape znak, který se nevyskytuje v komprimovaném souboru.

Příklad 3.12: Máme vstupní 16 bitový řetězec `0000000001111111`, který obsahuje 9 nul a 7 jedniček. V případě, že k zakódování počtu opakování použijeme 4 bity (9 = `1001`, 7 = 0111), potom získáme řetězec `1001001111` o délce 10 bitů. To znamená, že jsme vstupní řetězec zkomprimovali o 37,5%.

3.3.2 Lempel-Ziv-Welch

Podle [6, 3] je komprimační algoritmus *Lempel-Ziv-Welch* (zkr. LZW) slovníkovou metodou komprese, která komprimuje vstupní soubor a vytváří kódovací tabulku (slovník) během průchodu souborem. Jedná se tedy o jednoprůchodovou metodu, která nevyžaduje předběžnou analýzu vstupního souboru.

Algoritmus LZW během komprese vyhledává opakující se posloupnosti znaků (řetězce), kterým přiřadí jednoznakový kód a vloží je do kódovací tabulky. Tyto posloupnosti znaků mohou být libovolně dlouhé, a proto je možné kódovat slabiky, slova nebo dokonce i celé věty. Kódované znaky musí mít délku (počet bitů) větší, než je délka znaků ze vstupní abecedy Σ (použité kódování ve vstupním souboru). Pokud například komprimujeme vstupní soubor v kódování ASCII (7 bitů), potom je vhodné pro kódované znaky použít 8 bitů. V praxi se pro kódové znaky obvykle využívá 12 bitů.

Před začátkem komprese je nutné vytvořit kódovací tabulku a vložit do ní všechny znaky a jejich kódy z použitého kódování (např. ASCII, Unicode) ve vstupním souboru. Pro zmiňované kódování ASCII by se jednalo o 128 znaků s hodnotami kódů od 0 do 127, tzn. že nově kódované znaky budou začínat hodnotou 128.

Během komprimace vstupního souboru provádíme následující kroky (viz obrázek 3.7):

1. Nalezneme nejdelší řetězec S v kódovací tabulce, který je prefixem (viz kapitola 3.1.1) nenačteného vstupu.
2. Zapíšeme do výstupního souboru kódovací hodnotu řetězce S.
3. Načteme jeden znak c, který následuje po S, ze vstupu.
4. Vložíme do kódovací tabulky nový řetězec $S + c$.

S	c	výstup
A	B	A(65)
B	C	B(66)
C	A	C(67)
A	B	-
AB	C	AB(256)
C	A	-
CA	-	CA(258)

kód	posloupnost
0..255	znaky
256	AB
257	BC
258	CA
259	ABC

Obrázek 3.7: Ukázka komprese řetězce `"ABCABCA"` algoritmem LZW.

Při dekompresi se stejně jako u komprese vytváří kódovací tabulka během průchodu vstupním souborem, pomocí které jsou zakódované znaky nahrazeny původními řetězci. Stejně jako komprese, před začátkem je nutné vytvořit kódovací tabulku a vložit do ní všechny znaky a jejich kódy z použitého kódování (např. ASCII, Unicode). Poté načteme první znak ze vstupního souboru a uložíme ho do řetězce *val* a provádíme následující kroky (viz obrázek 3.8):

1. Zapíšeme řetězec *val* do výstupního souboru.
2. Načteme zakódovaný znak x ze vstupu.
3. Do řetězce S vložíme hodnotu pro zakódovaný znak x.
4. Vložíme do kódovací tabulky nový řetězec $val + c$, kde c je první znak z S.
5. Nahradíme řetězec *val* řetězcem S.

val	x	S	c	výstup
A(65)	-	-	-	A
A(65)	B(66)	B	B	B
B(65)	C(67)	C	C	C
C(67)	AB(256)	AB	A	AB
AB(256)	CA(258)	CA	C	CA

kód	posloupnost
0..255	znaky
256	AB
257	BC
258	CA
259	ABC

Obrázek 3.8: Ukázka dekomprese vstupního řetězce `"65 66 67 256 258"` algoritmem LZW.

Při dekompresi se může stát, že v kroku 3 nenalezneme řetězec pro zakódovaný znak x, potom pokud je x posledním znakem ve vstupním souboru, potom volíme $S = val + v$, kde v je prvním znakem řetězce val, v opačném případě znak x je neplatný a vstupní soubor nelze dekódovat.

Výhodou tohoto algoritmu je, že si během dekomprese vytváří stejnou kódovací tabulku, která byla použita pro komprimaci, a proto nemusí být součástí komprimovaných dat.

4 Návrh GUI

Pro výuku algoritmů a datových struktur je vhodné, aby GUI zobrazovalo jejich zdrojový kód a v něm označovalo právě vykonávanou řádku. Zároveň by mělo znázorňovat práci algoritmu nebo aktuální stav datové struktury na obrázku nebo jiným způsobem. Uživatel by tak měl snadněji pochopit jejich princip.

Součástí GUI dále musí být přehledně uspořádané *kontrolky* (textová pole, tlačítka atd.), které umožní uživateli pracovat s algoritmem nebo datovou strukturou. Tyto kontrolky často bývají uspořádány do formulářů.

Dále je nutné, aby GUI obsahovalo přehledné menu pro snadný výběr algoritmů či datových struktur.

Problém návrhu GUI jsem rozdělil do čtyř částí:

1. *návrh menu*;
2. *návrh kontrolek pro práci s jednotlivými algoritmy a datovými strukturami*;
3. *návrh zobrazení stavu datové struktury či práce algoritmu*;
4. *návrh zobrazení zdrojového kódu algoritmu nebo datové struktury.*

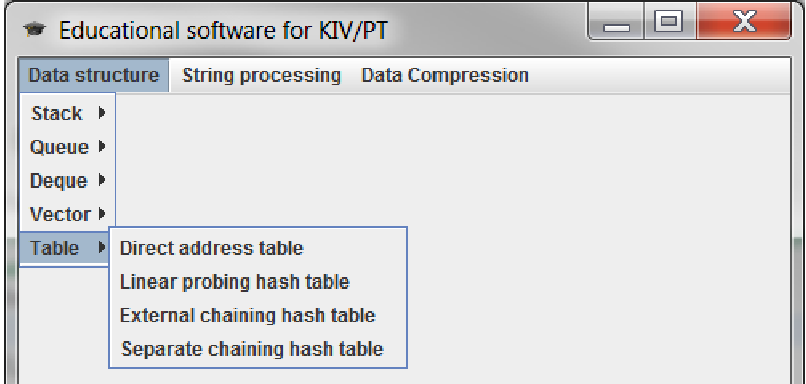
Návrh kontrolek pro práci s jednotlivými algoritmy a datovými strukturami zde nebude uveden, protože pro každý algoritmus nebo datovou strukturu byl navržen jiný formulář. Jednotlivé formuláře a jejich funkčnost si lze prohlédnout v uživatelské příručce (viz příloha E).

4.1 Návrh menu

Vzhledem k tomu, že vyučované algoritmy a různé implementace datových struktur jsou rozděleny do *kategorií* (skupin), je proto vhodné využít toto rozdělení při návrhu menu. Pro tento program jsem se rozhodoval mezi třemi návrhy menu.

4.1.1 Lišta menu

První možností návrhu menu bylo vytvoření *lišty s roletkovými nabídkami* (lišta menu), ve které by se nacházely jednotlivé kategorie (viz obrázek 4.1). Tato lišta by se nacházela v horní části GUI a roletky by obsahovaly položky s algoritmy či druhy implementace datových struktur.



Obrázek 4.1: Ukázka lišty s roletkovými nabídkami pro vyučované algoritmy a datové struktury

Při volbě algoritmu či druhu implementace datové struktury by se v případě využití tohoto druhu menu měnil obsah okna aplikace.

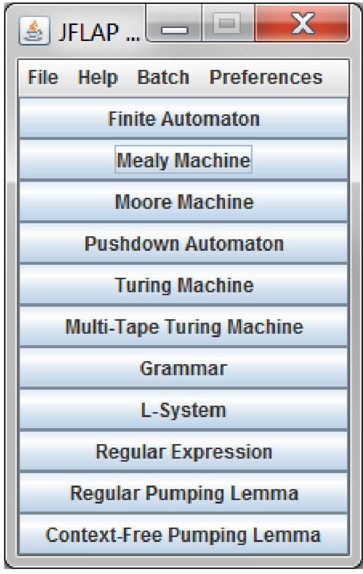
Výhodou tohoto menu je jeho úsporná velikost (pouze lišta v okně aplikace) a možnost využití klávesových zkratek, které by měly být shodné pro různé jazykové sady GUI. Další výhodou této lišty je, že každá roletka může obsahovat další roletky, tzn. že každá kategorie může obsahovat podkategorie.

Nevýhodou tohoto menu je, že uživatel na první pohled nevidí, které algoritmy a druhy implementace datových struktur jsou v programu zahrnuty. Další nevýhoda vzniká v případě, kdy menu obsahuje velké množství položek (kategorií), kvůli kterým se „nevejde“ do okna aplikace. V tomto případě se začnou položky překrývat a menu se stává nepřehledným.

Tuto možnost menu jsem nepoužil z důvodu, že v programu zatím nejsou zahrnuty všechny algoritmy a počet kategorií v menu se bude měnit. Menu by se v průběhu vývoje mohlo stát nepřehledným. Je ale možné, že se tento druh menu stane součástí některé z dalších verzí programu.

4.1.2 Menu v samostatném okně

U tohoto druhu menu je při spuštění programu otevřeno okénko obsahující menu, které je obvykle tvořeno tlačítky ležícími pod sebou (viz obrázek 4.2).



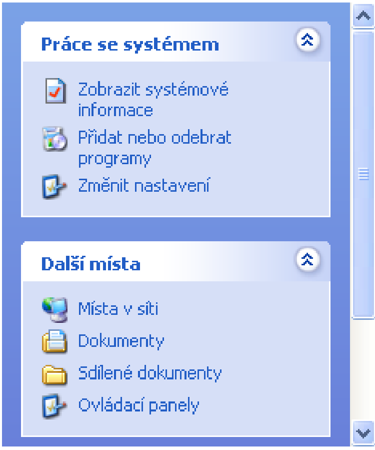
Obrázek 4.2: Ukázka menu v samostatném okně programu JFLAP, který slouží k práci s automaty, gramatikami, formálními jazyky atd.

Při volbě některé položky menu je otevřeno nové okno, které by v tomto případě obsahovalo komponenty pro výuku zvoleného algoritmu či druhu implementace datové struktury. Poté je okénko obsahující menu nejčastěji skryto. V některých aplikacích zůstává okénko s menu viditelné a uživatel může zvolit více možností najednou (otevření více oken). To by mohla být výhoda, pokud by chtěl uživatel porovnávat činnost různých algoritmů najednou.

Využití tohoto menu může přinášet problém nepřehlednosti, kdy uživatel musí přeskakovat mezi okny kvůli volbě některé z položek. Navíc toto menu je spíše vhodné pro malý počet položek.

4.1.3 Podokno úloh

Tento druh menu je tvořen kategoriemi obsahujícími položky. Tyto kategorie jdou obvykle „zabalit“ popř. „rozbalit“, tzn. že jsou skryty nebo zobrazeny položky v této kategorii. Tento druh menu se vyskytuje v mnoha aplikacích, nejznámější z nich je průzkumník operačního systému Windows XP.



Obrázek 4.3: Ukázka podokna úloh v průzkumníku operačního systému Windows XP

Nevýhodou tohoto menu je jeho větší velikost, která je závislá na délce názvů položek (v případě, že je programově nezkracujeme) a na počtu položek a kategorií v menu.

Tato možnost menu mi přišla jako nejvhodnější pro tuto aplikaci. Pro odstranění nevýhod bude toto menu umístěno do scrollovatelného panelu, který odstraní potíže s počtem kategorií a položek v menu. Dále bude uživateli umožněno skrýt menu a tím zmenšit jeho šířku.

4.2 Návrh zobrazení práce algoritmu

Jak už bylo uvedeno, v GUI bude zobrazován obrázek, který bude reprezentovat práci algoritmu a který se bude během činnosti algoritmu měnit. Pro lepší názornost budou tyto obrázky podobné obrázkům v přednáškách předmětu (viz [3]).

Vzhledem k tomu, že obrázky reprezentující práci algoritmu mohou mít různou velikost, je vhodné, aby byly vykresleny do speciálního panelu (viz kapitola 4.2.1).

4.2.1 Panel pro obrázek

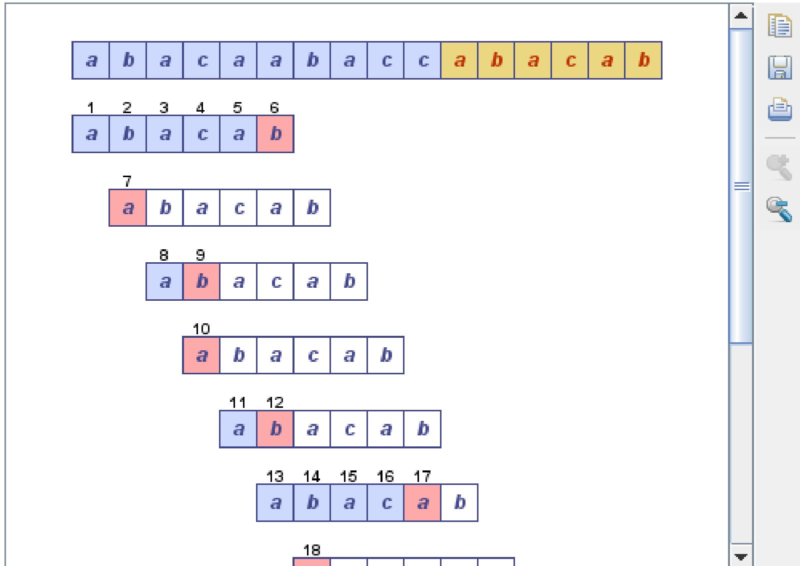
Jak už bylo uvedeno, obrázky reprezentující práci algoritmu mohou mít různou velikost a může se stát, že se nevejdou do GUI. Proto bude obrázek vkládán do scrollovatelného panelu. Obrázek bude v tomto panelu zarovnán nahoru a na střed.

Součástí tohoto panelu bude nástrojová lišta, která bude obsahovat následující nástroje pro práci s obrázkem:

- *Nástroj pro zkopírování obrázku do systémové schránky*, který umožní uživateli zkopírování obrázku do systémové schránky, ze které může být vložen do libovolného grafického programu.
- *Nástroj pro uložení obrázku do souboru*, který umožní uživateli uložit obrázek do libovolné složky na svém disku. Pro uložení bude použito knihovní dialogové okno pro ukládání souborů, ve kterém si uživatel bude moct zvolit mezi různými formáty obrázků, které jsou podporovány jazykem Java. Podporovanými formáty jsou: `BMP`, `GIF`, `JPG` popř. `JPEG` a `PNG`.
- *Nástroj pro tisk obrázku*, který bude sloužit ke snadnému vytištění obrázku.
- *Nástroj pro zvětšení a zmenšení obrázku*, který uživateli umožní zmenšit obrázek (např. pokud je moc veliký). Tento nástroj umožní zvětšení pouze zmenšeného obrázku, tzn. že neumožní zvětšení přes 100%.

Při návrhu umístění této lišty jsem se rozhodoval mezi levou a horní stranou panelu. Obě varianty nevypadaly esteticky dobře, protože lišta „splývala“ s okolními komponenty panelu. Proto jsem se rozhodl umístit lištu na pravou stranu panelu, na které je lišta vizuálně oddělena.

Výsledný panel po dokončení návrhu si lze prohlédnout na obrázku 4.4.



Obrázek 4.4: Ukázka navrženého panelu s obrázkem vyhledávání řetězce algoritmem hrubé síly

4.3 Návrh zobrazení stavu datové struktury

Pro zobrazení stavu datové struktury budu využívat tři komponenty:

1. *Seznam prvků v datové struktuře*, který bude obsahovat prvky datové struktury v pořadí, ve kterém jsou uloženy.
2. *Tabulka operací s datovou strukturou*, do které se budou vkládat záznamy o provedených operací s datovou strukturou.
3. *Obrázek znázorňující datovou strukturu*, který bude vložen ve stejném panelu jako obrázek pro zobrazení práce algoritmu (viz kapitola 4.2.1).

4.3.1 Navržené komponenty pro zobrazení stavu

Navržené komponenty pro seznam prvků v datové struktuře a pro tabulku operací s datovou strukturou budou ukázány na příkladě 4.1. Panel obsahující obrázek, který znázorňuje stav datové struktury bude stejný jako panel na obrázku 4.4.

Příklad 4.1: Tabulka 4.1 obsahuje ukázku operací se zásobníkem a jejich vliv na původně prázdný zásobník S.

Operace	**Výstup**	$\boldsymbol{S}$
push(5)	–	(5)
push(3)	–	(5, 3)
pop()	3	(5)
push(7)	–	(5, 7)
pop()	7	(5)
top()	5	(5)
pop()	5	()
pop()	chyba	()
isEmpty()	true	()
push(9)	–	(9)
push(7)	–	(9, 7)
size()	2	(9, 7)
push(3)	–	(9, 7, 3)
push(5)	–	(9, 7, 3, 5)
pop()	5	(9, 7, 3)

Tabulka 4.1: Ukázka operací se zásobníkem S

Po provedení poslední operace z příkladu 4.1 budou v zásobníku 3 prvky. Komponenta reprezentující seznam prvků tohoto zásobníku je na obrázku 4.5.

Obrázek 4.5: Ukázka komponenty pro seznam prvků v datové struktuře

Na obrázku 4.6 je tabulka reprezentující tabulku operací s datovou strukturou. Tato tabulka je shodná s tabulkou 4.1

#	Operation	Output	S
7	pop()	5	()
8	pop()	"error"	()
9	isEmpty()	true	()
10	push(9)	-	(9)
11	push(7)	-	(9, 7)
12	push(3)	-	(9, 7, 3)
13	push(5)	-	(9, 7, 3, 5)
14	pop()	5	(9, 7, 3)

Obrázek 4.6: Ukázka komponenty pro tabulku operací s datovou strukturou

4.4 Návrh zobrazení zdrojového kódu

Jak už bylo uvedeno, v GUI bude zobrazován zdrojový kód algoritmu či datové struktury. V tomto zdrojovém kódu bude označována právě vykonávaná řádka. Tyto zdrojové kódy budou v programovacím jazyce Java, protože algoritmy a datové struktury jsou v předmětu Programovací techniky vyučovány v tomto jazyce.

Zdrojový kód bude umístěn v textovém poli, které bude podobné textovému poli vývojových prostředí, tzn. že bude obsahovat lištu s čísly řádek a bude zvýrazňovat syntaxi jazyka Java.

4.4.1 Umístění textového pole se zdrojovým kódem

Pro umístění textové pole obsahující zdrojový kód algoritmu či datové struktury jsem vybíral mezi dvěma možnostmi.

První možností bylo umístění textového pole do okna aplikace. Nevýhodou této možnosti je, že obvyklá maximální délka jedné řádky zdrojového kódu je 80 znaků, a proto by velikost textového pole pro zdrojový kód byla příliš veliká. Pro menší rozlišení obrazovek bych musel v textovém poli volit menší písmo a zdrojový kód by se mohl stát hůř čitelným. Pokud bych zmenšil velikost textového pole, potom by nebyl vidět celý zdrojový kód.

Kvůli nevýhodám první možnosti jsem zvolil druhou možnost, kterou bylo umístění textového pole do nového okna. Tato možnost přináší mnoho výhod:

1. Pokud uživatel nemá zájem vidět zdrojový kód, může okno se zdrojovým kódem kdykoliv uzavřít (okno lze opětovně znovuotevřít).

2. Uživatel může zvolit místo, na kterém chce mít okno se zdrojovým kódem.

3. Snadná volba velikosti textového pole se zdrojovým kódem změnou velikosti okna.

4.4.2 Nástroje pro práci se zdrojovým kódem

Vzhledem k tomu, že ve zdrojovém kódu bude zvýrazňována aktuálně prováděná řádka, je velice vhodné, aby uživatel mohl měnit rychlost činnosti algoritmu popř. některé z operací s datovou strukturou. Rychlosti činnosti lze měnit délkou prodlevy mezi příkazy (řádky) zdrojového kódu.

Dalším důležitým nástrojem je pozastavení činnosti algoritmu popř. prováděné operace datové struktury. Po pozastavení činnosti bude uživateli nabídnuta možnost krokování ve zdrojovém kódu nebo opětovné spuštění „automatické“ činnosti. Možnost krokování zdrojového kódu je velice užitečná funkce, která připomíná *debugger*, tj. nástroj pro ladění programů, ve kterém uživatel může laděný program libovolně pozastavovat a krokovat.

Pro práci se zdrojovým kódem jsou vhodné další nástroje:

- *Nástroj pro zkopírování zdrojového kódu do systémové schránky*, který urychlí uživateli zkopírování zdrojového kódu do systémové schránky, ze které může být vložen například do vývojového prostředí.

- *Nástroj pro uložení zdrojového kódu do souboru*, který umožní uživateli uložit zdrojový kód do libovolné složky na svém disku. Pro uložení bude použito knihovní dialogové okno pro ukládání souborů, ve kterém bude nutné zabránit uživateli měnit název souboru, protože název souboru musí být shodný s názvem veřejné (`public`) třídy.

- *Nástroj pro tisk zdrojového kódu*, který bude sloužit ke snadnému vytištění zdrojového kódu.

- *Nástroje pro možnost změny stylu zvýraznění syntaxe*, které budou sloužit k změně stylu zvýraznění syntaxe. Jednotlivé styly budou převzaty ze známých vývojových prostředí.

Nástroje pro práci se zdrojovým kódem budou umístěny do nástrojové lišty (toolbaru), která se bude nacházet v horní části okna pro zdrojový kód, tzn. nad textovým polem pro zdrojový kód.

4.4.3 Navržené okno pro zobrazení zdrojového kódu

Výsledné okno pro zobrazení zdrojového kódu algoritmu či datové struktury si lze prohlédnout na obrázku 4.7.

Source Code

Delay: 500 ms

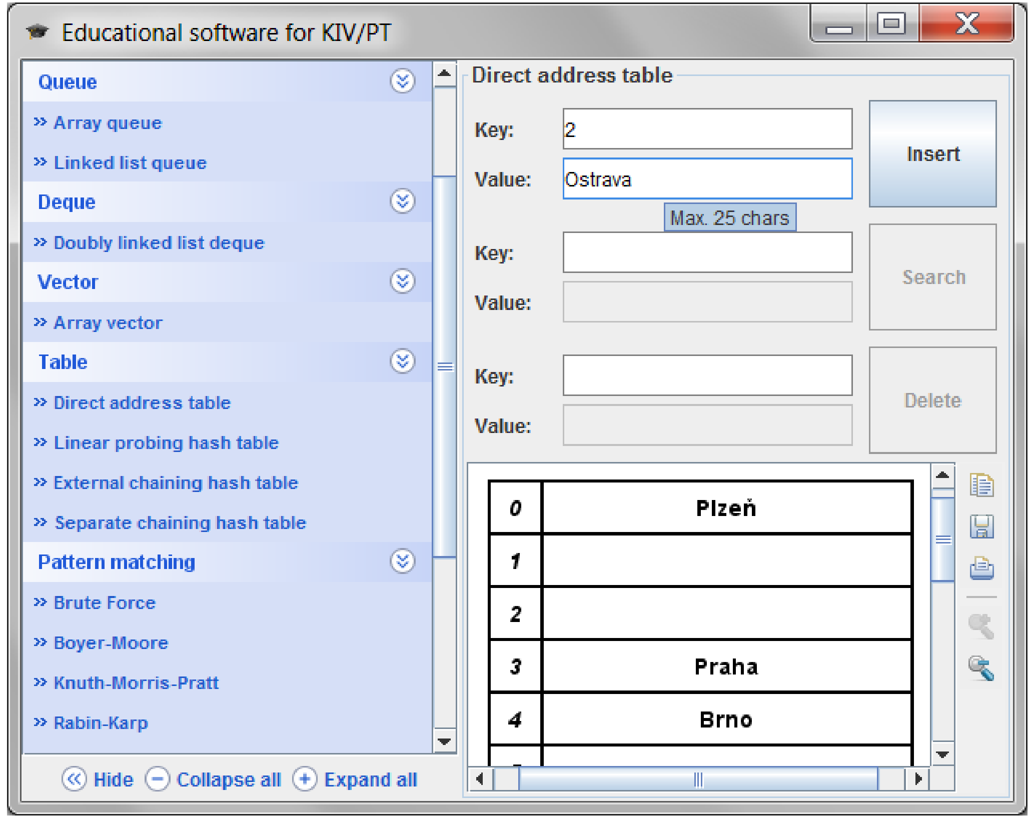
```
1  package kiv.pt.patternmatching;
2
3  /**
4   * Algoritmus "hrubé síly" (Brute Force Algorithm) pro vyhledávání řetězců.
5   *
6   * @author Mr.FrAnTA (Michal Dékány)
7   */
8  public class BruteForce {
9      public static int bruteForceMatch(String text, String pattern) {
10         int n = text.length();
11         int m = pattern.length();
12         for (int i = 0; i < (n - m + 1); i++) {
13             int j = 0;
14             while ((j < m) && (text.charAt(i + j) == pattern.charAt(j))) {
15                 j++;
16             }
17
18             if (j == m) {
19                 return i;
20             }
21         }
22
23         return -1;
24     }
25 }
```

Obrázek 4.7: Ukázka navrženého okna se zobrazeným zdrojovým kódem pro vyhledávání řetězce algoritmem hrubé síly

4.5 Navržené hlavní okno GUI

Navržené hlavní okno programu (viz obrázek 4.8) je rozděleno do dvou částí. V levé části se nachází menu pro výběr algoritmu či druhu implementace datové struktury. V pravé části se nachází panel pro algoritmus či datovou strukturu, který obvykle obsahuje formulář pro práci s algoritmem či datovou strukturou a panel pro zobrazení obrázku.

Při výběru položky v menu se bude měnit panel v pravé části okna a zároveň dojde ke změně zdrojového kódu v okně pro zobrazení zdrojového kódu.



Obrázek 4.8: Ukázka hlavního okna GUI

5 Implementace GUI

Výše navržené GUI jsem implementoval v Javě s využitím knihovny *Swing*, která poskytuje aplikační rozhraní pro tvorbu a obsluhu GUI.

5.1 Implementace modulů

Pro implementaci modulu (panelu pro algoritmus či datovou strukturu) je nutné odědit abstraktní třídu `AlgorithmPanel`. Tato třída je v balíčku `kiv.pt.algorithm` a obsahuje abstraktní metody:

- `clear()` – ruší provedené změny v panelu (v algoritmu či datové struktuře);
- `getSourceCode()` – vrací zdrojový kód algoritmu či datové struktury;
- `backgroundAction(Object, String)` – tato metoda slouží pro akci na pozadí (viz kapitola 5.4) vyvolanou tlačítky či textovými poli v panelu.

Tlačítka a textová pole umístěná v tomto panelu musí využívat rozhraní `ActionListener`, které je implementováno třídou `AlgorithmPanel`, tzn. že u tlačítek a textových polí bude volána metoda `addActionListener(this)`.

Před začátkem vykonávání některého z algoritmů či operace datové struktury musí být všechny kontrolky v panelu deaktivovány, aby uživatel nemohl spustit více akcí najednou. Po dokončení akce musí být tyto kontrolky opět aktivovány. Jejich aktivace může být vložena na konec metody `backgroundAction(Object, String)` nebo do připravené metody `done()`, která je volána po dokončení akce na pozadí.

Ukázka implementovaného modulu je uvedena v příloze A.

5.1.1 Implementace algoritmů a datových struktur

Pro implementaci algoritmů popř. datových struktur je vhodné využít abstraktní třídu `Algorithm`, která je v balíčku `kiv.pt.algorithm`. Tato třída

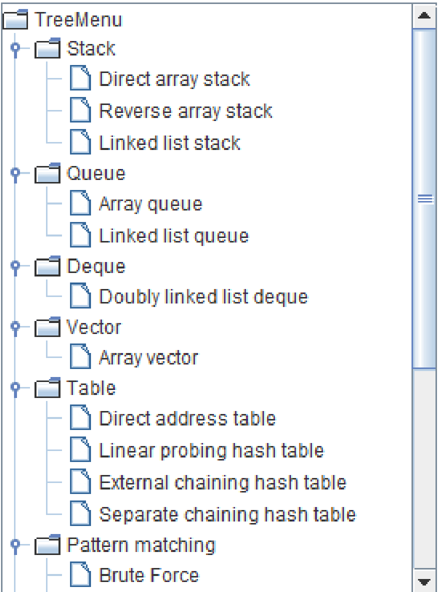
má pouze jednu abstraktní metodu `getImage()`, která vrací obrázek reprezentující práci algoritmu nebo aktuální stav datové struktury. Ke kreslení obrázků je vhodné využít metody, které tato třída poskytuje.

Při návrhu a implementaci této třídy jsem uvažoval o univerzální jednotce pro kreslení obrázků. Touto jednotkou se stal „čtverečník“, který je určen zvoleným násobkem maximální šířky nebo výšky znaků z nastaveného písma. Tento čtverečník lze získat metodou `getSquareSize()`. Výchozím písmem pro tuto třídu je tučné bezpatkové písmo o velikosti 14 pt.

Tato třída dále obsahuje `protected` proměnnou `code` pro zdrojový kód, kterou je nutné v konstruktoru třídy inicializovat. Tuto proměnnou lze používat pro zvýrazňování právě vykonávané řádky zdrojového kódu.

5.2 Menu

Pro vytvoření menu jsem využil vlastností knihovní komponenty `JTree` (viz obrázek 5.1), která slouží pro zobrazení stromové struktury.



Obrázek 5.1: Ukázka neupravené komponenty `JTree` obsahující kategorie a položky menu

Na obrázku (viz obrázek 5.1) je vidět, že `JTree` zobrazuje jednotlivé uzly stromu pod sebou, tzn. že každý řádek reprezentuje jeden uzel stromu. Tyto uzly mohou mít libovolný počet *následníků* (synů). Uzel, který má nějakého následnovníka se označuje jako *rodič* a je označen ikonou . Uzel, který nemá žádného následovníka je označen ikonou .

Důležitou vlastností této komponenty je, že umožňuje uživateli zobrazit popř. skrýt všechny následovníky zvoleného rodiče. Pokud si představíme, že rodičovský uzel je kategorií menu a jeho následovníci jsou položky menu v této kategorii, potom úpravou této komponenty získáme podokno úloh.

5.2.1 Implementace menu

V balíčku `org.swingx.menu` se nachází třída `TreeMenu`, která reprezentuje komponentu menu. Jak už název třídy napovídá, tato komponenta je odděděna od komponenty `JTree`.

Pro úpravu vizuální stránky menu bylo nutné implementování vlastních tříd pro `CellRenderer` a `CellEditor`. Třída `CellRenderer` slouží k zobrazení uzlu stromu a `CellEditor` slouží k jeho úpravě. Implementace vlastní třídy `CellEditor` je velice důležitá, protože kliknutí na uzel stromu (položku menu) je považováno za jeho úpravu.

Aby v třídách pro zobrazení a úpravu uzlu byly jednoduše odlišeny kategorie od položek menu, bylo nutné navrhnout další třídy:

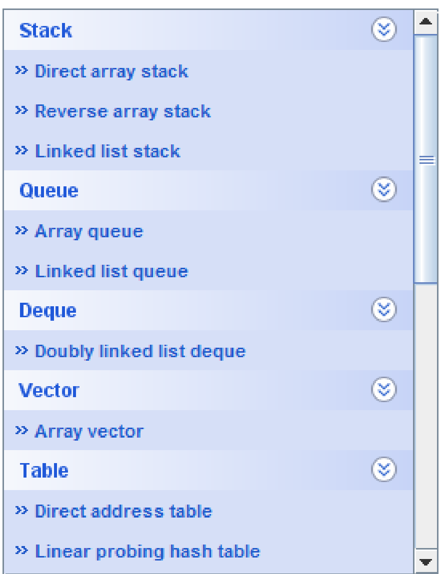
- `TreeMenuCategory` – reprezentuje kategorii menu. Každá kategorie bude obsahovat seznam položek, které jsou v této kategorii umístěny.
- `TreeMenuItem` – reprezentuje položku menu. Položka obsahuje akci `TreeMenuAction`, která bude vykonána při zvolení položky.
- `TreeMenuAction` – reprezetuje akci, která bude vykonána při zvolení položky. Každá akce umožňuje uchovávat popisek položky menu a libovolnou hodnotu, která bude společně s popiskem po zvolení položky předána posluchačům menu (viz dále).

Dále bylo nutné navrhnout grafické komponenty pro menu reprezentující kategorii (třída `TreeMenuLabel`) a položku menu (třída `TreeMenuButton`). Nyní je možné rozlišit kategorie a položky menu a díky tomu mohou být ve

vlastních třídách `TreeMenuRenderer` a `TreeMenuEditor` vytvářeny správné komponenty pro uzly menu.

Posledním krokem implementace menu bylo navrhnutí rozhraní pro posluchače menu, které bude reagovat na výběr položky. Registrovanému posluchači bude předán popisek zvolené položky a libovolný objekt, který obsahuje akce položky. Tento posluchač zajišťuje komunikaci menu s ostatními komponenty.

Ve výše uvedeném postupu byla komponenta `JTree` (viz obrázek 5.1) upravena na menu, které je velice podobné podoknu úloh (viz obrázek 5.2).



Obrázek 5.2: Ukázka implementovaného menu

5.2.2 Implementace menu do hlavního okna

Pro implementaci menu do GUI bylo nutné navrhnout způsob načtení menu (vytvoření kategorií a položek menu). Dále bylo nutné navrhnout způsob, kterým bude po výběru položky v menu vytvořen panel, který bude v GUI zobrazen.

První implementace GUI obsahovala výčet prvků (`enum`), ve kterém byly uvedeny všechny položky menu a menu bylo vytvářeno „ručně“ v programu.

Při volbě některé položky v menu byl v registrovaném posluchači menu vybrán panel pomocí příkazu `switch`, kterému byly předány správné parametry a který byl následně zobrazen. Tento způsob nebyl příliš efektivní a přidávání nových položek a kategorií bylo velice složité a nepřehledné. Proto musela být zvolena jiná metoda implementace menu do GUI.

Pro odstranění nutnosti využívat příkaz `switch` je vhodné použít technologii *Java Reflection API* (reflexe). Jde o programové rozhraní, díky kterému můžeme za běhu programu získávat informace o třídách a jejich proměnných a metodách, aniž by bylo nutné znát tyto informace v době kompilace. Stejně tak je možné vytvářet za běhu programu instance od zadaných tříd. To znamená, že položky menu budou obsahovat informaci o třídě, která má být instancována využitím reflexe. Vzhledem k tomu, že více druhů algoritmů či datových struktur využívá stejný panel, je nutné přidat do položky menu také parametr pro tento panel, kterým budou odlišeny jednotlivé algoritmy popř. datové struktury.

Jelikož je menu tvořeno kategoriemi, které obsahují jednotlivé položky, je vhodné pro uložení menu použít `XML`. Pro práci s `XML` souborem menu byla zvolena technologie *JAXB*, která nabízí snadnou konverzi `XML` dat na Java objekty a naopak. Aby bylo možné začít využívat JAXB, je nejprve nutné vytvořit `XSD` šablonu, ve které jsou popsány pravidla pro strukturu `XML` souboru (šablona pro menu je uvedena v příloze C). Z této šablony jsou nástrojem `XJC` vygenerovány třídy, které jsou pro práci s `XML` využívány.

Pro načítání menu z `XML` souboru se využívá třída `XMLMenuLoader`, která implementuje rozhraní `MenuLoader`. V této třídě jsou načteny kategorie a položky menu technologií JAXB z vygenerovaných tříd, které se nacházejí v balíčku `kiv.pt.jaxb`. Před začátkem načtení projde `XML` soubor validací, a pokud je ve špatném tvaru, je program ukončen chybou. `XML` soubor pro menu je uveden v příloze D.

Do hodnoty akce (`TreeMenuAction`) načtené položky menu je vložena třída `ContentType`, která obsahuje:

- *název položky*;
- *název třídy panelu*;
- *parametr pro panel.*

Tato třída je při volbě položky předána registrovaným posluchačům menu. Jedním z nich je hlavní okno aplikace, které užitím reflexe vytvoří správný panel, který vloží do pravé části okna. Vytvoření panelu užitím reflexe si lze prohlédnout ve zdrojovém kódu 5.1

```
try {
  Class<?> clazz = Class.forName(type.getClazz());
  Constructor<?> construct = clazz.getConstructors()[0];
  Object[] arguments = type.getArguments();

  Object instance;
  if (arguments == null) {
    instance = construct.newInstance();
  }
  else {
    instance = construct.newInstance(arguments);
  }

  AlgorithmPanel panel = (AlgorithmPanel) instance;
  panel.setTitle(type.getName());
  this.setContent(panel);
} catch (Throwable exc) {
  ErrorDialog.showErrorDialog(exc);
}
```

Zdrojový kód 5.1: Ukázka vytvoření nového panelu použitím reflexe

5.2.3 Úpravy menu

Využití `XML` souboru pro menu a načítání panelů užitím reflexe umožňuje snadné přidávání popř. upravování položek a kategorií v menu.

Ve zdrojovém kódu 5.2 je ukázka zápisu kategorie menu v `XML` souboru. Každá kategorie má dva povinné parametry:

- Povinný parametr `order` slouží k určení pořadí kategorie v menu. Tento parametr bude v dalších verzích programu nepovinný.
- Povinný parametr `name` slouží k určení jména kategorie.

```
<category order="9" name="Graph">
  <!-- Polozky v kategorii -->
</category>
```

Zdrojový kód 5.2: Ukázka kategorie pro grafy

Ve zdrojovém kódu 5.3 je ukázka zápisu položky menu v `XML` souboru. Každá položka má tři povinné parametry a jeden nepovinný:

- Povinný parametr `order` slouží k určení pořadí položky v kategorii. Tento parametr bude v dalších verzích programu nepovinný.
- Povinný parametr `name` slouží k určení jména položky.
- Povinný parametr `class` slouží k určení panelu, který bude zobrazen při volbě položky.
- Nepovinný parametr `argument` slouží k určení argumentu pro panel.

```
<category order="9" name="Graph">
  <item order="1"
        name="Dijkstra's algorithm"
        class="kiv.pt.graph.GraphPanel"
        argument="0" />

  <item order="2"
        name="Floyd-Warshall algorithm"
        class="kiv.pt.graph.GraphPanel"
        argument="0" />

  <!-- Dalsi polozky v kategorii -->
</category>
```

Zdrojový kód 5.3: Ukázka položek v kategorii pro grafy

5.3 Zdrojový kód algoritmu či dat. struktury

V balíčku `kiv.pt.sourcecode` se nachází třídy pro práci se zdrojovým kódem. Nejdůležitější z nich je třída `SourceCode`, která obsahuje zdrojový kód algoritmu či datové struktury. Při vytváření instance třídy je nutné zadat parametr, který reprezentuje název třídy, jejíž zdrojový kód má být načten ze

složky `sources`. Název třídy je ve tvaru celého jména třídy včetně názvů balíčků (např. `"kiv.pt.vector.ArrayVector"`). Dalším možným parametrem je číslo první označené řádky zdrojového kódu.

Tato třída může registrovat posluchače `SourceCodeListener`, kteří naslouchají změnám ve zdrojovém kódu pomocí metod:

- `selectedLine(int)` – volána v případě, že byla ve zdrojovém kódu označena nějaká řádka;
- `sourceCodeChanged()` – volána v případě, že byl zdrojový kód změněn;
- `processRunning(boolean)` – volána v případě, že zdrojový kód začal pracovat.

Třída `SourceCode` poskytuje základní metody:

- `getSource()` – vrací text zdrojového kódu;
- `setSource(String)` – nastaví text zdrojového kódu a změnu oznámí posluchačům;
- `selectLine(int)` – označí řádku se zadaným číslem a toto označení oznámí posluchačům;
- `selectedLine()` – vrací číslo označené řádky;
- `startRunning()` – nastaví, že zdrojový kód začal pracovat, a tuto změnu oznámí posluchačům;
- `stopRunning()` – nastaví, že zdrojový kód přestal pracovat, a tuto změnu oznámí posluchačům.

5.3.1 Zvýraznění právě vykonávané řádky

Spuštění činnosti algoritmu nebo operace s datovou strukturou je nutné v akci na pozadí (viz kapitola 5.4), protože tato akce bude v průběhu činnosti pozastavována, a pokud by nebyla puštěna na pozadí, docházelo by k zamrzání GUI. Dále je nutné uživateli znemožnit spouštět další algoritmy nebo provádět operace s datovou strukturou, aby nedocházelo ke kolizím mezi jednotlivými akcemi.

Na začátku každého algoritmu popř. operace s datovou strukturou je nutné zavolat metodu `startRunning()`, která všem registrovaným posluchačům `SourceCodeListener` oznámí, že začala činnost algoritmu popř. operace s datovou strukturou. Díky tomu lze pozastavovat a krokovat činnost zdrojového kódu.

Poté během činnosti algoritmu voláme metodu `selectLine(int)`, pomocí které označujeme právě vykonávané řádky kódu. Třída `SourceCode` oznámí označení řádky všem zaregistrovaným posluchačům. Jedním z nich je posluchač, kterého zaregistrovalo okno pro zdrojový kód `SourceCodeDialog`. Tento posluchač oznámí označení řádky textovému poli a poté spustí v třídě `SourceCodeDelay` časovač (knihovní třída `Timer`). Tento časovač zablokuje akci na pozadí pomocí `Mutexu` (`Semaphore(1)`) a začne odpočítávat prodlevu mezi dalším krokem. Po vypršení časovače je akce na pozadí opět uvolněna a může pokračovat v činnosti.

Při pozastavení činnosti zdrojového kódu se zastavuje časovač pro prodlevu. Při krokování je pouze časovač vyresetován, přičemž je pozastavená akce na pozadí uvolněna do dalšího kroku, kde opět dojde k pozastavení při označení vykonávané řádky.

Po dokončení algoritmu popř. operace s datovou strukturou je nutné zavolat metodu `stopRunning()`, která všem registrovaným posluchačům oznámí, že byla ukončena činnost algoritmu popř. operace s datovou strukturou. Nyní již není možné krokovat činnost zdrojového kódu popř. ji pozastavovat.

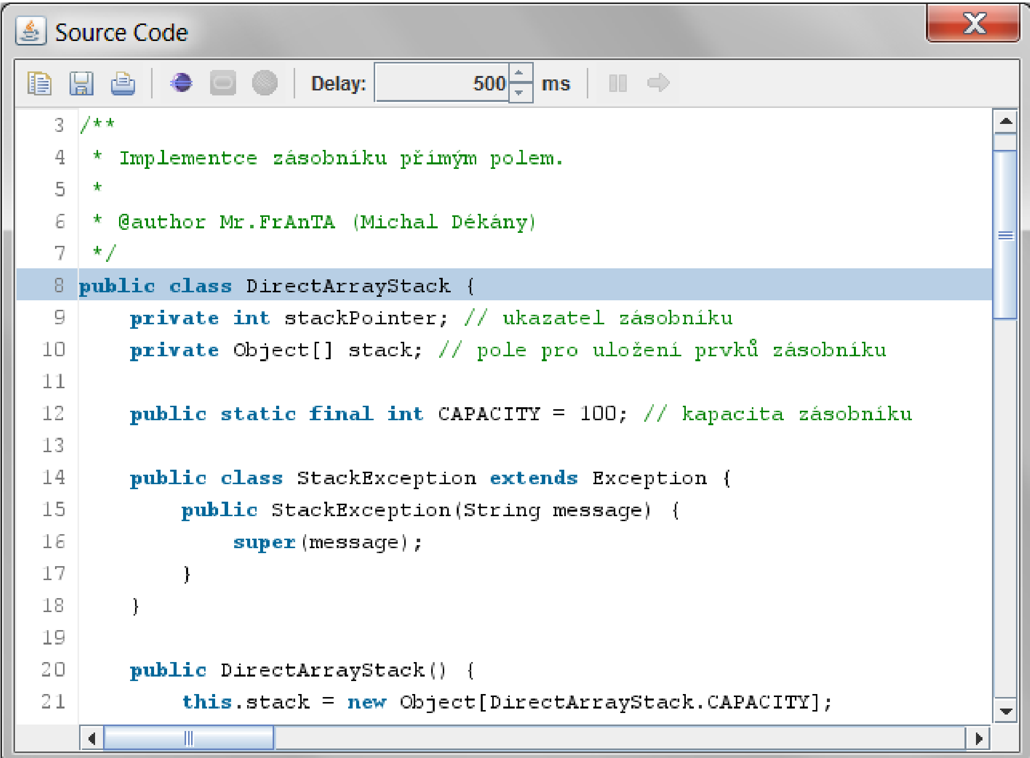
5.3.2 První verze textového pole pro zdrojový kód

V první verzi programu bylo implementováno textové pole z open-source balíčku *jEdit Syntax Package*[1] (viz obrázek 5.3), který vznikl na základě prvních verzí textového editoru jEdit[2].

Toto textové pole však mělo spoustu nevýhod. První z nich byla nutnost vlastní implementace lišty s čísly řádek. Další nevýhodou byla špatná čitelnost použitého písma, kvůli které musel být využit *Anti-aliasing* (vyhlazování hran), který čitelnost textu zlepšil. Dále textové pole neumožňovalo

[1] Dostupný na adrese: `http://syntax.jedit.org/`

[2] Open-source textový editor implementovaný v programovacím jazyce Java a dostupný pod licencí GNU General Public License.



Obrázek 5.3: Ukázka okna se zobrazeným zdrojovým kódem v textovém poli z balíčku jEdit Syntax Package

tisk zdrojového kódu, a proto muselo být součástí GUI „neviditelné“ obyčejné textové pole, které má implementovanou podporu tisku.

Hlavním problémem totoho textového pole byly chyby při rychlém označování řádek, které se objevovaly nahodile a způsobovaly pád aplikace. Rychlejší označování řádek přinášelo další problém, kterým bylo chybné překreslování textového pole. Kvůli chybnému překreslování bylo v textovém poli označeno více řádek, což by mohlo být pro uživatele velice matoucí. Chybné překreslování také způsobovalo, že se zdrojový kód stával nečitelným.

5.3.3 Druhá verze textového pole pro zdrojový kód

Kvůli problémům s první verzí textového pole, bylo nutné najít nějakou alternativu, kterou se stalo textové pole z open-source projektu *RSyntaxTextArea*[3], který je dostupný pod *BSD licencí*[4].

Použití tohoto textového pole přineslo řadu zlepšení oproti textovému poli z balíčku jEdit Syntax Package. První výhodou je speciální scrollovatelný panel `RTextScrollPane`, do kterého je textové pole vloženo a který obsahuje lištu s čísly řádek. U tohoto textového pole nebylo nutné upravovat písmo pro zdrojový kód, protože je použité písmo dobře čitelné. Dále je v tomto textovém poli plně implementován tisk textu v metodě `print()`. Textové pole také umožňuje výrazně větší možnosti ve zvýrazňování syntaxe.

Nevýhodou tohoto textového pole je jeho dlouhá doba vytváření (zavolání konstruktoru), která je způsobena načítáním písma textového pole a vytvářením stylů pro zvýrazňování syntaxe. Proto je po zobrazení GUI zobrazena ikona pro načítání (tzv. „kytička“) a uživateli je znemožněna práce s GUI. Dále je spuštěna akce na pozadí (viz kapitola 5.4), ve které je vytvořeno celé okno pro zdrojový kód. Po dokončení vytvoření je ikona pro načítání skryta a uživateli je povolena práce s GUI.

5.4 Akce na pozadí

Při implementaci grafického rozhraní je nutné dbát na dodržení několika zásad, protože GUI je první věc, která může uživatele zaujmout nebo odradit. Proto by GUI nemělo „zamrzat“ a mělo by být stále schopné reagovat na akce uživatele. Pokud je nutné, aby byl program nedostupný, je vhodné uživatele o nedostupnosti programu informovat.

Pro zdlouhavé či speciální operace je nutné využít akce v pozadí, které jsou implementovány využitím knihovní třídy `SwingWorker`. Tato třída má dva typové parametry `T` a V, a proto byly navrženy třídy `BackgroundAction` a `BackgroundActions`, které obalují třídu `SwingWorker` a zjednodušují její využití. Tyto třídy jsou v balíčku `org.swingx.actions`.

[3]Dostupný na adrese: `http://fifesoft.com/rsyntaxtextarea/`

[4]Umožňuje volné šíření licencovaného obsahu, přičemž vyžaduje pouze uvedení autora a informace o licenci, spolu s upozorněním na zřeknutí se odpovědnosti za dílo.

Abstraktní třída `BackgroundAction` obsahuje tři metody:

- `doInBackground()` – tato abstraktní metoda slouží k akci na pozadí;
- `done()` – tato metoda je volána po skončení akce na pozadí;
- `execute()` – spustí činnost akce na pozadí.

Statická třída `BackgroundActions` má pouze jednu statickou metodu `execute`, jejíž parametrem je objekt typu `BackgroundAction`. Tato metoda vytvoří instanci třídy `SwingWorker<Object, Object>`, která vykoná akci v pozadí, která je popsána v třídě `BackgroundAction`.

Akce na pozadí jsou důležité pro zvýrazňování aktuálně vykonávané řádky zdrojového kódu či datové struktury, protože mezi jednotlivými zvýrazněními je časová prodleva, během které je tato akce pozastavena. Navíc uživatel může činnost této akce pozastavit ručně. Pokud by zvýrazňování aktuáně vykonávané řádky nebylo v akci na pozadí, zamrzalo by i celé GUI.

Akce na pozadí jsou také využívány k vytvoření okna pro zdrojový kód (viz zdrojový kód 5.4) a k tiskovým úlohám.

```
BackgroundAction action = new BackgroundAction() {
    @Override
    public Object doInBackground() {
    Component glass = MainFrame.this.getGlassPane();
    glass.setVisible(true);
    MainFrame.this.dialog = new
       SourceCodeDialog(MainFrame.this);
    MainFrame.this.dialog.init();
    glass.setVisible(false);

    return MainFrame.this.dialog;
}

action.execute();
```

Zdrojový kód 5.4: Ukázka akce na pozadí, která vytváří okno pro zdrojový kód

5.5 Knihovna JEP

U nastavení tabulek s rozptýlenými položkami (viz kapitola 2.7.4) je uživateli umožněno zadat vlastní tvar rozptylové funkce (hašovací funkce). Tato rozptylovací funkce je zadána jako řetězec, jehož hodnotu je nutné vypočítat. K tomu je využita knihovna JEP (Java – Math Expression Parser), která je ve verzi 2.4.1 dostupná pod licencí *GNU General Public License*. Tato knihovna umí vypočítat hodnotu libovolné rovnice, která je zadána jako řetězec. Ve zdrojovém kódu 5.5 je ukázán princip funkčnosti této knihovny.

```
JEP hashFunction = new JEP();
hashFunction.setAllowAssignment(false); // zakazani rovnic
hashFunction.addVariable("k", 0); // klic polozky
hashFunction.addVariable("N", 0); // velikost tabulky

String hashFunctionStr = "((3 * k + 7 * N + 1) % 998) % N";
hashFunction.parseExpression(hashFunctionStr);
if (hashFunction.hasError()) {
  String error = hashFunction.getErrorInfo();
  JOptionPane.showMessageDialog(null, error,
      "Hash function error", JOptionPane.ERROR_MESSAGE);
}
else {
  hashFunction.setVarValue("N", 15) // velikost tabulky je 15
  hashFunction.setVarValue("k", 150) // vlozen prvek s
     klicem 150
  System.out.println(hashFunction.getValue()); // vypise 1
}
```

Zdrojový kód 5.5: Ukázka principu knihovny JEP

6 Závěr

V této práci jsem se zabýval návrhem a implementací výukového programu pro předmět Programovací techniky. V tomto programu je možné sledovat vlastnosti datových struktur a činnost algoritmů na zobrazených interaktivních prvcích a na zobrazeném zdrojovém kódu, ve kterém je označována právě vykonávaná řádka.

Program jsem vytvářel od prostudování vyučovaných datových struktur a algoritmů, přes důkladný návrh GUI až po implementaci a otestování výsledného programu.

Mezi možná vylepšení programu patří přidání podpory více jazyků v GUI, které je nyní dostupné pouze v anglickém jazyce. To může zhoršovat kvalitu výukového charakteru programu.

Dalším vylepšením programu je nalezení vhodnějšího systému pro označování aktuálně prováděné řádky zdrojového kódu, které je nyní realizováno pomocí pevných čísel řádek. Pokud by byl některý z ukázkových zdrojových kódů upraven, musela by být zároveň provedena změna všech čísel v příslušném modulu.

Všechny body zadání byly splněny a výukový program je připraven ke zkušebnímu nasazení v příštím semestru výuky. Přesto však práci na výukovém programu nepovažuji za dokončenou, protože v něm byly vynechány datové struktury a algoritmy, které budou do programu zařazeny v dalších verzích. Proto hodlám nadále pokračovat ve spolupráci s vedoucím práce na dalším vývoji tohoto programu.

Seznam zkratek

ADT Abstract Data Type – množina hodnot a příslušných operací, které jsou přesně specifikovány, a to nezávisle na konkrétní implementaci.

API Application Programming Interface – programové rozhraní aplikace, které si programátor vytvoří sám nebo použije již někým vytvořené.

ASCII American Standard Code for Information Interchange – znaková sada, která definuje znaky anglické abecedy a jiné znaky používané v informatice.

BMP Windows Bitmap – grafický formát pro ukládání bitmapové grafiky.

GIF Graphics Interchange Format – grafický formát pro ukládání bitmapové grafiky bezztrátovou kompresí.

JPEG Joint Photographic Experts Group – grafický formát pro ukládání bitmapové grafiky ztrátovou kompresí.

GUI Graphical user interface – umožňuje ovládání programu interaktivními grafickými prvky.

JAXB Java Architecture for XML Binding – rozhraní pro snažší práci s XML dokumenty.

JVM Java Virtual Machine – virtuální stroj, který slouží ke spuštění počítačových programů a skriptů vytvořených v jazyce Java.

PNG Portable Network Graphics – grafický formát pro ukládání bitmapové grafiky bezztrátovou kompresí.

XML Extensible Markup Language – obecný značkovací jazyk, který byl vyvinut a standardizován konsorciem W3C.

XSD XML Schema Definition – schéma popisující strukturu XML dokumentu.

Literatura

[1] CORMEN, T. H. *Introduction to algorithms.* 2nd ed. Cambridge : Addison-Wesley, 2001. xxi, 1180 s. ISBN 0-262-03293-7.

[2] HUDEC, B. *Programovací techniky.* 2. přeprac. vyd. Praha : ČVUT, 1989. 192 s.

[3] MAUTNER, P. Programovací techniky. [online], 2011 [cit. 4. května 2013]. Dostupné z: `http://www.kiv.zcu.cz/~mautner/Pt/`.

[4] MIT. 6.006 Introduction to Algorithms. [online], 2011 [cit. 13. dubna 2013]. Dostupné z: `http://courses.csail.mit.edu/6.006/spring11/rec/`.

[5] RYCHLÍK, J. *Programovací techniky.* České Budějovice : Kopp, 1994. 188 s. ISBN 80-85828-05-7.

[6] SEDGEWICK, R. – WAYNE, K. *Algorithms.* 4th ed. Boston : Addison-Wesley Professional, 2011. 976 s. ISBN 978-0-321-57351-3.

[7] SKIENA, S. S. *The algorithm design manual.* New York : Springer, 1998. 486 s. ISBN 0-387-94860-0.

[8] TAMASSIA, R. – GOODRICH, M. T. *Data structures and algorithms in Java.* 2nd ed. New York : John Wiley & Sons, 2001. xiii, 641 s. ISBN 0-471-38367-8.

A Panel pro kompresi dat

```
package kiv.pt.compression;

import java.awt.*;
import javax.swing.*;
import kiv.pt.algorithm.*;
import kiv.pt.sourcecode.SourceCode;
import org.swingx.SpringUtilities;
import org.swingx.form.*;
import org.swingx.image.ImagePanel;

/**
 * Compression Panel.
 *
 * @author Mr.FrAnTA (Michal Dekany)
 * @version 1.0
 */
public class CompressionPanel extends AlgorithmPanel {
  private Compression compression;

  private FormTextArea input;
  private JTextArea output;
  private JButton compress;
  private JButton decompress;
  private ImagePanel imagePanel;

  public CompressionPanel(int type) {
    super();

    CompressionType compressionType =
       CompressionType.fromValue(type);
    switch (compressionType) {
      case RLE:
        compression = new RLE();
        break;
      case RLE_ES:
        compression = new EscapeSequenceRLE();
        break;
      case LZW:
        compression = new LZW();
        break;

      default:
        throw new IllegalArgumentException(
            "Invalid compression type!");
    }
```

```
    setLayout(new BorderLayout());
    initTop();
    initImagePanel();
  }

  private void initTop() {
    JPanel panel = new JPanel(new BorderLayout());
    FormPanel top = new FormPanel();

    FormLabel label = top.addLabel("Input:");
    input = new FormTextArea(3, 10);
    input.setLineWrap(true);
    label.setLabelFor(input);
    top.add(input.getScrollPane());

    label = top.addLabel("Output:");
    output = new JTextArea(3, 10);
    output.setLineWrap(true);
    output.setEditable(false);
    label.setLabelFor(output);
    JScrollPane pane = new JScrollPane(output,
        JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
        JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);
    top.add(pane);

    top.makeCompactGrid();
    panel.add(top, BorderLayout.NORTH);

    JPanel buttonsPanel = new JPanel();
    JPanel buttons = new JPanel(new SpringLayout());
    compress = new JButton("Compress");
    compress.addActionListener(this);
    buttons.add(compress);
    decompress = new JButton("Decompress");
    decompress.addActionListener(this);
    buttons.add(decompress);

    SpringUtilities.makeCompactGrid(buttons,1,2,5,0,5,5);

    buttonsPanel.add(buttons);
    panel.add(buttonsPanel, BorderLayout.CENTER);
    add(panel, BorderLayout.NORTH);
  }

  private void initImagePanel() {
    imagePanel = new ImagePanel();
    compression.addAlgorithmListener(new AlgorithmListener()
       {
```

```
        @Override
        public void algorithmChanged() {
          Image image = compression.getImage();
          imagePanel.setImage(image);
        }
      });

      add(imagePanel, BorderLayout.CENTER);
    }

    @Override
    public void clear() {
      input.setText("");
      output.setText("");

      if (imagePanel != null) {
        imagePanel.setImage(null);
      }
    }

    @Override
    public SourceCode getSourceCode() {
      return compression.getSourceCode();
    }

    private void setEnabledForAll(boolean enabled) {
      input.setEnabled(enabled);
      output.setEnabled(enabled);
      compress.setEnabled(enabled);
      decompress.setEnabled(enabled);
    }

    @Override
    protected void backgroundAction(Object source, String
       label) {
      setEnabledForAll(false);

      String text = input.getText();
      output.setText("");
      if (source == compress) {
        if (!compression.isValidToCompress(text)) {
          showError("Invalid text for compression");
        }
        else {
          try {
            String compressed = compression.compress(text);
            output.setText(compressed);
          } catch (Exception exc) {
            showError(exc);
```

```
        }
      }
    }
    else {
      if (!compression.isValidToDecompress(text)) {
        showError("Invalid compressed text for
            decompression");
      }
      else {
        try {
          String compressed = compression.decompress(text);
          output.setText(compressed);
        } catch (Exception exc) {
          showError(exc);
        }
      }
    }

    setEnabledForAll(true);
  }
}
```

B Zvýrazňování vykonávané řádky

```
package kiv.pt.patternmatching;

/**
 * Algoritmus "hrube sily" (Brute Force Algorithm) pro
     vyhledavani retezcu.
 *
 * @author Mr.FrAnTA (Michal Dekany)
 */
public class BruteForce {
  public static int bruteForceMatch(String text, String
      pattern) {
    int n = text.length();
    int m = pattern.length();
    for (int i = 0; i < (n - m + 1); i++) {
      int j = 0;
      while ((j < m) && (text.charAt(i + j) ==
          pattern.charAt(j))) {
        j++;
      }

      if (j == m) {
        return i;
      }
    }

    return -1;
  }
}
```

Zdrojový kód B.1: Zdrojový kód, ve kterém se bude zvýrazňovat

```
public int patternMatch(String text, String pattern) {
  this.text = text;
  this.pattern = pattern;
  this.code.startRunning();
  this.code.selectLine(9);

  this.code.selectLine(10);
  int n = text.length();
  this.code.selectLine(11);
  int m = pattern.length();
  this.code.selectLine(12);
  for (int i = 0; i < n - m + 1; i++) {
    this.code.selectLine(13);
```

```
    int j = 0;
    this.code.selectLine(14);
    while ((j < m) && (text.charAt(i + j) !=
       pattern.charAt(j))) {
      this.code.selectLine(15);
      j++;
      this.code.selectLine(14);
    }

    this.code.selectLine(18);
    if (j == m) {
      this.code.selectLine(19);
      this.code.stopRunning();

      return i;
    }
  }

  this.code.selectLine(23);
  this.code.stopRunning();

  return -1;
}
```

Zdrojový kód B.2: Zdrojový kód, který bude zvýrazňovat

C XSD šablona pro menu

```
<?xml version="1.0" encoding="utf-8" ?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified">

  <xs:annotation>
    <xs:documentation xml:lang="en">
      Schema for Main frame menu.
      Copyright 2013 Mr.FrAnTA. All rights reserved.
    </xs:documentation>
  </xs:annotation>

  <xs:complexType name="itemType">
    <xs:attribute name="order"
                  type="xs:positiveInteger"
                  use="required" />

    <xs:attribute name="name"
                  type="xs:string"
                  use="required" />

    <xs:attribute name="class"
                  type="xs:string"
                  use="required" />

    <xs:attribute name="argument"
                  type="xs:nonNegativeInteger"
                  use="optional" />
  </xs:complexType>

  <xs:complexType name="categoryType">
    <xs:sequence>
      <xs:element name="item"
                  type="itemType"
                  minOccurs="0"
                  maxOccurs="unbounded" />
    </xs:sequence>

    <xs:attribute name="order"
                  type="xs:positiveInteger"
                  use="required" />

    <xs:attribute name="name"
                  type="xs:string"
                  use="required" />
```

```
  </xs:complexType>

  <xs:complexType name="menuType">
    <xs:sequence>
      <xs:element name="category"
                  type="categoryType"
                  minOccurs="0"
                  maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>

  <xs:element name="menu" type="menuType" />
</xs:schema>
```

D XML soubor pro menu

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>

<menu>
  <category order="1" name="Stack">
    <item order="1"
          name="Direct array stack"
          class="kiv.pt.stack.StackPanel"
          argument="0" />

    <item order="2"
          name="Reverse array stack"
          class="kiv.pt.stack.StackPanel"
          argument="1" />

    <item order="3"
          name="Linked list stack"
          class="kiv.pt.stack.StackPanel"
          argument="2" />
  </category>

  <category order="2" name="Queue">
    <item order="1"
          name="Array queue"
          class="kiv.pt.queue.QueuePanel"
          argument="0" />

    <item order="2"
          name="Linked list queue"
          class="kiv.pt.queue.QueuePanel"
          argument="1" />
  </category>

  <category order="3" name="Deque">
    <item order="1"
          name="Doubly linked list deque"
          class="kiv.pt.deque.DequePanel" />
  </category>

  <category order="4" name="Vector">
    <item order="1"
          name="Array vector"
          class="kiv.pt.vector.VectorPanel" />
  </category>
```

```
<category order="5" name="Table">
  <item order="1"
        name="Direct address table"
        class="kiv.pt.table.TablePanel"
        argument="0" />

  <item order="2"
        name="Linear probing hash table"
        class="kiv.pt.table.TablePanel"
        argument="1" />

  <item order="3"
        name="External chaining hash table"
        class="kiv.pt.table.TablePanel"
        argument="2" />

  <item order="4"
        name="Separate chaining hash table"
        class="kiv.pt.table.TablePanel"
        argument="3" />
</category>

<category order="6" name="Pattern matching">
  <item order="1"
        name="Brute Force"
        class="kiv.pt.patternmatching.PatternMatchingPanel"
        argument="0" />

  <item order="2"
        name="Boyer-Moore"
        class="kiv.pt.patternmatching.PatternMatchingPanel"
        argument="1" />

  <item order="3"
        name="Knuth-Morris-Pratt"
        class="kiv.pt.patternmatching.PatternMatchingPanel"
        argument="2" />

  <item order="4"
        name="Rabin-Karp"
        class="kiv.pt.patternmatching.PatternMatchingPanel"
        argument="3" />

  <item order="5"
        name="Rabin-Karp (Arithmetic shifts)"
        class="kiv.pt.patternmatching.PatternMatchingPanel"
        argument="4" />
</category>
```

```
  <category order="7" name="String Similarity">
    <item order="1"
        name="Longest Common Subsequence"
        class="kiv.pt.stringsimilarity.StringSimilarityPanel"
        argument="0" />

    <item order="2"
        name="Shortest Common Supersequence"
        class="kiv.pt.stringsimilarity.StringSimilarityPanel"
        argument="1" />
  </category>

  <category order="8" name="Compression algorithms">
    <item order="1"
          name="RLE (Run-length encoding)"
          class="kiv.pt.compression.CompressionPanel"
          argument="0" />

    <item order="2"
          name="RLE using escape sequences"
          class="kiv.pt.compression.CompressionPanel"
          argument="1" />

    <item order="3"
          name="LZW (Lempel-Ziv-Welch)"
          class="kiv.pt.compression.CompressionPanel"
          argument="2" />
  </category>
</menu>
```

E Uživatelská příručka

1 Požadavky programu

Pro úspěšné spuštění tohoto programu je nutné, aby byla na Vašem počítači nainstalována Java(TM) SE minimální verze 6. Ta je volně dostupná ke stažení na webových stránkách `http://java.com/en/download/`.

2 Spuštění programu

Tento program lze spustit dvojklikem nebo příkazem v příkazové řádce operačního systému:

```
java -jar "Educational software for KIV-PT.jar"
```

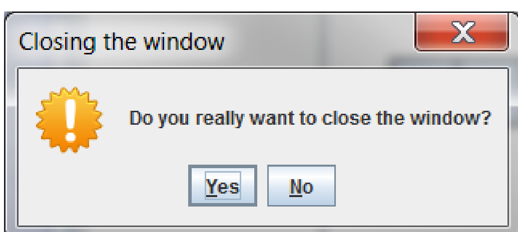
Uživatelé operačního systému *Windows* mohou využít ke spuštění programu připravený `EXE` soubor.

3 Grafické uživatelské rozhraní programu

Po spuštění programu je zobrazeno grafické uživatelské rozhraní (dále GUI), jehož velikost je vždy přizpůsobena velikosti obrazovky. V zobrazeném GUI je na pár sekund zobrazena ikona načítání (tzv. „kytička“) a GUI je neaktivní, protože se načítá písmo a styly pro textové pole, které je v okénku pro zdrojový kód (viz sekce 5). GUI programu je dostupné pouze v anglickém jazyce. Podpora více jazyků je plánována v dalších verzích.

3.1 Uzavření GUI

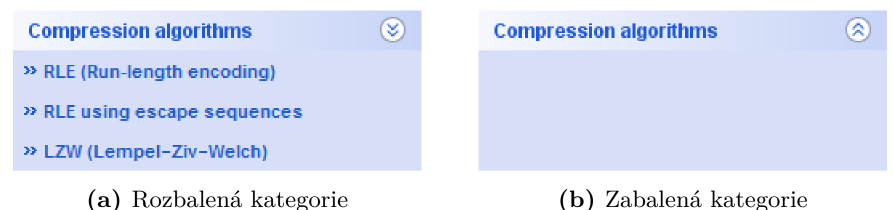
Při pokusu o uzavření hlavního okna programu se zobrazí ověřovací dialogové okno (viz obrázek 1). Po potvrzení tohoto dialogového okna (stisknutím tlačítka **OK** nebo stisknutím klávesy **ENTER**) je hlavní okno uzavřeno a program ukončen.



Obrázek 1: Dialogové okno pro ověření uzavření hlavního okna

4 Menu

Menu tvoří kategorie, které obsahují položky reprezentující jednotlivé algoritmy a datové struktury. Každou kategorii lze zabalit (skrýt položky kategorie) kliknutím na ni. Zabalené kategorii se změní ikonka z na a opětovným kliknutím lze kategorii opět rozbalit (zobrazit položky kategorie). Na obrázku 2 je znázorněna ukázka zabalená a rozbalená kategorie.



(a) Rozbalená kategorie **(b)** Zabalená kategorie

Obrázek 2: Ukázka zabalené a rozbalené kategorie pro kompresní algoritmy

Po kliknutí na některou z položek menu dojde v pravé části grafického rozhraní k zobrazení kontrolek pro práci se zvoleným algoritmem nebo datovou strukturou. Dále se zobrazí okénko pro zdrojový kód (viz sekce 5) v případě, že je skryté. Pokud je okénko již zobrazené, dojde ke změně zdrojového kódu. Při opětovném kliknutí na stejnou položku menu již ke změně nedojde.

Ve spodní části menu je umístěn ovládací panel obsahující tři tlačítka:

- **Hide** Skryje menu do levé části hlavního okna. Tlačítko se změní na . Po opětovném stisknutí se menu opět stane viditelným a dojde ke změně tlačítka na původní.
- **Collapse all** Zabalí všechny rozbalené kategorie v menu.
- **Expand all** Rozbalí všechny zabalené kategorie v menu.

5 Okénko pro zdrojový kód

Pro každý algoritmus nebo datovou strukturu je zobrazeno okénko obsahující zdrojový kód (viz obrázek 3). Tento zdrojový kód je uložen v textovém poli, které obsahuje čísla řádek a zvýrazňuje syntaxi jazyka Java. Toto textové pole také označuje aktuálně vykonávanou řádku zdrojového kódu.

Source Code

Delay: 500 ms

```
1 package kiv.pt.patternmatching;
2
3 /**
4  * Algoritmus "hrubé síly" (Brute Force Algorithm) pro vyhledávání řetězců.
5  *
6  * @author Mr.FrAnTA (Michal Dékány)
7  */
8 public class BruteForce {
9     public static int bruteForceMatch(String text, String pattern) {
10        int n = text.length();
11        int m = pattern.length();
12        for (int i = 0; i < (n - m + 1); i++) {
13            int j = 0;
14            while ((j < m) && (text.charAt(i + j) == pattern.charAt(j))) {
15                j++;
16            }
17
18            if (j == m) {
19                return i;
20            }
21        }
22
23        return -1;
24    }
25 }
```

Obrázek 3: Ukázka okénka pro zdrojový kód algoritmu hrubé síly pro vyhledávání řetězců

5.1 Nástrojová lišta

Okénko pro zdrojový kód obsahuje nástrojovou lištu, která je umístěna v horní části okénka a která obsahuje následující tlačítka:

Vloží zdrojový kód do systémové schránky (viz sekce 5.2).

Zobrazí dialogové okno, které slouží k uložení zdrojového kódu do souboru (viz sekce 5.3).

Zobrazí dialogové okno k vytištění zdrojového kódu (viz sekce 5.4).

Změní zvýraznění syntaxe zdrojového kódu na výchozí styl textového pole.

Změní zvýraznění syntaxe zdrojového kódu na styl, který je používán ve vývojovém prostředí Eclipse.

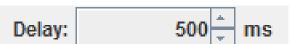
Změní zvýraznění syntaxe zdrojového kódu na styl, který je používán na webových stránkách Oracle (viz obrázek 3).

Změní zvýraznění syntaxe zdrojového kódu na styl, který je používán ve vývojovém prostředí SciTE.

Umožňuje pozastavení aktuálně vykonávané části zdrojového kódu, po kterém dojde ke změně ikony tlačítka na . Při opětovném stisknutí tlačítka dojde k obnovení činnosti a ke změně ikony na původní.

Toto tlačítko umožňuje manuální krokování vykonávané části zdrojového kódu v případě, že je jeho činnost pozastavena.

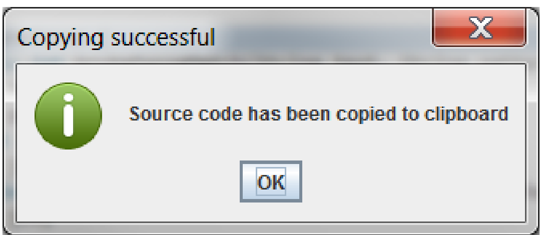
Nástrojová lišta dále obsahuje kontrolku pro nastavení prodlevy mezi kroky zdrojového kódu (viz obrázek 4). Prodleva může být 0–60 000 milisekund.



Obrázek 4: Kontrolka pro volbu prodlevy mezi kroky zdrojového kódu

5.2 Vložení zdrojového kódu do systémové schránky

Po stisknutí tlačítka je zdrojový kód vložen (zkopírován) do systémové schránky, ze které může být vložen například do vývojového prostředí. Po dokončení kopírování se zobrazí dialogové okno, které informuje o úspěšném dokončení (viz obrázek 5).

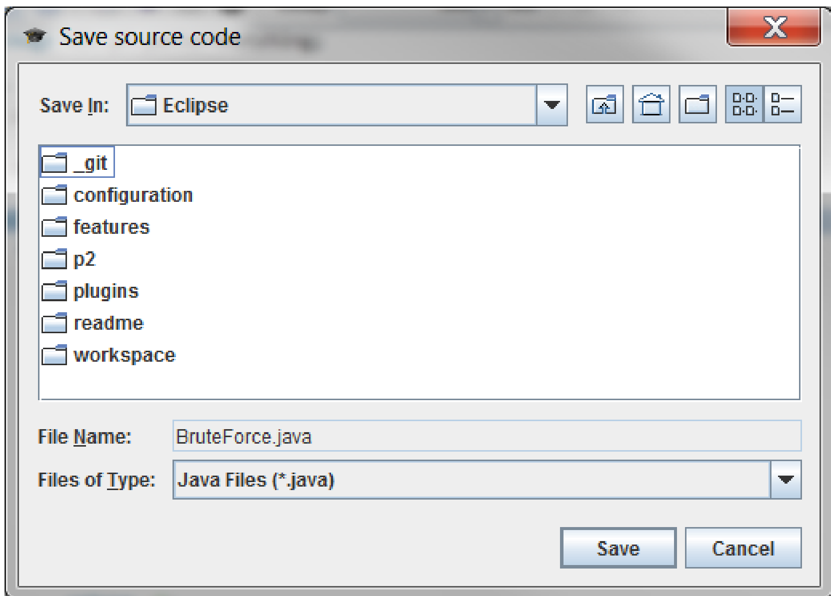


Obrázek 5: Dialogové okno, které informuje o dokončení kopírování zdrojového kódu do systémové schránky

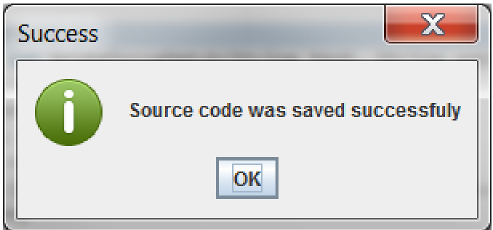
5.3 Uložení zdrojového kódu

Po stisknutí tlačítka se zobrazí dialogové okno pro uložení zdrojového kódu do souboru (viz obrázek 6). Jméno souboru nelze měnit, protože jeho název musí být stejný jako název třídy, kterou zdrojový kód obsahuje.

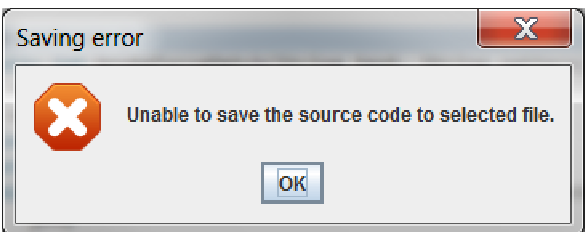
Po stisknutí tlačítka **Save** je zobrazeno dialogové okno informující o úspěšném uložení (viz obrázek 7) popř. dialogové okno informující o chybě (viz obrázek 8).



Obrázek 6: Dialogové okno pro uložení zdrojového kódu do souboru



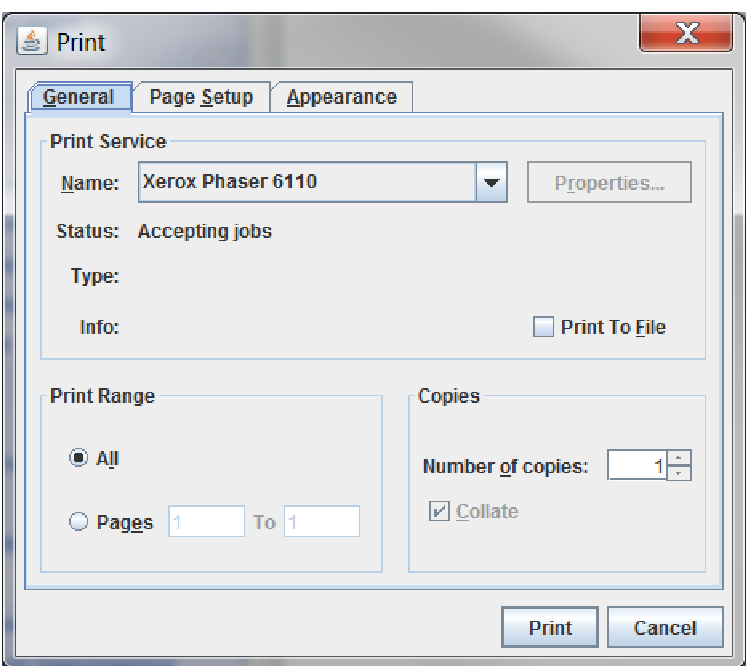
Obrázek 7: Dialogové okno informující o úspěšném uložení do souboru



Obrázek 8: Dialogové okno informující o chybě během ukládání do souboru

5.4 Tisk zdrojového kódu

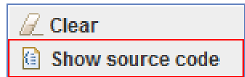
Po stisknutí tlačítka se zobrazí dialogové okno pro nastavení tisku zdrojového kódu (viz obrázek 9). Během tisku jsou zobrazována dialogová okna, která informují o stavu tisku a případných chybách během tisku.



Obrázek 9: Dialogové okno pro nastavení tisku zdrojového kódu

5.5 Uzavření a znovuotevření okénka

V případě uzavření okénka dojde pouze k jeho skrytí a prodleva mezi kroky zdrojového kódu je nastavena na 0 milisekund. K jeho znovuotevření (zobrazení) je nutné v panelu pro algoritmus (viz sekce 6) pravým tlačítkem vyvolat popup menu (viz obrázek 10), ve kterém lze zobrazovat popř. skrývat okénko se zdrojovým kódem.



Obrázek 10: Popup menu panelu pro algoritmus s červeně zvýrazněnou položkou pro zobrazení popř. skrytí okénka se zdrojovým kódem

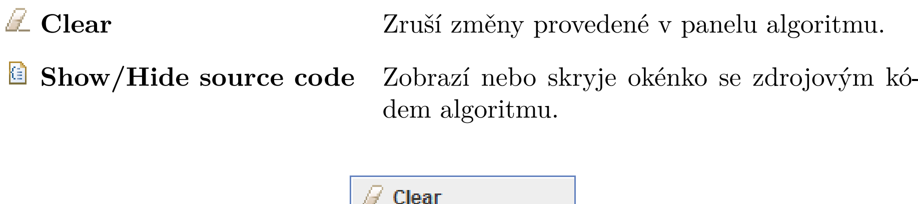
6 Panel pro algoritmus

Při zvolení určitého algoritmu nebo datové struktury v menu (viz sekce 4) je v pravé části GUI zobrazen panel pro práci s tímto algoritmem. Tento panel často obsahuje formulář(e) pro práci s daným algoritmem a panel s obrázkem algoritmu (viz sekce 6.1).

V tomto panelu může být pravým tlačítkem vyvoláno popup menu (viz obrázek 11), které má dvě tlačítka:

Clear Zruší změny provedené v panelu algoritmu.

Show/Hide source code Zobrazí nebo skryje okénko se zdrojovým kódem algoritmu.



Obrázek 11: Popup menu panelu pro algoritmus

6.1 Panel s obrázkem algoritmu

Každý algoritmus vykresluje obrázek, který znázorňuje práci algoritmu popř. jeho aktuální stav. Tento obrázek je vykreslen ve speciálním scrollovatelném panelu (viz obrázek 12), který obsahuje na pravé straně nástrojovou lištu s nástroji pro práci s tímto obrázkem.

0	150	Praha
1	100	Plzeň
2		
3		
4		
5	155	Ostrava
6		
7		

Obrázek 12: Ukázka panelu s obrázkem tabulky

Nástrojová lišta panelu obsahuje tato tlačítka:

Vloží obrázek do systémové schránky.

Zobrazí dialogové okno k uložení obrázku do souboru (viz sekce 6.1.2).

Zobrazí dialogové okno k vytištění obrázku (viz sekce 6.1.3).

Zvětší zmenšený obrázek o 10%. Maximální velikost obrázku je 100%.

Zmenší obrázek o 10%. Minimální velikost obrázku je 1%.

Tato tlačítka jsou neaktivní v případě, že algoritmus vykresluje prázdný obrázek.

6.1.1 Vložení obrázku do systémové schránky

Po stisknutí tlačítka je obrázek vložen (zkopírován) do systémové schránky, ze které může být vložen do libovolného grafického programu. Po dokončení kopírování se zobrazí dialogové okno, které informuje o úspěšném dokončení. Toto dialogové okno je podobný dialogovému oknu na obrázku 5.

6.1.2 Uložení obrázku do souboru

Po stisknutí tlačítka se zobrazí dialogové okno pro uložení obrázku do souboru, které je podobné dialogovému oknu na obrázku 6. Obrázek může být uložen do čtyř formátů: `BMP`, `JPG` popř. `JPEG`, `PNG` a `GIF`.

Pokud je uložení obrázku úspěšné, je zobrazeno dialogové okno podobné dialogovému oknu na obrázku 7. V případě chyby při ukládání obrázku je zobrazeno dialogové okno, které je podobné dialogovému oknu na obrázku 8.

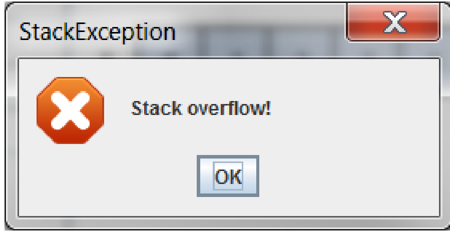
6.1.3 Tisk obrázku

Po stisknutí tlačítka se zobrazí dialogové okno pro nastavení tisku obrázku. Toto dialogové okno je stejné jako dialogové okno na obrázku 9. Během tisku jsou zobrazována dialogová okna, která informují o stavu a o chybách během tisku.

Tato funkce není vhodná pro veliké obrázky, protože v případě tisku například tabulky o velikosti 1000 na stránku A4 dojde k velikému zmenšení obrázku, který se stane nečitelným.

6.2 Chybová hlášení algoritmu

V případě, že je v algoritmu vyhozena výjimka, je zobrazeno dialogové okno informující o chybě (viz obrázek 13).

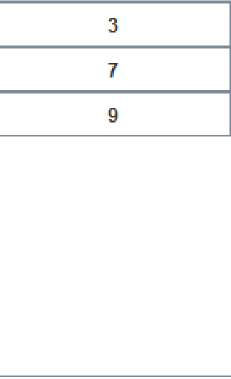


Obrázek 13: Dialogové okno informující o výjimce `StackException`

6.3 Abstraktní datové typy

Panel abstraktních datových typů (dále ADT) je rozdělen do třech částí:

1. Seznam prvků v ADT (viz obrázek 14), který se nachází v levé části panelu.
2. Formulář(e) pro práci s ADT a panel s obrázkem, který informuje o stavu ADT. Tyto komponenty jsou umístěny v pravé části panelu.
3. Tabulka operací s ADT (viz obrázek 15), která je umístěna ve spodní části panelu.



Obrázek 14: Ukázka seznamu prvků v ADT

#	Operation	Output	S
7	pop()	5	()
8	pop()	"error"	()
9	isEmpty()	true	()
10	push(9)	-	(9)
11	push(7)	-	(9, 7)
12	push(3)	-	(9, 7, 3)
13	push(5)	-	(9, 7, 3, 5)
14	pop()	5	(9, 7, 3)

Obrázek 15: Ukázka tabulky operací se zásobníkem

Mezi ADT patří:

- **Zásobník** (viz sekce 6.4);
- **Fronta** (viz sekce 6.5);
- **Obousměrná fronta** (viz sekce 6.6);
- **Vektor** (viz sekce 6.7).

6.4 Zásobník

Formulář pro práci se zásobníkem (viz obrázek 16) obsahuje pět dvojic (řádků) složených z textového pole a tlačítka. Tyto dvojice reprezentují operace se zásobníkem (metody). Formulář má aktivní pouze textové pole z první dvojice, ostatní jsou pouze pro čtení.

Obrázek 16: Formulář v horní části panelu pro práci se zásobníkem

Operace se zásobníkem je provedena po stisknutí náležitého tlačítka. Po dokončení operace je vložen záznam o této operaci do tabulky operací s ADT (viz obrázek 15).

Tlačítka pro operace se zásobníkem:

Push Toto tlačítko je možné stisknout po vyplnění prvku k vložení do zásobníku. Maximální délka prvku je 25 znaků. Po stisknutí tlačítka je tento prvek vložen na vrchol zásobníku (metoda `push`).

Pop Po stisknutí tlačítka je vyjmut prvek z vrcholu zásobníku (metoda `pop`), který je vložen do textového pole.

Top Po stisknutí tlačítka je vložen do textového pole prvek na vrcholu zásobníku bez jeho odstranění (metoda `top`).

Size Po stisknutí tlačítka je vložen do textového pole počet prvků v zásobníku (metoda `size`).

Is Empty Po stisknutí tlačítka je provedena metoda `isEmpty` a její výsledek je zobrazen v textovém poli. Pokud je do textového pole vložena hodnota `true`, změní se jeho pozadí na zelenou. V případě, že je vložena hodnota `false`, je pozadí textového pole změněno na červené.

Během operací **Push** a **Pop** je aktualizován seznam prvků v zásobníku (viz obrázek 14).

6.5 Fronta

Formulář pro práci s frontou (viz obrázek 17) obsahuje pět dvojic (řádků) složených z textového pole a tlačítka. Tyto dvojice reprezentují operace s frontou (metody). Formulář má aktivní pouze textové pole z první dvojice, ostatní jsou pouze pro čtení.

Obrázek 17: Formulář v horní části panelu pro práci s frontou

Operace s frontou je provedena po stisknutí náležitého tlačítka. Po dokončení operace je vložen záznam o této operaci do tabulky operací s ADT (viz obrázek 15).

Tlačítka pro operace s frontou:

Enqueue Toto tlačítko je možné stisknout po vyplnění prvku k vložení do fronty. Maximální délka prvku je 25 znaků. Po stisknutí tlačítka je tento prvek vložen na konec fronty (metoda `enqueue`).

Dequeue Po stisknutí tlačítka dojde k vyjmutí prvku z čela fronty (metoda `dequeue`), který je vložen do textového pole.

Front Po stisknutí tlačítka je vložen do textového pole prvek z čela fronty bez jeho odstranění (metoda `front`).

Size Po stisknutí tlačítka je vložen do textového pole počet prvků ve frontě (metoda `size`).

Is Empty Po stisknutí tlačítka je provedena metoda `isEmpty` a její výsledek je zobrazen v textovém poli. Pokud je do textového pole vložena hodnota `true`, změní se jeho pozadí na zelenou. V případě, že je vložena hodnota `false`, je pozadí textového pole změněno na červené.

Během operací **Enqueue** a **Dequeue** je aktualizován seznam prvků ve frontě (viz obrázek 14).

6.6 Obousměrná fronta

Formulář pro práci s frontou (viz obrázek 18) obsahuje osm dvojic (řádků) složených z textového pole a tlačítka. Tyto dvojice reprezentují operace s obousměrnou frontou (metody). Formulář má aktivní pouze první dvě textové pole z první dvojice, ostatní jsou pouze pro čtení.

Obrázek 18: Formulář v horní části panelu pro práci s obousměrnou frontou

Operace s obousměrnou frontou je provedena po stisknutí náležitého tlačítka. Po dokončení operace je vložen záznam o této operaci do tabulky operací s ADT (viz obrázek 15).

Tlačítka pro operace s obousměrnou frontou:

Insert First Toto tlačítko je možné stisknout po vyplnění prvku k vložení do obousměrné fronty. Maximální délka prvku je 25 znaků. Po stisknutí tlačítka je tento prvek vložen na začátek fronty (metoda `insertFirst`).

Insert Last Toto tlačítko je možné stisknout po vyplnění prvku k vložení do obousměrné fronty. Maximální délka prvku je 25 znaků. Po stisknutí tlačítka je tento prvek vložen na konec fronty (metoda `insertLast`).

Remove First Po stisknutí tlačítka je vyjmut prvek z čela obousměrné (metoda `removeFirst`), který je vložen do textového pole.

Remove Last Po stisknutí tlačítka je vyjmut prvek z konce obousměrné (metoda `removeLast`), který je vložen do textového pole.

First Po stisknutí tlačítka je vložen do textového pole prvek z čela obousměrné fronty bez jeho odstranění (metoda `first`).

Last Po stisknutí tlačítka je vložen do textového pole prvek z konce obousměrné fronty bez jeho odstranění (metoda `last`).

Size Po stisknutí tlačítka je vložen do textového pole počet prvků v obousměrné frontě (metoda `size`).

Is Empty Po stisknutí tlačítka je provedena metoda `isEmpty` a její výsledek je zobrazen v textovém poli. Pokud je do textového pole vložena hodnota `true`, změní se jeho pozadí na zelenou. V případě, že je vložena hodnota `false`, je pozadí textového pole změněno na červené.

Během operací **Insert First**, **Insert Last**, **Remove First** a **Remove Last** je aktualizován seznam prvků v obousměrné frontě (viz obrázek 14).

6.7 Vektor

Formulář pro práci s vektorem (viz obrázek 19) se liší od formulářů ostatních ADT. První čtyři řádky formuláře jsou tvořeny dvěma textovými poli s návěštími a tlačítkem. První textové pole (návěští **Element**) slouží k zadání popř. zobrazení prvku a druhé (návěští **Rank**) k zadání pozice. Dále formulář obsahuje dva řádky složené z textového pole a tlačítka.

Obrázek 19: Formulář v horní části panelu pro práci s vektorem

Každá řádka formuláře reprezentuje operaci s vektorem (metodu). Operace s vektorem je provedena po stisknutí náležitého tlačítka. Po dokončení operace je vložen záznam o této operaci do tabulky operací s ADT (viz obrázek 15).

Tlačítka pro operace s vektorem:

Insert Toto tlačítko je možné stisknout po vyplnění prvku k vložení a pozice, na kterou má být vložen. Maximální délka prvku je 25 znaků. Po stisknutí tlačítka je tento prvek vložen do vektoru na zadanou pozici (metoda `insertAtRank`).

Replace Toto tlačítko je možné stisknout po vyplnění prvku a pozice. Maximální délka prvku je 25 znaků. Po stisknutí tlačítka tento prvek nahradí prvek na zadané pozici (metoda `replaceAtRank`). Nahrazený prvek je vložen do textového pole.

Element Toto tlačítko je možné stisknout po vyplnění pozice. Po stisknutí tlačítka je vložen do textového pole prvek na zadané pozici bez jeho vyjmutí (metoda `elemAtRank`).

Remove Toto tlačítko je možné stisknout po vyplnění pozice. Po stisknutí tlačítka je vyjmut prvek na zadané pozici (metoda `removeLast`), který je vložen do textového pole.

Size Po stisknutí tlačítka je vložen do textového pole počet prvků ve vektoru (metoda `size`).

Is Empty Po stisknutí tlačítka je provedena metoda `isEmpty` a její výsledek je zobrazen v textovém poli. Pokud je do textového pole vložena hodnota `true`, změní se jeho pozadí na zelenou. V případě, že je vložena hodnota `false`, je pozadí textového pole změněno na červené.

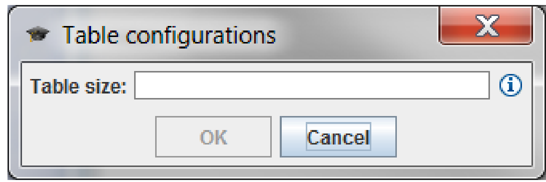
Během operací **Insert**, **Replace** a **Remove** je aktualizován seznam prvků ve vektoru (viz obrázek 14).

6.8 Tabulka

Při volbě některé z tabulek v menu je zobrazeno dialogové okno pro nastavení dané tabulky. Toto dialogové okno se liší v závislosti na zvoleném druhu tabulky.

6.8.1 Nastavení tabulky s přímým přístupem

Tabulce s přímým přístupem je nutné nastavit její velikost, jejíž maximální hodnota může být 1000 položek. Velikost tabulky se zadává v textovém poli s návěštím **Table size**.

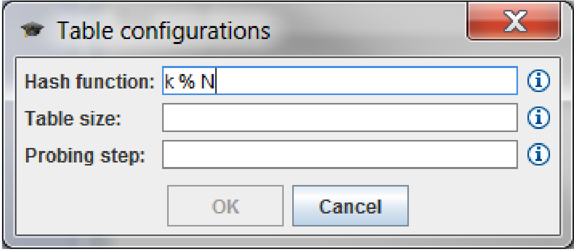


Obrázek 20: Dialogové okno pro nastavení tabulky s přímým přístupem

6.8.2 Nastavení tabulky s otevřeným rozptýlením a lin. zkoušením

Tabulce s otevřeným rozptýlením a lineárním zkoušením musíme nastavit následující vlastnosti:

- **Hašovací funkce** (textové pole s návěštím **Hash function**), která může obsahovat základní matematické operátory (včetně operátoru modulo), závorky a proměnné N pro velikost tabulky a k pro hodnotu klíče. V hašovací funkci nesmí být použito implicitní násobení.
- **Velikost tabulky** (textové pole s návěštím **Table size**), jejíž maximální hodnota může být 1000 položek.
- **Krok lineárního zkoušení** (textové pole s návěštím **Probing step**), který musí být větší než 1 a menší než velikost tabulky.



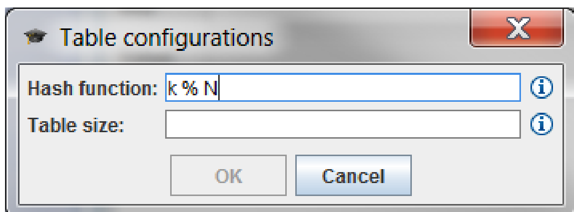
Obrázek 21: Dialogové okno pro nastavení tabulky s otevřeným rozptýlením a lineárním zkoušením

6.8.3 Nastavení tabulek s vnějším a lineárním zřetězením

Tabulkám s vnějším a lineárním zřetězením je nutné nastavit následující vlastnosti:

- **Hašovací funkce** (textové pole s návěštím **Hash function**), která může obsahovat základní matematické operátory (včetně operátoru modulo), závorky a proměnné N pro velikost tabulky a k pro hodnotu klíče. V hašovací funkci nesmí být použito implicitní násobení.

- **Velikost tabulky** (textové pole s návěštím **Table size**), jejíž maximální hodnota může být 1000 položek.



Obrázek 22: Dialogové okno pro nastavení tabulek s vnějším a lineárním zřetězením

6.8.4 Formulář pro práci s tabulkou

V horní části panelu je formulář (viz obrázek 23), který je rozdělen do tří částí. Každá část je tvořena dvěma textovými poli s návěštími a tlačítkem. První textové pole (návěští **Key** slouží k zadání číselného klíče položky tabulky (datový typ `int`). Druhé textové pole (návěští `Value`) slouží k zadání popř. zobrazení hodnoty položky tabulky. Každá část formuláře reprezentuje operaci s tabulkou (metodu).

Obrázek 23: Formulář v horní části panelu pro práci s tabulkou

Tlačítka pro operace s tabulkou:

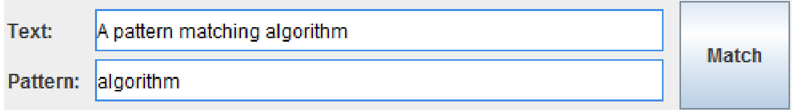
Insert Toto tlačítko je možné stisknout po vyplnění klíče a hodnoty položky. Maximální délka hodnoty položky je 25 znaků. Po stisknutí tlačítka je tato položka vložena do tabulky (metoda `insert`).

Search Toto tlačítko je možné stisknout po vyplnění klíče položky. Po stisknutí tlačítka je v tabulce vyhledána položka se zadaným klíčem (metoda `search`). Výsledek hledání (hodnota nalezené položky nebo `null`) je vložen do textového pole pro hodnotu.

Delete Toto tlačítko je možné stisknout po vyplnění klíče položky. Po stisknutí tlačítka je v tabulce vyhledána a odstraněna položka se zadaným klíčem (metoda `delete`). Hodnota nalezené a odstraněné položky popř. `null` je vložen do textového pole pro hodnotu.

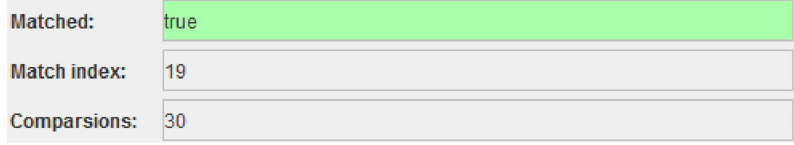
6.9 Vyhledávání řetězců

V horní části panelu je formulář (viz obrázek 24) obsahující dvě textová pole a jedno tlačítko. Prvním textové pole (s návěštím **Text**), které má výchozí hodnotu „A pattern matching algorithm", slouží k zadání textu. Druhé textové pole (s návěštím **Pattern**) slouží k zadání řetězce, který bude vyhledáván v zadaném textu. Výchozí hodnotou tohoto pole je „algorithm". Obě textová pole formuláře jsou omezena 255 znaky. Po stisknutí tlačítka **Match** se spustí vyhledávání.



Obrázek 24: Formulář v horní části panelu pro vyhledávání řetězců

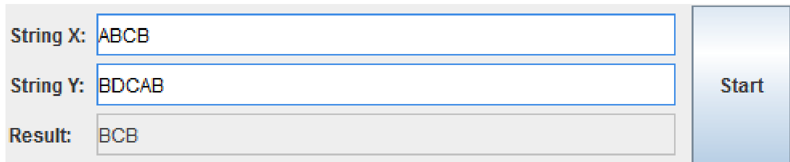
Po dokončení vyhledávání řetězce jsou zobrazeny výsledky ve formuláři (viz obrázek 25), který je ve spodní části panelu. První políčko (s návěštím **Matched**) informuje, zda byl řetězec nalezen. Druhé políčko (s návěštím **Match index**) obsahuje index, na kterém se nachází vyhledávaný řetězec. Poslední políčko (s návěštím **Comparsions**) obsahuje počet provedených porovnání mezi znaky řetězců.



Obrázek 25: Formulář ve spodní části panelu pro výsledky vyhledávání

6.10 Hledání podobnosti řetězců

V horní části panelu je formulář (viz obrázek 26) obsahující tři textová pole a jedno tlačítko. První textové pole (s návěštím **String X**) slouží k zadání řetězce X. Druhé textové pole (s návěštím **String Y**) slouží k zadání řetězce Y. Obě textová pole formuláře jsou omezena 255 znaky. Po stisknutí tlačítka **Start** se spustí hledání. Po dokončení hledání je výsledek zobrazen v posledním textovém poli (s návěštím **Result**).



Obrázek 26: Formulář v horní části panelu pro hledání podobnosti řetězců

6.11 Komprese dat

V horní části panelu je formulář (viz obrázek 27) obsahující dvě textová pole a dvě tlačítka. První textové pole (s návěštím **Input**) slouží k zadání vstupního textu pro kompresi popř. dekompresi. Při stisknutí tlačítka je vstupní text zkontrolován, zda je ve správném tvaru, a poté dojde ke kompresi (tlačítko **Compress**) nebo dekompresi (tlačítko **Decompress**) vstupního textu. Výsledek komprese popř. dekomprese je zobrazen v druhém textovém poli (s návěštím **Output**).

Obrázek 27: Formulář v horní části panelu pro kompresi dat

7 Servis a řešení chyb

Pokud se v programu vyskytne jakákoliv chyba, můžete kontaktovat autora na adrese `michal.dekany@seznam.cz`.