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

Bakalářská práce

Hashovací funkce pro textové účely

Plzeň, 2013 Radek Petruška

Prohlášení

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

V Plzni dne _____ _____

podpis

Abstrakt

Cílem bakalářské práce je seznámení se základními metodami tvorby hashovacích funkcí pro textové účely. Provedení analýzy jak hashování textových řetězců je ovlivněno vlastnostmi hashovacích funkcí.

Výsledkem práce je návrh a realizace aplikace, která umožní získat výsledky vhodné pro experimentální porovnání vlastností vybraných hashovacích funkcí. Výsledky jsou porovnávány pomocí tabulkového editoru Excel. Získané výsledky vybraných hashovacích funkcí jsou vyhodnoceny vůči hashovací funkci určené vedoucím bakalářské práce.

Abstract

The main goal of this work is to familiarize with the basic methods of constructing hash functions for text purposes. Conducting analysis of how hashing of text string is affected by properties of hash functions.

The result of this work is to design and implement the application for gathering results necessary for comparing properties of chosen hash functions. Results from application are compared using table editor Excel. Gathered results of chosen hash functions are compared to hash function determined by thesis supervisor.

Obsah

1 Úvod

Tématem bakalářské práce je porovnání vlastností vybraných hashovacích funkcí pro textové účely. V tomto případě hashovací funkce slouží pro výpočet indexu do hashovací tabulky, nejedná se o kryptografické hashovací funkce. Jako vstupní data slouží slovníky chemických sloučenin. K tomuto účelu je potřeba vytvořit aplikaci, díky které bude možné získat výsledky vhodné pro porovnání. Práce volně navazuje na bakalářskou práci pana Citriaka [1], která se zabývá použitím hashovacích algoritmů vůči cizojazyčným slovníkům.

Teoretická část bakalářské práce je zaměřena na definice základních pojmů, konstrukci hashovacích funkcí pro textové účely a analýzu základních vlastností textových řetězců. Dále obsahuje informace o použitých hashovacích algoritmech. Poslední část, teoretické části, je zaměřena na vlastnosti hashovacích algoritmů, podle kterých je lze porovnávat.

Realizační část bakalářské práce obsahuje popis datové struktury hashovací tabulky, jednotlivé implementační kroky a algoritmy potřebné pro vytvoření aplikace. Zabývá se také způsobem, kterým bude probíhat porovnání a vyhodnocení získaných dat.

Závěr této práce se bude věnovat porovnání a vyhodnocení získaných výsledků vybraných hashovacích funkcí vůči hashovací funkci určené vedoucím bakalářské práce.

2 Teoretická část

2.1 Hashovací funkce

2.1.1 Obecně

Hashovací funkce obecně slouží k mapování bitového vektoru na jiný bitový vektor, který je obvykle kratší než původní bitový vektor a má pevně danou délku. Obecně jsou na hashovací funkce kladeny dva základní požadavky. První požadavek spočívá v co možná nejrychlejším času výpočtu hashovací hodnoty a druhý požadavek se zaměřuje na unikátnost vypočítané hashovací hodnoty.

Hashovací funkce mají tři základní použití [2]:

1. Fast table lookup - rychlé dohledávání dat.
2. Message digest - porovnání vektorů.
3. Encryption - kryptografie.

Každé ze základních použití má specifické vlastnosti. Hashovací funkce pro Fast table lookup slouží k výpočtu indexu do hashovací tabulky. Aby bylo ukládání do hashovací tabulky co nejrychlejší, je kladen důraz především na rychlost použitých algoritmů.

Funkce Message digest slouží pro porovnání dlouhých bitových vektorů, zvláště tam kde je rychlejší porovnávat výsledky funkce než jednotlivé vektory bit po bitu. Pokud je výsledná hodnota funkce stejná pro oba vektory, je velká pravděpodobnost, že jsou totožné.

Kryptografické hashovací funkce slouží k transformaci vstupního vektoru na vektor dat, který je většinou delší než původní vstupní vektor. Cílem kryptografických hashovacích funkcí je vytvořit vektor takový, aby nebylo možné přečíst původní vstupní vektor. V první řadě kryptografické funkce kladou důraz na kryptografické vlastnosti. Rychlost samotného výpočtu pro ně není prvořadá. Mezi hlavní kryptografické vlastnosti patří především unikátnost každé vypočítané hashovací hodnoty. Používají se například pro ochranu hesel zaznamenaných v databázích.

Z výše uvedených požadavků na jednotlivé typy hashovacích funkcí lze sestavit seznam obecných vlastností pro hashovací funkce:

1. různé vstupní vektory generují výstupní vektory stejné délky
2. malá změna vstupního vektoru vyvolá velkou změnu výstupního vektoru
3. z výstupního vektoru je prakticky nemožné získat vstupní vektor
4. pro každý vstupní vektor existuje unikátní hashovací hodnota (eliminace kolizí)

Výše uvedené vlastností může splňovat pouze tzv. „perfektní hashovací funkce“. U normálních hashovacích funkcí se vyskytuje především problém kolize. Kolize znamená, že hashovací funkce pro různé vstupní vektory vypočítala stejnou hashovací hodnotu. Případy kolize se pro dané hashovací funkce řeší podle účelu použití hashovací funkce. Pokud lze vyjádřit libovolný výsledek hashovací funkce matematickým zápisem, tak platí:

$$\forall x \in M; \exists y \in N: y = h(x) \tag{2.1}$$

Kde M je množina mapovaných vektorů, N je množina výsledných vektorů a $h(x)$ je hashovací funkce. Potom platí, že množina N je konečnou množinou. Výjimkou jsou speciální hashovací funkce (kryptografické funkce). Zde nastává problém, že množina M je obecně větší než konečná množina N a kvůli tomu nemůže být dodržena 4. vlastnost „Pro každý vstupní vektor existuje unikátní hashovací hodnota“. Matematicky lze důsledek tohoto problému zapsat:

$$\exists x_1, \exists x_2 \in M, x_1 \neq x_2: h(x_1) = h(x_2) \tag{2.2}$$

Z toho vyplývá možný vznik kolizí. Pokud je množina M menší nebo stejně velká jako množina N, je možně najít hashovací funkci splňující podmínku:

$$\forall x_1, x_2 \in M: x_1 \neq x_2 <=> h(x_1) \neq h(x_2) \tag{2.3}$$

Hashovací funkce, která splňuje takovouto podmínku, se nazývá „perfektní hashovací funkce“. V praxi se takováto hashovací funkce hledá velmi špatně, proto se používají hashovací funkce, které se svými vlastnostmi blíží k „perfektní hashovací funkci“ [1 str. 4].

2.1.2 Metoda konstrukce - obecně

Obecně všechny hashovací funkce pracují podle stejného postupu. Tento postup se může vyjádřit pomocí několik kroků [2]:

1. Inicializace hashovací funkce. Načtení vstupního vektoru a inicializace konstantních hodnot.
2. Rozdělení vstupního vektoru na stejné části.
3. Přiřazení číselných hodnot jednotlivým částem.
4. Vlastní výpočet hashovací hodnoty.

V případě textového řetězce je nejjednodušší postup rozdělit vstupní řetězec na jednotlivé znaky a každému znaku přiřadit číselnou hodnotu z ASCII tabulky. Tento postup zpracování řetězce je využit při implementaci vlastní aplikace.

2.1.3 Metoda konstrukce - textové účely

Konstrukce hashovací funkce pro textové účely může vycházet z analýzy jazyka, pro který je vytvářena (viz podkapitola 2.3 Analýza textových řetězců). Nebo můžou k řetězcům přistupovat obecně.

Metoda přístupu k textovému řetězci především ovlivní způsob rozdělení vstupního řetězce. Například může vstupní řetězec rozdělit na jednotlivé části podle daného jazyka. Příkladem pro český jazyk může být slovo *velryba*, které obsahuje kořen slova **ryb**, předponu **vel** a koncovku **a**. Po takovémto rozdělení může hashovací funkce přidělit jednotlivým částem slova priority, které ovlivní výstupní hashovací hodnotu. Takto vytvořené hashovací funkce jsou použitelné pouze pro specifický jazyk a specifický případ použití.

Obecný přístup k textovému řetězci znamená, že všechny vstupní řetězce jsou rozdělovány stejně bez ohledu na jazyk. Příkladem může být opět slovo *velryba*, kde každé písmeno má pro výpočet hashovací hodnoty stejnou váhu. Při obecném přístupu můžeme ovlivnit směr načtení vstupního vektoru a pozorovat chování hashovacích funkcí při přímém nebo obráceném (reverzním) načítáním (*velryba* / *abyrlev*).

Obecný přístup k vstupnímu vektoru je využit ve vlastní implementaci aplikace s možností určit směr načítání.

2.2 Hashovací tabulka

Hashovací tabulka je obecná datová struktura, která slouží k ukládání dat. Data jsou do hashovací tabulky ukládána pomocí dvojice klíč-hodnota, kde klíč je vypočítáván pomocí některého hashovacího algoritmu. Díky tomu hashovací tabulka kombinuje výhody vyhledávání pomocí indexu se složitostí $O(1)$ a procházení seznamu se složitostí $O(n)$, kde n je počet prvků v seznamu [3]. Kvůli kombinaci těchto vlastností se potom složitost hledání v hashovací tabulce blíží k $O(1)$, protože index v hashovací tabulce odpovídá vypočítanému klíči hledané hodnoty. Za předpokladu úplného selhání hashovací funkce může tabulka zdegenerovat do seznamu a následné hledání bude prováděno se složitostí $O(n)$.

Při vkládání dat do hashovací tabulky může nastat kolize (viz podkapitola 2.1). Vzniklé kolize se řeší pomocí metod zřetězeného rozptylování nebo otevřeného rozptylování.

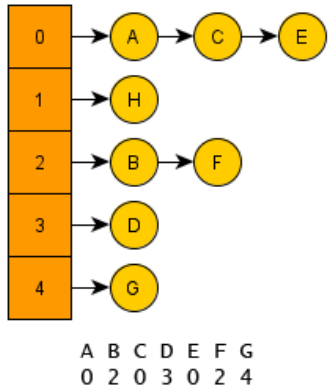
2.2.1 Zřetězené rozptylování

Zřetězené rozptylování je metoda jak se vypořádat s kolizemi uvnitř hashovací tabulky. Prvky se stejnou hashovací hodnotou jsou poté ukládány do pomocných datových struktur. Tyto struktury mohou být tvořeny spojovými seznamy, dynamickými poli nebo stromy. Pravděpodobně nejjednodušší postup je pomocí spojových seznamů. Nastane-li kolize (daný index už je obsazen), nově vkládaný prvek se přidá nakonec

spojového seznamu pro daný index (viz Obrázek 2.2.1-1: Zřetězené hashování, zdroj obrázku [3]).

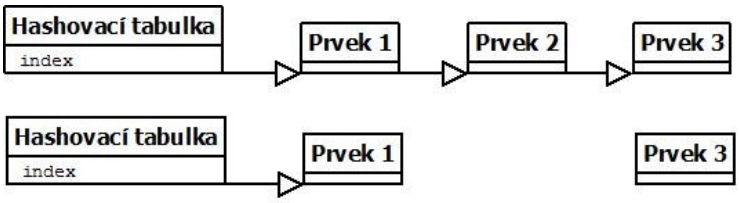
Nevýhoda této metody ukládání hodnot spočívá v možně degeneraci hashovací tabulky do několika spojových seznamů při velkému výskytu kolizí nebo při velkém zaplnění tabulky. Potom celé hledání dat spočívá v sekvenčním prohledávání spojových seznamů. Stupeň degenerace hashovací tabulky je závislý na velikosti tabulky a použitém algoritmu pro výpočet hashovacích hodnot.

Tato metoda ukládání spolu se spojovým seznamem je využita při implementaci vlastní aplikace.



Obrázek 2.2.1-1: Zřetězené hashování

Odebírání prvků při použití metody zřetězeného rozptylování je triviální záležitost. O úrovni obtížnosti odebraní prvku z hashovací tabulky rozhoduje pouze složitost použité pomocné datové struktury. V případě použití spojových seznamů se musí brát zřetel na to, aby jednotlivé seznamy nebyly přerušeny a nedošlo ke ztrátě ostatních prvků, viz Obrázek 2.2.1-2: Ztráta prvku.



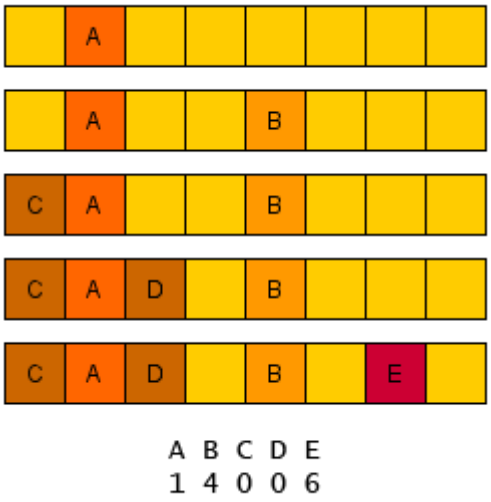
Obrázek 2.2.1-2: Ztráta prvku

Z uvedeného obrázku je patrné, že při odebrání druhého prvku došlo i ke ztrátě třetího prvku, protože už nebylo opraveno napojení mezi prvky. Vlastní způsob odebírání prvků z hashovací tabulky je plně závislý na použité pomocné datové struktuře. V případě stromu by bylo důležité zjišťovat, jestli odebíraný prvek je list nebo uzel stromu a na základě těchto faktů strom přepočítávat.

2.2.2 Otevřené rozptylování

Tato metoda ukládá nové prvky přímo do pole, které implementuje hashovací tabulku. Základní myšlenka této metody spočívá v tom, že je předem znám konečný počet všech prvků pro vložení do tabulky. Při nesprávném použití strategie otevřeného rozptylování může nastat situace, že hashovací tabulka bude úplně zaplněná a pro nový prvek v ní nebude fyzicky místo. Tato situace teoreticky nemůže při použití zřetězeného rozptylování nastat. Pokud při použití strategie otevřeného rozptylování nastane kolize (vypočítaný index je obsazen), je řešena pomocí některé ze dvou strategií. Těmito strategiemi jsou Linear Probing nebo Double Hashing [3].

Při použití strategie Linear Probing se nejprve vypočte index, kam má být hodnota uložena. Je-li index obsazen, posune se hodnota indexu o jedno místo dál a uložení se provede znovu. Takto se postupuje, dokud se nepovede hodnotu uložit do tabulky. Nevýhoda této strategie spočívá ve vytvoření shluků (tzv. clustering). Tyto shluky se poté musejí sekvenčně procházet. Velká nevýhoda shluků spočívá v tom, že jeden shluk může obsahovat i více prvků různých klíčů (viz Obrázek 2.2.2-1: Linear Probing, zdroj obrázku [3]). Z tohoto vyplývá, že při použití strategie Linear Probing využíváme pouze jeden hashovací algoritmus.



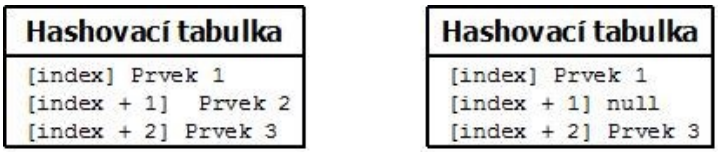
Obrázek 2.2.2-1: Linear Probing

Odebrání prvku při použití metody otevřeného adresování není triviální záležitost. Odebírání prvku z hashovací tabulky je možno provést dvěma způsoby.

Prvním způsobem je po každém odebrání celou hashovací tabulku přepočítat. Tento způsob je velmi nepraktický a především výpočetně velmi náročný pro tabulky větších rozsahů.

Druhým používaným způsobem je během mazání prvku daný prvek nahradit speciálním objektem (tzv. Sentinel). Tento objekt se vkládá na prázdné místo z důvodu, aby nedošlo k rozpadu shluků. Při vkládání nového prvku je tento objekt ignorován a index se chová jako prázdný.

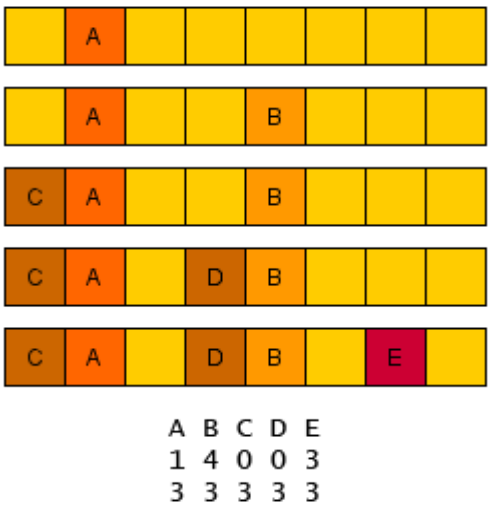
Když dojde v hashovací tabulce k rozpadu shluků, nelze v ní spolehlivě vyhledávat, protože při vyhledávání může a nemusí procházet shluk prvků. Za předpokladu hledání konkrétního prvku, který byl již uložen v hashovací tabulce, se nejdříve vypočte index, kde má daný prvek ležet a potom se na toto místo v hashovací tabulce přistoupí. Na vypočteném indexu musí ležet buď hledaný prvek, nebo shluk, který ho obsahuje. Když je prohledávaný shluk přerušený, tak se prohledávání zastaví dřív, než se projde celý shluk. Pokud hledaný prvek leží v části shluku, která se neprojde kvůli rozpadu, tak tento prvek nelze nalézt do té doby, než dojde k opětovnému přepočítání hashovací tabulky. Pro lepší představu je zde uvedena ilustrace viz Obrázek 2.2.2-2: Chyba shluku.



Obrázek 2.2.2-2: Chyba shluku

Z obrázku lze vidět, že hashovací algoritmus vypočítal pro přidání prvků 1-3 stejný index. Díky tomu prvky 1-3 vytvořily shluk. Při odebrání prvku 2 dojde k rozpadu shluku. Následně je prvek 3 dostupný až po přepočítání hashovací tabulky.

Strategie Double Hashing funguje na stejném principu jako Linear Probing s tím rozdílem, že pokud je daný index obsazen, použije druhou funkci na výpočet posunu. Za předpokladu, že je dané místo obsazené, dojde opět k posunu pomocí druhé funkce. Zásadní rozdíl mezi strategiemi Linear Probing a Double Hashing spočívá v tom, že Double Hashing potřebuje pro přidávání nových prvků dva hashovací algoritmy. Díky tomu tato strategie dokáže eliminovat shluky (viz Obrázek 2.2.2-3: Double Hashing, zdroj obrázku [3]).



Obrázek 2.2.2-3: Double Hashing

2.3 Analýza textových řetězců

Geometrické (matematické) vstupní vektory (data), lze vyjádřit jako vektor $< x, y, z >$. Počet položek vstupního vektoru je konečné číslo dané podstatou objektu, který zastupuje. Problémem je, že jednotlivé prvky vektoru můžou nabývat hodnot na intervalu $(-\infty, +\infty)$. Naopak pro textová data platí, že délka vstupního vektoru může být nekonečná, ale hodnota jednotlivých prvků vektoru je limitována abecedou [4].

Analýza textových řetězců ovlivňuje především rozdělení a načtení vstupního vektoru. Pomocí analýzy textového řetězce se můžou zlepšit vlastnosti specifické hashovací funkce pro specifický případ. Za předpokladu dobrého rozdělení vstupního řetězce můžou být jednotlivé části vstupního řetězce zpracovávány v závislosti na důležitosti pro daný problém. Příkladem může být přidělování priorit k jednotlivým částem vstupního řetězce při tvorbě hashovací hodnoty. Priority následně určují, jak která část vstupního řetězce ovlivní výslednou hashovací hodnotu. V praxi by to ale znamenalo malou možnost využití takovéto hashovací funkce.

2.3.1 Analýza podle oboru

Analýza podle oboru vychází z pochopení vstupních dat. Z pochopení toho, které části vstupního řetězce jsou důležité a rozhodující pro správné zařazení v rámci daného problému. Takto vyrobená hashovací funkce by měla pouze jediné specifické použití a to v rámci jednoho jazyka, protože vlastní jazyk silně ovlivňuje názvosloví každého oboru.

Příkladem může být obor chemie, protože o počtu molekul v celkové sloučenině rozhoduje předpona a přípona slova a kořen slova tvoří použitý prvek. Příklad na jednoduchých oxidech v rámci českého jazyka potom může vypadat jako:

- oxid uhličitý CO_2
- oxid uhelnatý CO

Počet molekul jednotlivých prvků je zde vyjádřen názvem skupiny, do které sloučeniny spadají. Jméno skupiny je **oxid**. A příponami použitého prvku v uvedeném konkrétním případě jsou přípony **-ičitý** a **-natý**. Zbytek použitého prvku potom ukazuje, že se jedná o prvek **uhlíku**.

V rámci anglického jazyka budou potom názvy sloučenin vypadat takto:

- Carbon dioxide CO_2
- Carbon monoxide CO

O počtu molekul zde rozhoduje předpona a nikoliv přípona jako v českém jazyce. Předponami jsou potom **di-** a **mono-**. Dalším rozdílem je fakt, že předpony nejsou připojeny k použitému prvku, ale k názvu skupiny dané sloučeniny. Jméno použitého prvku stojí v názvu sloučeniny samostatně.

Tento jednoduchý příklad je ukázkou toho, jak názvosloví jednotlivých oborů je závislé na použitém jazyce.

2.3.2 Analýza podle jazyka

Jazyky dělíme podle genetické (genealogické) klasifikace jazyka do takzvaných rodin. Celý seznam obsahuje 11 rodin, pro názornost jsou zde některé uvedeny [5].

1. indoevropské - většina evropských jazyků
2. uralské - maďarština, finština
3. tibetočínské - čínština, tibetština
4. austroasijské - malajština, indonéština
5. africké - súdánské, bantuské
6. indiánské
7. kavkazské - gruzínština

V rámci každé rodiny jsou jazyky rozděleny do skupin. Příbuzné jazyky, které jsou ve stejné jazykové rodině, obsahují společné prvky nebo podobně znějící slova se stejným nebo podobným významem. Příkladem může být slovo matka. Uvedená ukázka byla převzata z učebnice pro střední školy [6].

- matka - čeština
- matka - polsky
- mať - rusky
- Mutter - německy
- mother - anglicky
- mater - latinsky

Dalším krokem dělení je dělení jazyků podle skupin. Každá rodina jazyků má několik skupin. Pro indoevropskou rodinu:

1. západoslovanské – čeština, slovenština
2. východoslovanské – ruština, běloruština
3. jihoslovanské – slovinština, srbština, chorvatština

Celkovým příkladem porovnání jazyků potom může být porovnání češtiny a arabštiny, které spadají do stejné rodiny indoevropských jazyků. Okamžitě viditelný rozdíl spočívá v používání dvou naprosto rozdílných abeced. Další rozdíly mezi jazyky jsou tvořeny typologickými vlastnostmi.

Jazyky můžeme dále dělit podle typologických vlastností. Tyto vlastnosti jazyka popisují gramatiku (pravopis), syntax (vztah mezi slovy, skloňování) a fonologii (zvukovou stránku jazyka). Přehled některých základních typů [5]:

1. flexivní typ - slovanské jazyky - ohýbání koncovek
2. izolační typ - francouzština - ohýbání bez koncovek

Právě tyto vlastnosti jazyka jsou klíčové pro správnou analýzu textových řetězců. Především gramatiky a syntaxe jazyka. Fonologická stránka jazyka v případě textových řetězců není až tak podstatná.

2.4 Použité hashovací funkce

Pořadí hashovacích funkcí 2.4.1 až 2.4.9 bylo převzato z dokumentu dodaného vedoucím bakalářské práce [1 stránky 11-16].

2.4.1 Aditivní funkce

Aditivní funkce je základní hashovací funkcí, kde je hodnota každého znaku přičtena k celkové sumě. Tato suma je v posledním kroku modulárně dělena vybraným prvočíslem, které určuje velikost hashovací tabulky. Protože se jedná o základní hashovací algoritmus jsou jeho výsledky při použití na rozsáhlých datech špatné.

Poznámka:

$h(x)$	- hashovací hodnota
x_i	- znak vstupního vektoru
N	- velikost vstupního vektoru
p	- velikost hashovací tabulky

Obecná ukázka vzorce pro aditivní funkci:

$$h(x) = \left(\sum_{i=1}^{N} x_i \right) mod\ p \tag{2.4}$$

Za předpokladu použití zřetězeného rozptylování může být prvočíslo nahrazeno i libovolným číslem určujícím velikost hashovací tabulky. Ukázka použité implementace byla převzata z [1 str. 11].

```
ulong hash = (ulong)key.Length;
for (int i = 0; i < key.Length; ++i)
{
  hash += key[i];
}

return (hash % this.primeNumber);
```

2.4.2 XOR funkce

Další základní hashovací funkcí je XOR funkce. Rozdíl mezi XOR funkcí a aditivní hashovací funkcí spočívá pouze v tom, že sčítání v aditivní funkci je nahrazeno bitovým součtem. Protože zde bylo sčítání aditivní funkce nahrazeno pouze bitovým součtem, jsou výsledky na rozsáhlých datech velmi špatné. Ukázka použité implementace byla převzata z [1 str. 12].

```
ulong hash = 0;
for (int i = 0; i < key.Length; ++i)
{
  hash ^= key[i];
}

return (hash % this.primeNumber);
```

2.4.3 Rotační funkce

Poslední ze základních hashovacích funkcí je rotační hashovací funkce, která vychází z XOR funkce. Rozdíl mezi XOR a rotační funkcí je následující. Rotační funkce svůj bitový součet rozšířila ještě o bitový posun oběma směry. Díky tomu získává lepší rozložení než XOR hashovací funkce. Ukázka použité implementace byla převzata z [1 str. 12].

```
ulong hash = (ulong)key.Length;
for (int i = 0; i < key.Length; ++i)
{
 hash = (hash << 4) ^ (hash >> 28) ^ key[i];
}

return (hash % this.primeNumber);
```

2.4.4 Skala Václav, Hrádek Jan, Kuchař Martin funkce

Bakalářská práce je zaměřena na tuto specifickou hashovací funkci. Dále označovaná jako Shash. Závěrečné porovnání výsledků ostatních hashovacích funkcí je provedeno proti ní. Detailní popis hashovací funkce je popsán v dokumentu spolu se všemi zde uváděnými vzorci [7].

Poznámka:

$h(x)$ - hashovací hodnota

x_i - znak vstupního vektoru

L_x, L_{max} - délka vstupního vektoru, délka nejdelšího vstupního vektoru

N - počet vstupních vektorů ze vstupních dat bez duplicit
HS - velikost hashovací tabulky
q - parametr hashovací funkce
f - načítací faktor (doporučená hodnota 0.5 [7])

Funkce principiálně vychází z Aditivní funkce. Vzorec pro aditivní funkci (2.4). Vzorec je následné rozšířen o několik proměnných. Hashovací hodnota funkce Shash se potom vypočítává podle následujícího vzorce:

$$h(x) = \left\lfloor c \sum_{i=0}^{L_x-1} x_i \, q^i \right\rfloor \, mod \, HS \tag{2.5}$$

Základní myšlenka Shash vychází z geometrické řady. Konvergenci řady zaručuje parametr q. Pro konvergenci geometrické řady vyplývá z teorie, že kvocient geometrické řady musí být na intervalu $(0; 1)$. Proto i parametr q musí ležet na stejném intervalu $q \in (0; 1)$. Hodnota parametru q se získává pomocí simulace.

Ze vzorce Shash funkce je vidět, že parametr q ovlivní váhu jednotlivých znaků vstupního řetězce. Parametr q spolu s hodnotami znaků působí na výslednou hashovací hodnotu. Čím blíže je parametr q k limitě nabývající hodnoty 1, tím více poslední znaky ze vstupního řetězce ovlivní výslednou hashovací hodnotu.

Aby nedocházelo k „přetečení“, obsahuje vzorec konstantu c. Tato konstanta zajišťuje, že hodnota hashovací funkce zůstane na intervalu $< 0; h_{max} >$, kde h_{max} reprezentuje maximální hodnotu, která je určena použitým datovým typem. Konstanta c je vypočítávána podle uvedeného vzorce. Vzorec je upraven pro hashovací hodnoty reprezentované 64-bitovým číslem $h_{max} = 2^{64} - 1$.

$$c = \frac{(2^{64} - 1)(1 - q)}{L_{max}} \tag{2.6}$$

Ve vzorci je použita proměnná L_{max} reprezentující délku nejdelšího vstupního vektoru ze vstupních dat.

Pro optimalizaci výkonu výpočtu hashovací hodnoty může být velikost hashovací tabulky určena jako nejbližší druhá mocnina. Potom je ve vzorci Shash funkce nahrazena funkce $modulo$ za binární funkci and.

$$h(x) = \left\lfloor c \sum_{i=0}^{L_x-1} x_i \, q^i \right\rfloor \, and \, (HS - 1) \tag{2.7}$$

Velikost hashovací tabulky HS je vypočítána podle vztahu:

$$HS = 2^{\left\lceil \log_2\left(\frac{1}{f}N\right)\right\rceil} \tag{2.8}$$

Při implementaci vlastní aplikace je použita Shash funkce s binární funkcí. Ukázka použité implementace byla převzata z [1 str. 12].

```
double hash = 0;
double qValue = coefficientC * coefficientQ;

for (int i = 0; i < key.Length; ++i)
{
 hash += key[i] * qValue;
 qValue *= coefficientQ;
}

return (ulong)System.BitConverter.DoubleToInt64Bits(hash)
& this.secondPower;
```

2.4.5 Brian Kernighan, Dennis Ritchie funkce

Dále označována jako BKDR. Tato hashovací funkce byla publikována v knize The C Programming Language [8]. Autoři v knize popisují spolu s hashovací funkcí i strukturu použité hashovací tabulky. Pro ukázku v knize byla použita hashovací tabulka se zřetězeným rozptylováním. Výhoda BKDR funkce spočívá v její efektivitě. Ukázka použité implementace byla převzata z [1 str. 14].

```
ulong seed = 131;
ulong hash = 0;
for (int i = 0; i < key.Length; i++)
{
 hash = (hash * seed) + key[i];
}

return (hash % this.primeNumber);
```

2.4.6 Donald E. Knuth funkce

Hashovací funkce byla vytvořena panem Donaldem E. Knuthem a publikována v The Art Of Computer Programming Volume 3 [9]. Dále označována jako DEK. Implementace je upravena pro 64-bitovou hashovací hodnotu, protože původní návrh funkce přepokládal 32-bitovou hashovací hodnotu. Z tohoto důvodu je ve výsledné implementaci poslední krok algoritmu rozšířen o logický součin s velikostí hashovací tabulky. Ukázka použité implementace byla převzata z [1 str. 14].

```
ulong hash = (ulong)key.Length;

for (int i = 0; i < key.Length; i++)
```

```
{
 hash = ((hash << 5) ^ (hash >> 27)) ^ key[i];
}

return hash & this.secondPower;
```

2.4.7 Arash Partow funkce

Hashovací funkce byla vytvořena panem Arashem Partowem [10]. Dále označována jako AP. Autor hashovací funkce ve svém článku analyzoval několik jiných hashovacích funkcí. Na základě této analýzy vytvořil AP funkci. Ukázka použitě implementace byla převzata z [1 str. 14].

```
ulong hash = 0xAAAAAAAA;
for (int i = 0; i < key.Length; i++)
{
 if ((i & 1) == 0)
       hash ^= ((hash << 7) ^ key[i] * (hash >> 3));
 else
 hash ^= (~((hash << 11) + key[i] ^ (hash >> 5)));
}

return (hash % this.primeNumber);
```

2.4.8 Glen Fowler, Landon Curt Noll, Phong Vo funkce

Hashovací funkce byla vytvořena v roce 1991 pány, kteří se jmenují Glenn Fowler a Phong Vo [11]. Dále označována jako FNV. FNV funkce byla navržena tak, aby byla rychlá a zároveň docházelo k minimálnímu počtu kolizí. Proto je vhodná pro třídění podobných řetězců. Ukázka implementace byla převzata z [1 str. 15].

```
ulong fnv_prime = 1099511628211;
ulong fnv_offset = 14695981039346656037;

ulong hash = fnv_offset;
for (int i = 0; i < key.Length; i++)
{
 hash ^= key[i];
 hash *= fnv_prime;
}

return hash & this.secondPower;
```

Hodnoty fnv_prime a fnv_offset byly získány experimentálně. Tabulka hodnot je uvedena na webových stránkách [11].

2.4.9 Dan J. Bernstein funkce

Hashovací funkce byla vytvořena panem Danem Juliusem Bernsteinem. Dále označována jako DJB. Tato funkce je považována za jeden z nejefektivnějších algoritmů, protože dosahuje velmi dobrého rozložení prvků po hashovací tabulce [10]. Původní hashovací funkce DJB používala ve svém algoritmu sčítání. Obecná ukázka původní implementace v jazyce ansi C [12]:

```
unsigned djb_hash ( void *key, int len )
{
 unsigned char *p = key;
 unsigned h = 0;
 int i;

 for ( i = 0; i < len; i++ )
  h = 33 * h + p[i];

 return h;
}
```

Autor potom modifikoval funkci a nahradil operaci sčítání bitovým součtem. Modifikovaná verze obecné implementace vypadala takto:

```
. . .
  h = 33 * h ^ p[i];
. . .
```

V ukázkách obecných implementací autora se vyskytuje konstanta 33. Tato konstanta způsobuje lepší rozložení hashovacích hodnot v rámci hashovací tabulky. Proč při použití tohoto magického čísla dosahuje funkce nejlepších výsledků, nebylo zatím nikým pořádně vysvětleno. Při použití konstanty 33 nebo jiné dobré konstanty 17, 31, 63, 127 a 129 bylo rozložení prvků po hashovací tabulce v průměru až 86% [13].

Pro zvýšení rychlosti výpočtu bylo potom násobení nahrazeno bitovým logickým posunem. A konstanta 33 zaměněna za konstantu 5381. Ukázka implementace byla převzata z [1 str. 16].

```
ulong hash = 5381;
for (int i = 0; i < key.Length; i++)
{
 hash = ((hash << 5) + hash) + key[i];
}

return hash % this.primeNumber;
```

2.4.10 Java funkce

Hashovací funkce používaná v jazyce Javascript pro převod řetězce na celé číslo, dále označovaná jako Java, byla upravena pomocí funkce *modulo* pro výpočet indexů do hashovací tabulky [14]. Z ukázky použité implementace lze vidět, že se jedná o obdobu upravené DJB hashovací funkce.

```
ulong hash = 0;
for (int i = 0; i < key.Length; i++)
{
 hash = ((hash << 5) - hash) + key[i];
 hash = hash & hash;
}

return (hash % this.primeNumber);
```

2.4.11 SDMB funkce

SDMB funkce byla vytvořena pro knihovní databáze [14]. Jedná se o jinou obdobu, DJB funkce, která byla upravena pomocí funkce *modulo* pro výpočet indexů do hashovací tabulky. Ukázka použité implementace.

```
ulong hash = 0;

for (int i = 0; i < key.Length; i++)
{
 hash = key[i] + (hash << 6) + (hash << 16) - hash;
}

return (hash % this.primeNumber);
```

2.4.12 ELF funkce

Tato hashovací funkce je především používána v Unix systémech [10]. Ukázka implementace byla převzata z [15].

```
const uint c = 0xf0000000;

uint h = 0,
     g = 0;

unchecked
{
 for (int i = 0, len = key.Length; i < len; i++)
```

```
  {
    h = (h << 4) + key[i];
    if ((g = h & c) != 0) h ^= g >> 24;
    h &= ~g;
   }
 }
 return (h % this.primeNumber);
```

2.5 Kritéria porovnání hashovacích funkcí

Základním pojmem je slovo **bucket**. Tento pojem zastupuje velikost spojových seznamů vytvořených během vkládání dat do hashovací tabulky.

Následující podkapitoly se věnují ukázkám vzorců pro výpočet základních kritérií pro porovnání vlastností vybraných hashovacích funkcí.

2.5.1 Doba výpočtu

Základními požadavky pro hashovací funkce typu fast table lookup (rychlé dohledávání dat) jsou požadavky na rychlost výpočtu a na unikátnost hashovací hodnoty (eliminace kolizí). Proto je logickou vlastností těchto hashovacích funkcí i doba potřebná pro výpočet hashovací hodnoty. Otázkou zůstává jak požadovanou hodnotu měřit. Jestli má být měřena jako doba pro zpracování celého vstupního souboru nebo jako doba při jednotlivých výpočtech hashovacích hodnot. Jestliže bude hodnota měřena jako doba pro zpracování celého vstupního souboru, výsledná hodnota bude pravděpodobně v řádech $ms{\sim}s$. V druhém případě bude měřená hodnota pravděpodobně v řádech $\mu s{\sim}ms$. Naměřené hodnoty budou pro obecné porovnání neprůkazné, protože měření časových hodnot je velmi závislé na používaném hardwaru a softwaru.

2.5.2 Maximální bucket

Základní a nejjednodušším kritériem je maximální velikost vytvořeného spojového seznamu v hashovací tabulce (bucketu). Maximální velikost indikuje chování hashovací funkce pro nejhorší případ, tedy pro případ, kdy nastává nejvíce kolizí. Za předpokladu, že by se jednalo o „perfektní hashovací funkci“ by byla velikost maximálního bucketu rovna 1.

2.5.3 Lineární průměr obsazenosti bucketu

Dalším kritériem pro porovnání vlastnosti hashovacích funkcí je lineární průměr obsazenosti bucketu. Zde se počet všech uložených prvků v hashovací tabulce dělí počtem všech obsazených indexů. Hodnota I_a udává, kolik prvků průměrně připadá na jeden obsazený index v hashovací tabulce. Vzorec pro výpočet lineárního průměru obsazenosti bucketu byl odvozen od aritmetického průměru.

Poznámka:

I_a - lineární průměr obsazenosti bucketu
N - počet všech prvků uložených v tabulce
M - počet všech obsazených indexů v hashovací tabulce

$$I_a = \frac{N}{M} \tag{2.9}$$

2.5.4 Kvadratický průměr obsazenosti bucketu

Předposledním kritériem pro porovnání vlastností hashovacích funkcí je kvadratický průměr obsazenosti bucketu, který je počítán pomocí vzorce:

Poznámka:

I_{a2} - kvadratický průměr obsazenosti bucketu
I - velikost bucketu
C - počet bucketu dané velikosti
M - počet všech obsazených indexů v hashovací tabulce

$$I_{a2} = \sqrt[2]{\frac{\sum_{I=1}^{I_m} I^2 C_I}{M}} \tag{2.10}$$

Vzorec pro výpočet kvadratického průměru obsazenosti bucketů byl odvozen od klasického vzorce pro výpočet kvadratického průměru. Pomocí tohoto vzorce jsou získány výsledky. Tyto výsledky umožňují získat přesnější představu o průměrném rozložení prvků na obsazených indexech hashovací tabulky.

2.5.5 Kritérium Q

Značeno jako Q. Udává celkovou „cenu“ za vložení všech prvků do hashovací tabulky. Vychází z předpokladu, že se do hashovací tabulky nesmějí uložit dva identické prvky. Tato celková „cena“ je vypočítána pomocí vzorců [4] [7]. Vzorec je sestaven z několika případů, které nastávají během ukládání do hashovací tabulky. Kritérium Q jako celková „cena“ za vložení všech prvků do hashovací tabulky není vhodná pro porovnávání hashovacích funkcí, protože hodnota kritéria Q je přímo závislá na použitých datech a obecně o vlastnostech dané hashovací funkce nic neříká. Za předpokladu, že by se porovnávaly hashovací funkce podle kritéria Q, tak by musely být všechny porovnávané funkce testovány na identických datech (typově stejných např.

cizojazyčné slovníky). Proto je pro porovnávání zavedena další veličina relativní kritérium Q', která umožňuje porovnávat hashovací funkce na obecných datech.

Poznámka:

I - velikost bucketu
C - počet bucketu dané velikosti
N - počet všech prvků uložených v tabulce
Q - celková „cena“
$Q_{1,2,3}$ - jednotlivé „ceny“ za postupné vkládání

1. Bucket pro daný index je prázdný (není obsazený). „Cena“ vložení Q je vyjádřena pomocí následujícího výrazu:

$$Q_1 = 0 \tag{2.11}$$

2. Prvek není v bucketu uložen. Prohledávání a uložení do bucketu. „Cena“ vložení Q je vyjádřena pomocí vzorce:

$$Q_2 = \sum_{I=1}^{I_m} I^2 C_I \tag{2.12}$$

3. Prvek je v bucketu už uložen. Prohledávání bucketu a nalezení identického prvku. „Cena“ prohledání je vyjádřena pomocí vzorce:

$$Q_3 = \sum_{I=1}^{I_m} \frac{1}{2} I^2 C_I \tag{2.13}$$

Celková „cena“ vložených prvků může být vyjádřena jako součet předchozích „cen“. Vzorec pro výslednou „cenu“ je vyjádřen níže uvedeným vztahem:

$$Q = Q_1 + Q_2 + Q_3 = \frac{3}{2} \sum_{I=1}^{I_m} I^2 C_I \tag{2.14}$$

2.5.6 Relativní kritérium Q'

Značeno jako Q'. Udává průměrnou „cenu“ za vložení nové hodnoty do hashovací tabulky. Hashovací funkce s nejmenší hodnotou Q' je v podstatě nejlepší. Relativní kritérium Q' je získáno jako podíl celkového ceny Q a počtu všech prvků v tabulce N. Relativní kritérium Q' je vypočítáno pomocí vzorce uvedeného v materiálech [4] [7].

Poznámka:

N - počet všech prvků uložených v tabulce
Q - celková „cena“
Q' - relativní kritérium

$$Q' = \frac{Q}{N} \tag{2.15}$$

Díky tomuto lze relativní kritérium Q' využít při porovnávání hashovacích funkcí na obecných datech (typově obecných, např. vlastnosti hashovacích funkcí pro cizojazyčné slovníky vs. vlastnosti pro chemické názvy), protože relativní kritérium Q' udává „cenu“ za jeden vložený prvek.

3 Realizační část

3.1 Volba kritérií pro porovnávání

V Teoretické části bakalářské práce je popsáno několik základních kritérií pro porovnávání hashovacích funkcí. Pomocí těchto kritérií můžou být použité hashovací funkce porovnávány. Seznam použitých kritérií pro porovnání hashovacích funkcí.

1. relativní kritérium Q'
2. velikost maximálního bucketu
3. Lineární a kvadratická obsazenost bucketů
4. kritérium Q

V následujících odstavcích jsou odůvodněna použitá kritéria. Základním kritériem pro porovnání hashovacích funkcí je bezpochyby relativní kritérium Q'. Při zpracovávání výsledků jsou všechny použité hashovací funkce porovnávány podle něho. Protože pro výpočet relativního kritéria Q' je zapotřebí spočítat kritérium Q, je i kritérium Q obsaženo ve výstupním souboru aplikace. Na základě relativního kritéria Q' se může rozhodnout, která funkce je schopna nejlépe naplnit hashovací tabulku.

Dalším kritériem pro porovnávání je počet kolizí, které vzniknou při vkládání nových prvků do hashovací tabulky. Toto kritérium lze posuzovat podle třech základních hodnot:

- velikost maximálního bucketu
- lineární průměr obsazenosti bucketů
- kvadratický průměr obsazenost bucketů

Tyto hodnoty popisují, jak jsou vstupní data rozložena po hashovací tabulce. Hodnota maximálního bucketu představuje největší počet dosažených kolizí pro index v rámci hashovací tabulky. Lineární a kvadratický průměr obsazenosti ukazuje rozložení prvků v rámci obsazených indexů v hashovací tabulce.

Posledním měřitelným kritériem je doba výpočtu. Vlastní aplikace neměří konkrétní doby pro jednotlivé hashovací funkce, a tudíž nejsou funkce vůbec podle času porovnávány. Aplikace měří pouze celkovou dobu v rámci pracovního času aplikace. Tato hodnota slouží pouze jako informace uživateli a není zanášena do výstupního souboru.

3.2 Volba programovacího jazyka

Bakalářská práce, která navazuje na Projekt 5[1], je implementována pomocí programovacího jazyka C#. Tento jazyk byl použit pro implementaci Projektu 5 na žádost vedoucího bakalářské práce. Programovací jazyk byl zvolen proto, aby bylo možné porovnat výsledky Projektu 5 s výsledky bakalářské práce pana Citriaka [1]. Cílem práce je zjistit chování vybraných hashovacích funkcí pro atypická data. Testovaná data byla vybrána z oboru chemie, jako jsou názvy kyselin a jiných sloučenin. Důvodem pro vybrání tohoto typu vstupních dat jsou velmi dlouhé názvy jednotlivých sloučenin. Použité chemické názvy nespadají do určité kategorie chemie. Jedná se o průřez napříč kategoriemi. Použitá chemická data byla získána z veřejně dostupných databází nebo byla zapůjčena vlastníky databází pro účel bakalářské práce.

3.3 Popis programovacího jazyka C#

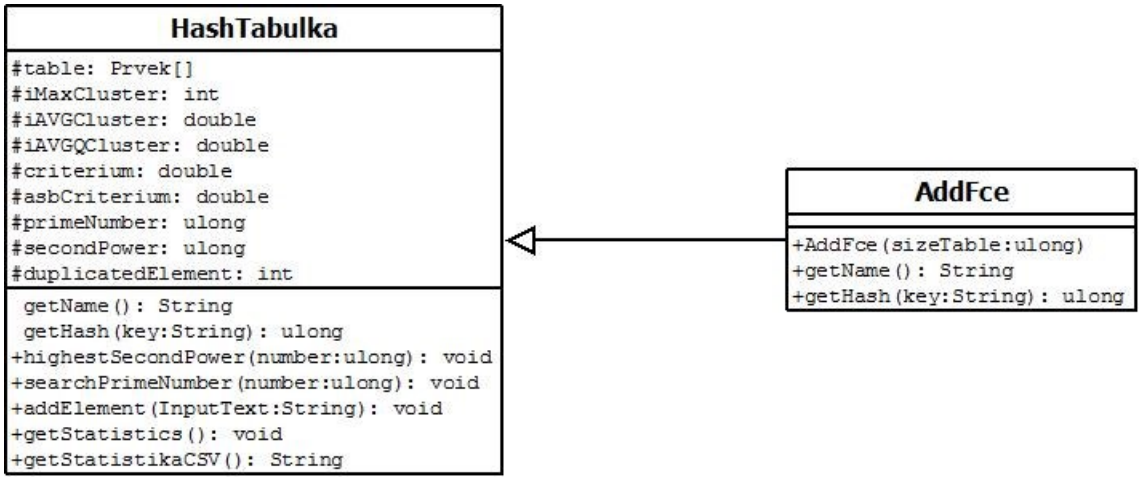
Programovací jazyk C# byl vyvinut firmou Microsoft a vychází z programovacích jazyků C/C++ a je velmi podobný programovacímu jazyku Java. Stejně jak Java má i C# virtuální stroj. V případě C# se virtuální stroj označuje zkratkou CLR (Common Language Runtime). Překlad vlastního zdrojového kódu má potom několik částí. Zdrojový kód se pomocí kompileru přeloží na mezikód, označovaný jako CIL (Common Intermediate Language). Jedná se o binární kód s jednodušší instrukční sadou, který podporuje objektové programování. Tento mezikód je potom přeložen pomocí CLR do strojového kódu určeného pro procesor. Přístup použitý pro překlad zdrojových kódu jazyka C# má několik výhod. Zdrojový kód je přenositelný nezávisle na platformě. Daná aplikace bude fungovat všude, kde funguje virtuální stroj [16].

Programovací jazyk C# je čistě objektově orientovaný se silnou typovou kontrolu, používá pouze jednoduchou dědičnost s možností několikanásobné implementace rozhraní. Jednoduchá dědičnost je zde použita proto, aby se předešlo problémům a velké složitosti při vícenásobné dědičnosti, jako je v jazyce C++. To znamená, že každá třída může dědit pouze od jedné třídy, ale může implementovat několik rozhraní. Správa paměti je zajištěna garbage collectorem jako v Javě. Proto je správa paměti automatická a nemůže dojít k ztrátě paměti (memory leak) jako v ansi C. Vývoj vlastní aplikace je „jednoduchý“, k dispozici je mnoho základních knihoven. Dále podporuje zpracování chyb pomocí vyjímek.

[1] výsledky Projektu 5 dostupné na http://home.zcu.cz/~radekp25/

3.4 Implementace

Pro implementaci vlastní aplikace je využitá jednoduchá dědičnost. Dědičnost se používá především pro snadnou implementaci jednotlivých hashovacích funkcí. Vztah dědičnosti je zobrazen v následném UML diagramu (viz Obrázek 2.5.6-1: UML diagram).



Obrázek 2.5.6-1: UML diagram

Z UML diagramu je vidět, že každá použitá hashovací funkce dědí ze třídy HashTabulka. Tato třída obsahuje všechny společné metody a atributy pro hashovací funkce. Nově založená třída hashovací funkce musí implementovat virtuální metody třídy HashTabulka a svůj konstruktor. Dále pro vlastní implementaci jsou využity standartní knihovny jazyka C#.

Jednoduché GUI (Graphical User Interface), které je součástí aplikace, bylo vytvořeno pomocí nástroje WPF (Window Presentation Foundation). Tento nástroj slouží pro tvorbu jednoduchých formulářových aplikací. Pro účel vlastní aplikace byl tento nástroj naprosto dostatečný. Cílem bakalářské práce je vytvoření aplikace pro experimentální porovnání hashovacích funkcí pro textové účely, proto zde nebudou ukázky zdrojového kódu pro vytvoření GUI.

3.4.1 Použité datové struktury

Klíčovou datovou strukturou je hashovací tabulka se zřetězeným rozptylováním. Tato tabulka je implementována jako jednorozměrné pole prvků, kde jsou jednotlivé prvky tvořeny pomocí třídy Prvek.

Jednotlivé instance třídy Prvek slouží k vytvoření jednoduchého spojového seznamu. Díky tomu lze vytvořit hashovací tabulku se zřetězeným rozptylováním.

Ukázka implementace třídy Prvek:

```
class Prvek
```

```
{
 private String inputText;
 private Prvek next;

 public Prvek(String str)
 {
   this.inputText = str;
   this.next = null;
 }
}
```

Pro správné objektově zapouzdření nejsou v ukázce implementace třídy Prvek zobrazeny „gettry“ a „settry“, nicméně jsou naimplementovány ve vlastní aplikaci.

3.4.2 Kódování vstupních a výstupních dat

Při načítání vstupních dat vzniká problém s kódováním, protože vstupní data můžou mít různé znakové sady. Problém kódování je vyřešen pomocí použití přednastavené znakové sady daného operačního systému. Proto musí být na vstupní data kladen požadavek, aby byly uloženy ve stejném kódování, jako je přednastavené kódování operačního systému. Tento požadavek je uveden v Uživatelské dokumentaci, která je umístěna v příloze bakalářské práce. Pro systém Windows 7 s českou lokalizací to znamená používat kódování Windows-1250 také označovaného jako CP-1250.

Výstupní data jsou ukládána do CSV souboru se znakovou sadou operačního systému. CSV soubor je jednoduchý textový soubor pro ukládání tabulkových dat. Jednotlivé sloupce jsou od sebe odděleny středníky a každá řádka v souboru reprezentuje jednu řádku v tabulce. CSV soubory se můžou dále rozlišovat podle toho, jaký používají oddělovač sloupců. Některé CSV soubory používají místo středníku tabulátor nebo čárku.

Práci se soubory zajišťuje ve vlastní aplikaci třída FileWorker. Pro načtení vstupních dat je využita kolekce ArrayList z knihovny System.Collections. Dále tato třída během načítání vstupního souboru zajišťuje zjištění základních informací potřebných pro pomocné výpočty hashovacích funkcí. Těmito informacemi jsou velikost vstupního souboru (počet všech vstupních vektorů spolu s duplicitními daty) a délka nejdelšího vstupního vektoru. Vlastní třídění duplicitních vektorů probíhá během ukládání do hashovací tabulky. Velikost vstupních dat je důležitá pro vlastní inicializaci velikostí všech hashovacích tabulek. Pro korektní výpočet Shash funkce je zapotřebí největší délka vstupního vektoru. Vlastní implementace třídy FileWorker se skládá z načítání a ukládání souborů. V praxi se jedná o základní použití knihoven pro práci se soubory, proto zde není uvedena žádná ukázka implementace.

3.4.3 Implementace třídy HashTabulka

Tato třída je hlavní třídou celé aplikace. Z této třídy dědí všechny použité hashovací funkce. Před vlastní implementací třídy se musí třída rozdělit na části, které jsou

pro každou hashovací funkci unikátní a části, které mají společné. Unikátními částmi jsou jméno hashovací funkce a její vlastní algoritmus pro výpočet indexu do hashovací tabulky, a proto budou implementovány pomocí virtuálních metod. Tyto metody jsou v jazyce C# označovány slovem virtual. V Javě stejné metody označujeme slovem abstract. Ze třídy, která má takto implementované metody, nelze udělat přímou instanci, lze od ní pouze dědit. Ukázka implementace:

```
class HashTabulka
{
 protected Prvek[] table;
 public HashTabulka()
 {
 }

 public virtual string getNameFce
 {
   get
   {
     throw new NotImplementedException("Hash name not
defined!");
   }
 }

 public virtual ulong getHash(string key)
 {
   throw new NotImplementedException("Hash algorithm not
defined!");
 }
}
```

Potom každá děděná třída bude požadovat implementaci těchto dvou virtuálních metod. Pokud virtuální metody nebudou naimplementovány, nepůjde přeložit ani spustit zdrojový kód. Překlad skončí výjimkou uvedenou ve zdrojovém kódu.

Společnými částmi hashovacích funkcí jsou výpočet velikosti hashovací tabulky, vkládání nových prvků a výpočet statistických hodnot potřebných pro porovnání hashovacích funkcí.

Výpočet potřebné velikosti je závislý na algoritmu dané hashovací funkce. Funkce, které používají matematickou funkci *modulo* vyžadují prvočíslo. Funkce, které používají bitovou funkci *and* vyžadují číslo, které je druhou mocninou. Pro hledání prvočísla je v třídě HashTabulka statické pole několika prvočísel. Pro hledání druhé mocniny je ve třídě implementována metoda. Do třídy HashTabulka byly přidány tyto proměnné a metody. Ukázka upravené implementace:

```
class HashTabulka
{
 . . .
```

```
private static ulong[] primeNumbers = {1, 2, 5, 11, 17,
37, 67, 131, 257, 521, 1031, 2053, 4099,8209, 16411,
32771, 65537, 131101, 262147, 524219, 1048583, 2097169,
4194319, 8388617, 16777259, 33554467, 67108879, 134217757,
268435459, 536870923, 1073741827, 2147483693};

public void searchPrimeNumber(ulong number)
{
  foreach (ulong tempPrimeNumber in primeNumbers)
  {
   if (tempPrimeNumber > number)
   {
    this.primeNumber = tempPrimeNumber;
    break;
   }
  }
 }

 public void highestSecondPower (ulong number)
 {
  double logValue = System.Math.Log(number, 2);
  int exponent = (int)System.Math.Ceiling(logValue);
  this.primeNumber = (ulong)System.Math.Pow(2, exponent);
  this.secondPower = this.primeNumber - 1;
 }
}
```

Zdrojový kód z předchozí ukázky byl nahrazen třemi tečkami. Metoda pro výpočet druhé mocniny byla převzata z [1].

Další společnou částí je metoda pro vkládání prvků do hashovací tabulky. Tato metoda musí zajistit vložení každého nového prvku a eliminaci duplicitních prvků. Poté třída HashTabulka musí být upravena o přidání další metody. Ukázka implementace metody:

```
. . .
public void addElement (String Inputtext)
{
 ulong hashKey = getHash(Inputtext);
 Prvek newElement = new Prvek(Inputtext);
 bool duplicated = false;

 if (table[hashKey] == null)
 {
  table[hashKey] = newElement;
 }
 else if (table[hashKey] != null)
 {
  Prvek tempPointer = table[hashKey];

  while (tempPointer.Next != null)
  {
```

```
   if (tempPointer.getText.Equals(newElement.getText) ==
true)
   {
     duplicated = true;
     duplicatedElement++;
     break;
    }
    tempPointer = tempPointer.Next;
   }

   if (duplicated == false)
   {
//test last element from bucket before adding new element
    if (tempPointer.getText.Equals(newElement.getText) ==
false)
    {
     tempPointer.Next = newElement;
    }
    else
    {
     duplicatedElement++;
    }
   }
  }
  else
  {
  throw new Exception("Prvek: " + hashKey + ". " +
Inputtext + "se nepovedlo přidat");
  }
}
```

Třemi tečkami je zde nahrazena implementace předchozích metod. Jedná se o while cyklus pro přidání prvku do spojového seznamu, který kontroluje názvy jednotlivých prvků pro odstranění duplicit. Jelikož se jedná o while cyklus musí být poslední prvek spojového seznamu kontrolován za vlastním tělem cyklu. Takto je zaručena eliminace všech duplicitních prvků. Pro statistiku vstupních dat je zde proměnná, která počítá duplicitní prvky.

Poslední společnou částí je výpočet statistických údajů potřebných pro porovnání vlastností jednotlivých hashovacích funkcí. Proto se musí do třídy HashTabulka přidat další metoda pro výpočet statistiky a proměnné pro uložení výsledků. Následně pak stačí zavolat metodu, která vrátí řetězec s vypočítanými výsledky ve formátu pro CSV soubor.

Výpočet jednotlivých statistických údajů vychází ze vzorců uvedených v podkapitole Kritéria porovnání hashovacích funkcí. V následujících podkapitolách budou uvedeny jednotlivé ukázky implementace použitých vzorců.

3.4.4 Implementace výpočtu max. Bucketu

Pro vlastní výpočet maximální bucketu se prochází celá hashovací tabulka. Při procházení se hledá spojový seznam největší délky. Dále se během procházení zaznamenává počet obsazených indexů v tabulce a celkový počet vložených prvků. Počty obsazených indexů a celkový počet prvků jsou potřeba pro výpočet lineární a kvadratické obsazenosti bucketů. Ukázka použité implementace:

```
public void getStatistics()
{
 ulong N = 0;
 int numberBucket = 0;
 ulong numberUseBucket = 0;

 Prvek tempPointer = null;
 //max size bucket
 foreach (Prvek elementTable in table)
  {
  if (elementTable != null)
  {
   numberUseBucket++;
   tempPointer = elementTable;
   while (tempPointer != null)
   {
    N++;
    numberBucket++;
    tempPointer = tempPointer.Next;
   }

  if (numberBucket > this.iMaxCluster) this.iMaxCluster =
numberBucket;
  numberBucket = 0;
 }
}
```

3.4.5 Implementace výpočtu lineární obsazenosti bucketů

Výpočet lineární obsazenosti se provádí podle vzorce (2.9). Při implementaci vzorce se jedná o jednoduché dělení dvou celých nezáporných čísel. Při samotném dělení se musí dělitel a dělenec přetypovat na desetinné číslo (double nebo float), aby i výsledkem dělení bylo desetinné číslo. Ukázka implementace výpočtu lineární obsazenosti:

```
. . .
this.iAVGCluster = (double)N / (double)numberUseBucket;
. . .
```

3.4.6 Implementace výpočtu kvadratické obsazenosti bucketů

Výpočet kvadratické obsazenosti se provádí podle vzorce (2.10). Pro samotný výpočet je potřeba zjistit velikosti všech bucketů, které jsou obsaženy v tabulce a jejich celkový počet. Pro uložení těchto hodnot je vytvořeno jednorozměrné pole, kde index pole reprezentuje velikost bucketu a hodnota v poli reprezentuje počet bucketů dané velikosti. Velikost pole je dána předpisem $velikost\ maximálního\ bucketu + 1$. Tímto posunem potom index pole reprezentuje jednotlivé velikosti spojových seznamů. S indexem 0 nastává problém. Tento problém může být řešen tak, že průchod polem hodnot začíná na indexu 1 nebo je řešen samotnou podstatou vzorce. V případě použitého vzorce pro kvadratickou obsazenost bucketu tento problém nenastává, protože při násobení 0 vždy dostaneme 0. Ukázka implementace pro získání pole hodnot:

```
. . .
int[] pole = new int[this.iMaxCluster + 1];
foreach (Prvek polozka in table)
 {
  if (polozka != null)
  {
   tempPointer = polozka;
   while (tempPointer != null)
   {
    numberBucket++;
    tempPointer = tempPointer.Next;
   }
   pole[numberBucket]++;
   numberBucket = 0;
  }
 }
. . .
```

Pole hodnot je důležité pro výpočet kvadratické obsazenosti bucketů a dále pro výpočet kritérií $Q\ a\ Q'$. Samotné vypočítání kvadratické obsazenosti je už pouhé dosazení do vzorce. Ukázka implementace:

```
. . .
double tempCompute = 0.0;

for (int i = 1; i < pole.Length; i++)
{
 tempCompute += Math.Pow(i , 2) * pole[i];
}

this.iAVGQCluster = Math.Sqrt((tempCompute /
numberUseBucket));
. . .
```

3.4.7 Implementace výpočtu kritérií Q a Q'

Výpočet kritéria Q se provádí podle vzorce (2.14) a výpočet relativního kritéria Q' se provádí podle vzorce (2.15). Pro vlastní výpočet kritérií jsou získány všechny potřebné proměnné z předchozích výpočtů. Pro vypočítání kritérií stačí pouze dosadit do výše uvedených vzorců. Ukázka implementace:

```
. . .
double tempSum = 0;

//compute criterium Q
for (int i = 1 ; i <= this.iMaxCluster ; i++)
{
 tempSum += Math.Pow(i, 2) * pole[i];
}

this.criterium = 1.5 * tempSum;

//compute realtive criterium Q
this.absCriterium = (this.criterium / N);
. . .
```

3.5 Ukázka implementace hashovací funkce

Implementace vlastní hashovací funkce bude ukázána na základní hashovací funkci. Každá hashovací funkce je přidána do aplikace jako třída, která je děděna od třídy HashTabulka. Pro vlastní implementaci každé nové hashovací funkce se musí minimálně naimplementovat konstruktor dané hashovací funkce a virtuální metody getHash() a getName().

V konstruktoru každé hashovací funkce je vytvářena hashovací tabulka. A pokud některá hashovací funkce vyžaduje výpočet konstant, jsou všechny potřebné metody pro výpočet konstant volány zde. Ukázka implementace aditivní funkce:

```
class AddFce : HashTabulka
{

 public AddFce(ulong sizeTable)
 {
  searchPrimeNumber(sizeTable);
  this.table = new Prvek[this.primeNumber];
 }

 public override string getNameFce
 {
  get
  {
   return "Aditivní funkce";
```

```
  }
 }

 public override ulong getHash (string key)
 {
  ulong hash = (ulong)key.Length;
  for (int i = 0; i < key.Length; ++i)
  {
   hash += key[i];
  }

  return (hash % this.primeNumber);
  }
}
```

Pro implementaci virtuálních metod musí být použito klíčové slovo override. Při korektní implementaci by měl konstruktor děděné třídy obsahovat metodu base () tato metoda volá konstruktor třídy, od které se dědí. Takto dojde k inicializaci děděných proměnných. Pokud není metoda base () volána ve zdrojovém kódu konstruktoru, zajistí její volání překladač. V programovacím jazyce Java je stejná metoda pojmenována názvem super () a umožňuje přístup i do překrytých metod rodiče.

Po vytvoření třídy se v logické třídě musí vyrobit instance dané hashovací funkce a postupně v logickém pořadí volat implementované a děděné metody. Tímto postupem se může do vlastní aplikace přidat neomezené množství hashovacích funkcí. Ukázka implementace Logické třídy:

```
class LogickaTrida
{
 . . .
 void runCompute(ArrayList vocabulary, ulong longestWord)
 {
  FileWorker fileWorker = new FileWorker();
  ulong number = 0;
  ulong sizeVocabulary = (ulong)vocabulary.Count;

  AddFce aditivniFce = new AddFce(sizeVocabulary);

  foreach (string slovo in vocabulary)
  {
   number++;
   PrubehZpracovani(number, vocabulary.Count);

   if (workFlag == true)
   {
    processingWord = slovo;
   }
   else
   {
    processingWord = reverseText(slovo);
   }
```

```
   processingWord = processingWord.Trim();
   aditivniFce.addElement(processingWord);
  }

 aditivniFce.getStatistics();

 word +=
"JmenoFce;Max.Bucket;kriterium;abskriterium;Ia;I2a" + "\n"
+aditivniFce.getNameFce+";"+aditivniFce.getStatistikaCSV()
+ "\n";
}
. . .
```

Tečky v ukázce zdrojového kódu zastupují implementaci konstruktoru a metod pro výpočet pomocných hodnot jako je hodnota progress baru a jiných doplňkových záležitostí, které přímo nesouvisejí se zadáním bakalářské práce, a proto jejich ukázka implementace není nutná.

3.6 Struktura výstupního souboru

Výstupní soubor je ve formátu CSV s přednastavenou znakovou sadou daného operačního systému, který je popsán v podkapitole 3.4.2. Začátek výstupního souboru je tvořen hlavičkou. Hlavička výstupního souboru obsahuje obecné informace o vstupních datech. Těmito informacemi jsou:

1. absolutní cesta ke vstupním datům např. C:\...\ slovnik-substances.txt
2. postup při načítání vstupního řetězce (normální / reverzní)
3. nejdelší vstupní vektor ze vstupních dat (nejdelší slovo)
4. počet nalezených duplicitních vstupních vektorů

Další část výstupního souboru tvoří vlastní tabulka, která obsahuje naměřené hodnoty, viz Tabulka 3.4.7-1: Ukázka struktury výstupního souboru.

Název hashovací funkce	Maximální bucket	Kritérium Q	Relativní kritérium Q'	Lineární obsazenost bucketu I_a	Kvadratická obsazenost bucketu I_{a2}
Aditivní funkce	8	17743,5	2,967636729	1,522148676	1,73535478

Tabulka 3.4.7-1: Ukázka struktury výstupního souboru

Hodnoty uložené v tabulce jsou uloženy na výchozí počet desetinných míst, který je umožněn programovacím jazykem C#. Tento přístup byl zvolen pro maximální flexibilitu dat tak, aby výstupní data vyhovovala co nejvíce požadavkům. O zaokrouhlení výstupních dat potom rozhoduje ten, kdo je zpracovává.

3.7 Interpretace naměřených hodnot

Hodnoty měřené v bakalářské práci jsou většinou poměry mezi jednotlivými proměnnými nebo hodnoty počítané podle specifický vzorců. Tyto hodnoty nejsou zastoupeny v Tabulce SI soustavy a jako takové nemají oficiální jednotky. Proto ani ve výstupním souboru nejsou žádné jednotky pro vypočítané hodnoty uváděny. Jednotky pro jednotlivé vypočítané hodnoty se můžou odvodit ze vzorců, podle kterých byly spočítány pro lepší představu toho, co tyto hodnoty reprezentují. Jediná měřená hodnota, která spadá do soustavy SI, je čas. Respektive doba, kterou vlastní aplikace potřebuje pro zpracování vstupního souboru. Tato hodnota je měřena pouze orientačně a není ani uváděna ve výstupním souboru. Hodnota času je měřena ve vteřinách.

3.7.1 Odvození jednotek výstupních hodnot

Z metody pro výpočet maximálního bucketu 2.5.2 může být odvozena jednotka. Touto jednotkou je největší počet prvků na obsazený index hashovací tabulky.

Ze vzorců pro výpočet lineární a kvadratické obsazenosti bucketů 2.5.3 a 2.5.4 lze odvodit, že tyto hodnoty reprezentují průměrný počet prvků na obsazený index v hashovací tabulce.

Vzorec pro výpočet relativního kritéria 2.5.6 udává jakousi „cenu“ pro vložení nového prvku do hashovací tabulky. Tato „cena“ nelze vyjádřit specifickými jednotkami. Rozdíl mezi kritériem Q a relativním kritériem Q' spočívá v tom, že kritérium Q udává „celkovou cenu“ za naplenění celé hashovací tabulky na rozdíl od relativního kritéria Q', které udává „cenu“ za vložení jednoho prvku. Přesto je tato hodnota nejdůležitější pro vlastní zpracování výstupního souboru, protože podle ní jsou řazeny použité hashovací funkce.

3.7.2 Zaokrouhlování výstupních hodnot

Pro smysluplné zpracování výstupních hodnot je doporučeno si tyto hodnoty rozumně zaokrouhlit. Vlastní zaokrouhlení hodnot může potom vypadat takto.

Velikost maximálního bucketu není potřeba většinou zaokrouhlovat, protože se vždy bude jednat o celé nezáporné číslo. Jediný důvod pro jeho zaokrouhlení nastává v případě, že hashovací funkce selžou na vstupních datech a v podstatě zdegenerují hashovací tabulku do několika spojových seznamů. V takovém případě by mělo smysl tuto hodnotu zaokrouhlit na nějakém vyšším řádu.

Lineární a kvadratický průměr obsazenosti bucketu je zaokrouhlován podle výsledných hodnot. Z povahy výpočtu těchto hodnot se vždy musí jednat o nezáporné hodnoty větší než 1. Pokud by tyto hodnoty byly menší než 1 nebo záporné indikuje to chybu při implementaci. Následná chyba se může vyskytovat v implementaci samotných vzorců nebo v implementaci celého postupu vkládání dat do hashovací tabulky od vložení nového prvku až po samostatnou strukturu hashovací tabulky.

V kapitole 4, ve které jsou uvedeny získané výsledky, jsou tyto hodnoty zaokrouhlené na dvě desetinná čísla.

Kritérium Q udává celkovou „cenu“ za naplnění hashovací tabulky. Tato hodnota lze použít pouze pro porovnání hashovacích funkcí na konkrétních datech. Pro porovnání hashovacích funkcí na větší sadě vstupních souborů je tato hodnota bezpředmětná, proto je zavedena hodnota relativního kritéria Q'. Tato hodnota může být použita pro relevantní porovnání. Zaokrouhlení této hodnoty je doporučeno podle výstupních dat. Nelze doporučit desetinný řád pro zaokrouhlení, protože rozdíl v relativních kritériích nastává v průměrných případech, až na nějakém vyšším desetinném řádu. Při špatné volbě řádu dochází k znehodnocení výstupních dat. Toto znehodnocení má za důsledek nemožnost objektivního porovnání hashovacích funkcí vůči sobě.

3.8 Ověření výsledků

3.8.1 V závislosti na implementaci

Práce s hashovacími funkcemi zahrnuje mnoho aritmetických operací. Od práce s celými čísly až po vyhodnocování výsledků, které jsou většinou reprezentovány desetinnými čísly. Zde nastává zásadní problém, protože reprezentace desetinného čísla je dána obecnou normou IEEE 754[2], ale ve skutečnosti je přesnost desetinného čísla přímo závislá na použitém programovacím jazyce nebo na použitém kompilátoru. Proto by pro adekvátní porovnávání měly být všechny hashovací funkce implementovány ve stejném programovacím jazyce tak, aby stejná aritmetická chyba desetinného čísla byla přenesena do všech výsledků stejně.

3.8.2 Shrnutí

Ověření výsledku bakalářské práce vůči jiným výsledkům zatím není pravděpodobně možné, protože není známa žádná jiná práce zabývající se touto tématikou se zaměřením na chemické názvy. Ověření výsledků bylo možné při Projektu 5, ze kterého tato bakalářská práce vychází. Projekt 5 byl zaměřen na použití hashovacích funkcí vůči cizojazyčným slovníkům stejně jako bakalářská práce pana Citriaka [1]. Ověření výsledků z Projektu 5 a z práce pana Citriaka [1] vypadalo uspokojivě. Jednotlivé výsledky prací se lišily ve vyšších desetinných řádech. Tyto rozdíly byly přičítány použití jiných cizojazyčných slovníků. V pozdějším zpracování bakalářské práce se ukázalo, že tyto chyby byly zaviněny implementační chybou při načítání vstupního vektoru.

[2] **IEEE Std 754-2008**. IEEE Standard for Floating-Point Arithmetic. doi: 10.1109/IEEESTD.2008.4610935

Na implementační chybu poukázala hashovací funkce Shash 2.4.4, která vychází z přidělování vah k jednotlivým znakům vstupního vektoru. Tato funkce při reverzním načítání vstupního vektoru úplně degradovala hashovací tabulku do několika spojových seznamů. V podstatě to znamenalo, že na začátku každého vstupního vektoru musí být velká shoda znaků. Namátková kontrola vstupních dat tuto možnost vyloučila. Později se ukázalo, že vstupní data nebyla zbavena bílých znaků. To v praxi způsobovalo, že při normálním načítání vstupního vektoru byly tyto znaky na konci a nemohly tolik ovlivnit výslednou hashovací hodnotu, na rozdíl od reverzního načítání vstupních vektorů, kde byly bíle znaky na začátku. Zde tyto bílé znaky měly největší váhu na výslednou hashovací hodnotu, a proto docházelo k degradaci hashovací tabulky.

Ostatní hashovací funkce nemohli na implementační chybu upozornit tak silnou reakcí, protože nejsou založeny na přidělování priorit k jednotlivým znakům vstupního vektoru. Tato v podstatě banální implementační chyba byla vyřešena pomocí metody Trim() ve třídě String. Po opravení chyby se zlepšily vlastnosti všech použitých hashovacích funkcí.

Kontrola výstupních dat bakalářské práce spočívá pouze v porovnání číselných řádů naměřených hodnot s již naměřenými hodnotami.

4 Dosažené Výsledky

4.1 Popis testovaných dat

Testovaná data byla získána z různých zdrojů. Zdroje, odkud byla jednotlivá data získána, jsou uvedeny u každého testování spolu s krátkým popisem.

Jelikož originální soubory získané z databází byly v různých formátech, které byly nevhodné pro vlastní testování hashovacích funkcí, byly všechny vstupní soubory převedeny do jednoduchého formátu v podobě TXT. Díky tomu všechny vstupní soubory odpovídají požadované podmínce na kódování a podmínce na strukturu souboru (jeden řádek souboru se rovná jedné chemické sloučenině). Podrobný popis požadavků na vstupní soubor je uveden v Uživatelské dokumentaci, která je součástí přílohy.

4.2 Postup při testování vlastností

V podkapitole Testovaná data, jsou uvedeny jednotlivé testované vstupní soubory spolu s výsledky jednotlivých vlastností. Každé testování obsahuje informace (tabulky) o výsledcích dosažených při použití normální a reverzní strategie načítaní vstupního vektoru. Vypočítané výsledky jsou vizuálně zobrazeny pomocí grafů. Dále jsou všechny hashovací funkce porovnávány vůči relativnímu kritériu Q' funkce Shash. V uvedených výsledcích není zaznamenáno kritérium Q, protože není potřeba k žádnému porovnávání. Získaná data jsou zaokrouhlena podle návrhu uvedeného v podkapitole 3.7.2 Zaokrouhlování výstupních hodnot.

4.3 Parametr q funkce Shash

Parametr q funkce Shash byl experimentálně získán pro každý vstupní soubor pomocí krátké simulace. Parametr q byl simulován pro normální načítání vstupního vektoru. Simulace probíhala na intervalu $< 0{,}8; 1)$. Tento interval byl určen z předchozí zkušenosti z práce na Projektu 5, kde bylo zjištěno, že pro klasická slovníková data je ideální interval $< 0{,}9; 1)$.

Při použití parametru q menšího než $0{,}8$ docházelo k degeneraci hashovací tabulky. V rámci chemických dat se pohyboval parametr na vymezeném intervalu podle struktury vstupních řetězců. Pokud za vstupní řetězce byly dosazovány jednoduché sloučeniny bez čísel a závorek pohyboval se parametr q na intervalu $< 0{,}9; 1)$, jako při práci se slovníkovými daty. Při použití složitých sloučenin, jejichž názvy obsahovaly různé číslice a pomocné závorky byl parametr na intervalu $< 0{,}8; 0{,}9)$.

4.4 Testovaná data

4.4.1 Registrované látky EU

Agentura ECHA je jedním z regulačních orgánu zabývajících se kontrolou a uplatňováním předpisu o chemických látkách v rámci Evropské unie. Prosazuje především bezpečné používání chemický látek [17].

Soubor disponuje přibližně 6 500 záznamy. Ukázka výsledků pro normální (viz Tabulka 4.4.1-1) a reverzní (viz Tabulka 4.4.1-2) zpracování řetězce:

Funkce	Max. Bucket	Q'	Porovnání Q'	I_a	I_{2a}
Shash	6	2,531	0,000	1,39	1,53
AP	5	2,564	0,034	1,40	1,55
Java	6	2,577	0,046	1,41	1,55
Rotační	5	2,590	0,060	1,41	1,56
ELF	6	2,599	0,069	1,42	1,57
DJB	6	2,600	0,069	1,41	1,56
SDBM	6	2,603	0,072	1,41	1,57
FNV	6	2,611	0,080	1,42	1,57
BKDR	6	2,625	0,094	1,42	1,57
DEK	9	2,907	0,376	1,48	1,69
Aditivní	8	2,968	0,437	1,52	1,74
XOR	65	70,325	67,795	34,17	40,02

Tabulka 4.4.1-1: Výsledky normální zpracování

Funkce	Max. Bucket	Q'	Porovnání Q'	I_a	I_{2a}
Java	5	2,550	-0,061	1,39	1,54
SDBM	5	2,584	-0,027	1,40	1,56
ELF	6	2,589	-0,022	1,41	1,56
DJB	5	2,599	-0,012	1,42	1,57
FNV	7	2,605	-0,006	1,41	1,57
Shash	7	2,611	0,000	1,41	1,56
AP	6	2,613	0,003	1,42	1,57
BKDR	5	2,626	0,016	1,42	1,58
Rotační	6	2,639	0,028	1,42	1,58
DEK	8	2,719	0,108	1,44	1,62
Aditivní	8	2,968	0,357	1,52	1,74
XOR	65	70,325	67,715	34,17	40,02

Tabulka 4.4.1-2: Výsledky reverzní zpracování

Tabulka 4.4.1-2 zobrazuje, že funkce Shash se díky použití reverzní strategie načítání řetězce propadla o pět míst. Ze sloupce Porovnání lze vidět, že rozdíly mezi Shash funkcí a funkcemi, které ji překonaly, jsou v řádech setin až tisícin

4.4.2 Databáze ChEBI

Zkratka ChEBI zastupuje jméno Chemical Entities of Biological Interest. Jedná se o volně dostupnou databázi, která je zaměřena na organické sloučeniny [18].

Databáze disponuje přibližně 38 000 záznamy. Ukázka výsledků pro normální (viz Tabulka 4.4.1-2) a reverzní (viz Tabulka 4.4.2-2) zpracování řetězce:

Funkce	Max. Bucket	Q'	Porovnání Q'	I_a	I_{2a}
Shash	5	2,357	0,000	1,32	1,44
FNV	6	2,381	0,025	1,32	1,45
AP	6	2,391	0,035	1,33	1,45
BKDR	6	2,393	0,036	1,33	1,45
DJB	6	2,393	0,036	1,33	1,45
SDBM	6	2,394	0,037	1,33	1,46
Java	7	2,396	0,039	1,33	1,46
ELF	6	2,408	0,052	1,33	1,46
Rotační	7	2,448	0,091	1,34	1,48
DEK	25	3,334	0,978	1,52	1,84
Aditivní	38	18,616	16,260	5,47	8,24
XOR	367	460,030	457,673	303,98	305,33

Tabulka 4.4.2-1: Výsledky normální zpracování

Funkce	Max. Bucket	Q'	Porovnání Q'	I_a	I_{2a}
DJB	6	2,3827	-0,0062	1,32	1,45
Shash	6	2,3890	0,0000	1,32	1,45
FNV	6	2,3891	0,0001	1,32	1,45
Java	6	2,3891	0,0001	1,33	1,45
SDBM	6	2,3923	0,0033	1,33	1,45
BKDR	6	2,3941	0,0052	1,33	1,46
AP	7	2,3944	0,0054	1,33	1,46
ELF	6	2,3963	0,0073	1,33	1,46
Rotační	11	2,4417	0,0527	1,34	1,47
DEK	23	3,0588	0,6698	1,46	1,73
Aditivní	38	18,6163	16,2274	5,47	8,24
XOR	367	460,0297	457,6407	303,98	305,33

Tabulka 4.4.2-2: Výsledky reverzního zpracování

4.4.3 Databáze NIST WebBook Chemie

Zkratka NIST je zkratkou pro National Institute of Standards and Technology a jedná se o Americkou databázi různých sloučenin. V rámci projektu EU byla přeložena pro Českou Republiku [19].

Databáze obsahuje přibližně 72 000 záznamů. Ukázka výsledků pro normální (viz Tabulka 4.4.3-1) a reverzní (viz Tabulka 4.4.3-2) zpracování řetězce:

Funkce	Max. Bucket	Q'	Porovnání Q'	I_a	I_{2a}
Shash	7	2,307	0,000	1,29	1,41
SDBM	6	2,324	0,017	1,30	1,42
AP	7	2,326	0,019	1,30	1,42
DJB	6	2,327	0,020	1,30	1,42
BKDR	6	2,328	0,021	1,30	1,42
Rotační	7	2,329	0,022	1,30	1,42
Java	6	2,332	0,025	1,30	1,42
FNV	7	2,340	0,033	1,31	1,43
ELF	77	2,537	0,230	1,31	1,49
DEK	70	3,286	0,979	1,38	1,74
Aditivní	38	23,153	20,846	9,48	12,10
XOR	2078	922,762	920,455	567,06	590,63

Tabulka 4.4.3-1: Výsledky normální zpracování

Funkce	Max. Bucket	Q'	Porovnání Q'	I_a	I_{2a}
SDBM	6	2,3214	-0,0095	1,30	1,42
DJB	6	2,3239	-0,0071	1,30	1,42
BKDR	6	2,3240	-0,0069	1,30	1,42
AP	6	2,3285	-0,0025	1,30	1,42
FNV	6	2,3307	-0,0002	1,30	1,42
Shash	5	2,3310	0,0000	1,30	1,42
Rotační	6	2,3362	0,0052	1,30	1,43
Java	7	2,3389	0,0080	1,30	1,43
ELF	28	2,3837	0,0527	1,31	1,44
DEK	15	2,5311	0,2002	1,34	1,51
Aditivní	38	23,1533	20,8224	9,48	12,10
XOR	2078	922,7618	920,4308	567,06	590,63

Tabulka 4.4.3-2: Výsledky reverzního zpracování

4.4.4 Databáze PDB

Zkratka PDB zastupuje název Protein Data Bank a jedná se o databázi krátkých proteinových sloučenin [20].

Databáze obsahuje přibližně 15 000 záznamů. Ukázka výsledků pro normální (viz Tabulka 4.4.4-1) a reverzní (viz Tabulka 4.4.4-2) zpracování řetězce:

Funkce	Max. Bucket	Q'	Porovnání Q'	I_a	I_{2a}
Shash	7	2,786	0,000	1,495	1,666
BKDR	6	2,818	0,032	1,509	1,684
FNV	7	2,834	0,048	1,507	1,687
Java	7	2,836	0,050	1,514	1,692
ELF	7	2,839	0,053	1,508	1,689
AP	6	2,839	0,053	1,508	1,690
Rotační	7	2,843	0,057	1,510	1,692
DJB	7	2,846	0,061	1,515	1,695
SDBM	7	2,851	0,065	1,515	1,697
DEK	7	2,978	0,192	1,548	1,753
Aditivní	8	3,636	0,850	1,815	2,098
XOR	145	173,147	170,361	114,383	114,906

Tabulka 4.4.4-1: Výsledky normální zpracování

Funkce	Max. Bucket	Q'	Porovnání Q'	I_a	I_{2a}
BKDR	7	2,812	-0,011	1,504	1,679
Shash	7	2,823	0,000	1,510	1,686
Java	8	2,837	0,014	1,506	1,688
DJB	6	2,849	0,026	1,514	1,696
AP	7	2,853	0,030	1,517	1,698
SDBM	7	2,854	0,031	1,520	1,701
ELF	7	2,855	0,032	1,511	1,696
Rotační	6	2,857	0,034	1,521	1,702
FNV	6	2,864	0,041	1,519	1,703
DEK	8	2,940	0,117	1,535	1,735
Aditivní	8	3,636	0,813	1,815	2,098
XOR	145	173,147	170,324	114,383	114,906

Tabulka 4.4.4-2: Výsledky reverzní načítání

4.4.5 Ekotoxikologická databáze

Jedná se o databázi sloučenin, která obsahuje toxikologická data popisující nebezpečnost vybraných látek. Dále označována jako EkoTox. Databáze byla zapůjčena RNDr. Pavlem Piskačem pro účely porovnávání hashovacích funkcí [21]. Tímto chci poděkovat za zapůjčení databáze.

Databáze obsahuje přibližně 200 000 záznamů. Jedná se o největší použitou databázi v rámci této bakalářské práce. Ukázka výsledků pro normální (viz Tabulka 4.4.5-1) a reverzní (viz Tabulka 4.4.5-2) zpracování řetězce:

Funkce	Max. Bucket	Q'	Porovnání Q'	I_a	I_{2a}
Shash	8	2,6699	0,0000	1,439	1,600
FNV	7	2,6728	0,0029	1,442	1,603
BKDR	8	2,6730	0,0031	1,442	1,603
AP	8	2,6752	0,0052	1,443	1,604
Java	6	2,6762	0,0063	1,443	1,604
DJB	7	2,6765	0,0065	1,443	1,604
Rotační	7	2,6774	0,0074	1,444	1,605
ELF	7	2,6790	0,0091	1,444	1,606
SDBM	7	2,6848	0,0148	1,446	1,609
DEK	25	3,0516	0,3817	1,524	1,761
Aditivní	85	49,5357	46,8657	12,419	20,251
XOR	1588	2046,2273	2043,5573	337,225	678,252

Tabulka 4.4.5-1: Výsledky normální zpracování

Funkce	Max. Bucket	Q'	Porovnání Q'	I_a	I_{2a}
Shash	6	2,6701	0,0000	1,441	1,601
Rotační	7	2,6729	0,0028	1,441	1,603
ELF	7	2,6733	0,0032	1,442	1,603
AP	8	2,6739	0,0038	1,441	1,603
SDBM	7	2,6773	0,0072	1,443	1,605
Java	7	2,6773	0,0073	1,442	1,605
BKDR	7	2,6789	0,0088	1,444	1,606
FNV	7	2,6803	0,0103	1,444	1,606
DJB	7	2,6814	0,0113	1,445	1,607
DEK	24	2,9329	0,2628	1,505	1,715
Aditivní	85	49,5357	46,8656	12,419	20,251
XOR	1588	2046,2273	2043,5572	337,225	678,252

Tabulka 4.4.5-2: Výsledky reverzní zpracování

4.5 Shrnutí výsledků

Z naměřených výsledků jsou patrné některé z vlastností hashovacích funkcí.

Aditivní a XOR hashovací funkce nejsou ovlivněny změnou strategie načítání vstupního řetězce. Ostatní hashovací funkce jsou ovlivněny změnou strategie. Z naměřených výsledků nelze určit, která strategie pro načítání chemických dat je

vhodnější, jelikož při změnách strategie načítání řetězců dochází k nepravidelným změnám na výstupních datech. Jednotlivé reakce funkcí na změnu strategie načítání vstupního řetězce jsou různé. Pro žádnou hashovací funkci nenastal případ, že při změně strategie došlo pouze ke zlepšení ve všech případech použití nebo naopak pouze ke zhoršení.

V rámci výsledků lze zjistit, že experimentální získávání parametru q pro Shash funkci je závislé, jak na povaze vstupních dat, tak i na způsobu jejich zpracování (strategie načítání vstupního řetězce).

Během porovnávání hashovacích funkcí při volbě strategie normálního načítání byla funkce Shash vždy nejlepší. Rozdíly mezi funkcemi, které se umístily v horních příčkách tabulky, jsou v řádech setin i menších řádů.

Při porovnávání vlastností hashovacích funkcí pro strategii reverzního načítání vstupního řetězce nelze určit, která hashovací funkce je nejlepší. Získané výsledky ukazují na Shash, protože při počtu funkcí, které se umístily do třetího místa, je její počet nejvyšší, viz Tabulka 4.4.5-1: Počet výskytů reverzní zpracování.

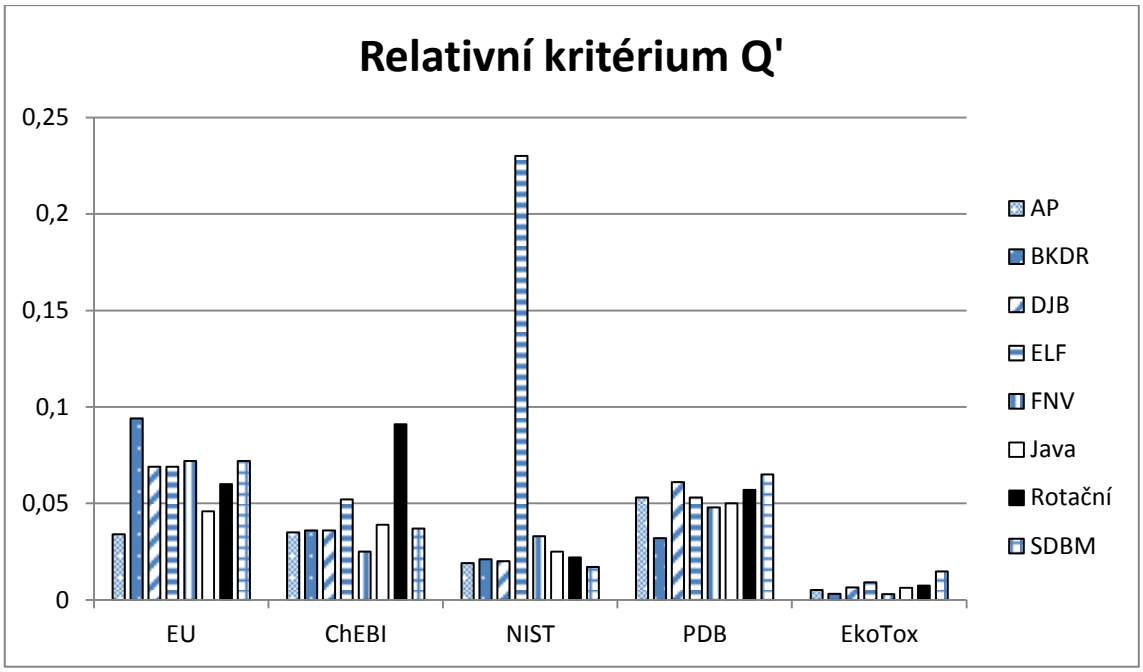
Funkce	Počet výskytů do 3. místa
Shash	3
BKDR	2
DJB	2
ELF	2
Java	2
SDBM	2
Rotační	1

Tabulka 4.4.5-1: Počet výskytů reverzní zpracování

Pro názornou představu, je porovnání vůči Shash funkci zobrazeno v následujících grafech. Z grafů byly vypuštěny následující funkce Aditivní, XOR a DEK hashovací funkce. Funkce z grafů byly vynechány záměrně pro jejich relativně špatně výsledky, aby nedocházelo ke zbytečnému zkreslení měřítka celého grafu.

4.5.1 Graf – normální zpracování řetězce

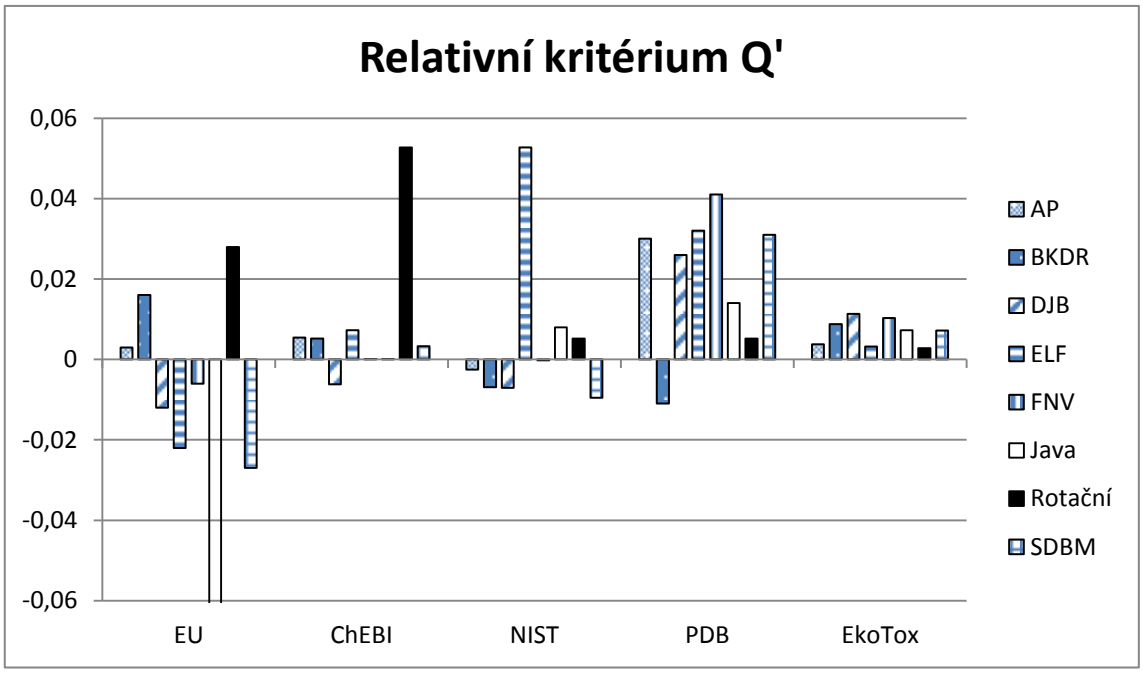
V následujícím grafu (viz Obrázek 4.5.1-1) jsou porovnávány hashovací funkce vůči Shash funkci. Kde hodnota 0 na ose y je v podstatě hodnota relativního kritéria Q' funkce Shash pro daný vstupní soubor. Z grafu vyplývá, že všechny uvedené funkce byly horší než Shash funkce. Ale rozdíl mezi funkcemi je v řádu desetin i v tom nejhorším případě.



Obrázek 4.5.1-1 Graf porovnání relativních kritérií vůči Shash pro normální zpracování řetězců

4.5.2 Graf – reverzní zpracování vstupního řetězce

V následujícím grafu (viz Obrázek 4.5.2-1) jsou porovnávány hashovací funkce vůči Shash funkci. Kde 0 na ose y je hodnota relativního kritéria Q' Shash funkce pro daný vstupní soubor. Hashovací funkce, které dosáhly záporných výsledků, jsou pro daný případ lepší než Shash funkce. A naopak funkce, které se umístily nad 0, jsou v daném případě horší než Shash funkce.



Obrázek 4.5.2-1Graf porovnání relativních kritérií vůči Shash pro reverzní zpracování řetězců

5 Závěr

Z výše uvedených výsledků vyplývá, že s rostoucí velikostí vstupního souboru (větší počet záznamů) a rostoucí velikostí hashovací tabulky se rozdíly v kritériu Q' mezi hashovacími funkcemi zmenšují. Tento jev je způsoben tím, že poměr rozložení prvků v hashovací tabulce v rámci dané hashovací funkce zůstává přibližně stejný, mění se pouze absolutní počet vkládaných prvků.

Jeden z překvapujících faktů je ten, že experimentální simulace parametru q je silně ovlivněna nejen podstatou dat vstupního řetězce, ale i způsobem jeho načítání. Rozdíly vzniklé v hodnotách kritéria Q' pro Shash funkci při změně strategie načítání byly v řádu desetin až setin. Reakce jednotlivých funkcí na změnu strategie byly různé, ale pro žádnou z používaných funkcí nenastal případ, že by jedna strategie zlepšila nebo naopak zhoršila všechny výsledky.

Z analýzy vlastností hashovacích funkcí v závislosti na textovém řetězci vyplývá, že vlastnosti hashovacích funkcí jsou silně ovlivněny jazykem i oborem, do kterého textový řetězec spadá.

Dále je patrné, že pro práci s předem známými daty (vzorkem dat) je vhodnější používat hashovací funkce s parametrem jako je Shash. Díky možné simulaci parametru q dosahovala tato funkce nejlepších výsledků. Pro neznámá data bych potom spíše volil některou z obecných funkcí, jako jsou BKDR, DJB nebo AP.

Přehled zkratek

Zkratky použitých databází

V následující tabulce jsou uvedeny všechny použité zkratky databázi, které jsou používány pro testování.

EU	Agentura ECHA registrované látky pro EU
ChEBI	Databáze organických sloučenin
NIST	WebBook Chemie – databáze přeložená v rámci projektu EU
PDB	Databáze proteinové banky
EkoTox	Ekotoxikologická databáze

Zkratky názvů hashovacích funkcí

V následující tabulce jsou uvedeny použité zkratky názvů z dokumentu.

Shash	Skala Václav, Hrádek Jan, Kuchař Martin hashovací funkce
BKDR	Brian Kernighan, Dennis Ritchie hashovací funkce
DEK	Donald E. Knuth funkce
AP	Arash Partow hashovací funkce
FNV	Glen Fowler, Landon Curt Noll, Phong Vo hashovací funkce
DJB	Dan J. Bernstein hashovací funkce

Zkratky používaných proměnných

V následující tabulce jsou uvedeny zkratky nejpoužívanějších proměnných.

Q'	Relativní kritérium
q	Parametr pro výpočet Shash funkce
I_a	Lineární průměr obsazenosti bucketů
I_{2a}	Kvadratický průměr obsazenosti bucketů

Literatura

[1] **Citriak, Peter.** *Hash funkce a jejich experimentální porovnání.* Bakalářská práce. Plzeň : Západočeská univerzita, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky, 2011. Vedoucí bakalářské práce prof. Ing. Václav Skala, CSc.

[2] **Mulvey, Bret.** Pluto Scarab - Hash Functions. *Hash Functions.* [Online] 2007. [Citace: 1. 4 2013.] http://home.comcast.net/~bretm/hash/.

[3] **Mička, Pavel.** Hashovací tabulka - Algoritmy.net. *Algoritmy.net.* [Online] 2008. [Citace: 1. 4 2013.] http://www.algoritmy.net/article/32077/Hashovaci-tabulka.

[4] **Skala, Václav, Hrádek, Jan a Kuchař, Martin.** *New Hash Function Construction for Textual and Geometric Data Retrieval.* Corfu : ASM 2010 conference, 2010. stránky 209-219. ISBN 978-960-474-213-4. [Online Draft] http://herakles.zcu.cz/~skala/publications.htm.

[5] **Havlíčková, Petra.** *Jazyk, jazykové skupiny.* [Online] 27. 1 2009. [Citace: 18. 4 2013.] http://www.havape.estranky.cz/clanky/literatura-pro-ctvrty-rocnik/jazyk_-jazykove-skupiny-_-rodiny_.html.

[6] **Sochrová, Marie.** *Český jazyk v kostce pro SŠ.* Praha : Nakladatelství Fragment, 2009. stránka 11. ISBN 978-80-253-0950-6.

[7] **Skala, Václav a Hrádek, Jan.** *Effecient Hash Function for Duplicate Elimination in Dictionaries.* Bratislava : Slovak University of Technology, 2009. stránky 382-391. ISBN 978-80-227-3032-7. [Online Draft] http://herakles.zcu.cz/~skala/publications.htm.

[8] **Kernighan, Brian a Ritchie, Dennis.** *The C Programming Language.* Prentice Hall, 2008. stránka 156. ISBN 0-13-110362-9.

[9] **Knuth, Donald Ervin.** *The Art Of Computer Programming: Sorting and Searching.* 2nd edition. Massachusetts : Addison-Wesley Profesional, 1998. ISBN 0-201-89685-0.

[10] **Partow, Arash.** *General Purpose Hash Function Algorithms.* [Online] [Citace: 8. 4 2013.] http://www.partow.net/programming/hashfunctions/.

[11] **Fowler, Glenn a Vo, Phong.** *FNV Hash.* [Online] 1994. [Citace: 8. 4 2013.] http://www.isthe.com/chongo/tech/comp/fnv/.

[12] **Walker, Julienne.** *Etternaly confluzzled.* [Online] 2008. [Citace: 18. 4 2013.] http://eternallyconfuzzled.com.

[13] **Engelschall, Ralf.** *PASTEBIN.* [Online] 2012. [Citace: 18. 4 2013.] http://pastebin.com/dDQ2kDkK.

[14] **Dzysyak, Sergiy.** Javascript hash functions to convert string into integer hash. *Erly Coder .com.* [Online] 2011. [Citace: 8. 4 2013.] http://erlycoder.com/49/javascript-hash-functions-to-convert-string-into-integer-hash-.

[15] **Guard, Damien.** Calculating Elf-32 in C# and .NET. *DamienG.* [Online] 2007. [Citace: 8. 4 2013.] http://damieng.com/blog/2007/11/24/calculating-elf-32-in-c-and-net.

[16] Úvod do C# a .NET frameworku. *devbook.cz.* [Online] 2013. [Citace: 15. 4 2013.] http://www.devbook.cz/c-sharp-tutorial-uvod-do-jazyka-a-dot-net-framework.

[17] **Evropská agentura pro chemické látky.** *ECHA.* [Online] European Chemicals Agency. [Citace: 18. 2 2013.] http://echa.europa.eu/.

[18] **de Matos, Paula a jiní.** ChEBI. *Chemical entities of biological interest.* [Online] 2009. [Citace: 18. 2 2013.] http://www.ebi.ac.uk/chebi/.

[19] **Matouš, Jaroslav a Šípek, Milan.** *NIST WebBook Chemie.* [Online] 2009. [Citace: 18. 2 2013.] http://webbook.nist.gov/chemistry/.

[20] **Feng, Zukang a jiní.** Ligand Expo. *Protien Data Bank.* [Online] 1. 9 2004. [Citace: 18. 2 2013.] http://ligand-expo.rcsb.org/.

[21] **Píškač, Pavel a Čermák, Vilém.** *Ekotoxikologická databáze.* [Online] 1996. [Citace: 20. 2 2013.] http://www.piskac.cz/ETD/Default.htm.

Přílohy

Seznam příloh: Uživatelská dokumentace

1 Uživatelská dokumentace

Obsah

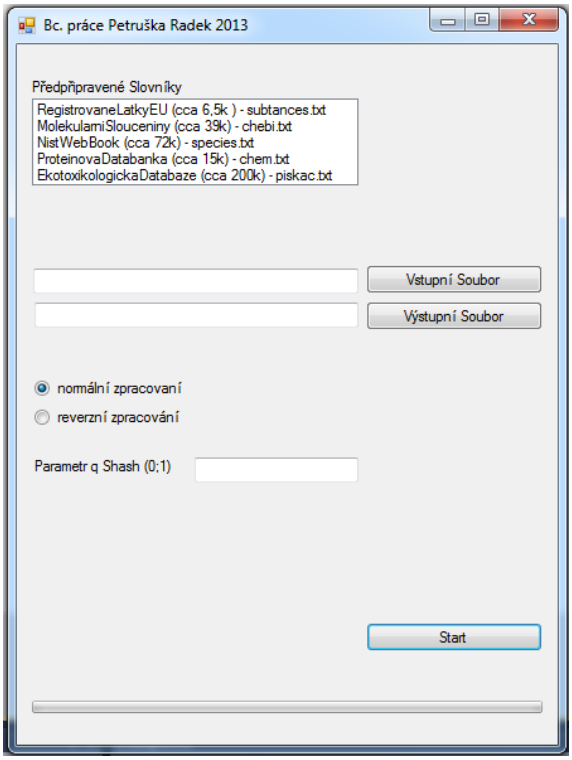
1.1 Obecný popis

Aplikace pro testování hashovacích funkcí je realizována pomocí jednoduchého formulářového zobrazení.

Vlastní aplikace byla testována na Windows 7 Professional s českou lokalizací spolu s balíkem knihoven Microsoft .NET Framework 4[3]. Kompatibilita s jinou verzí operačního systému (např. UNIX os) nebo jiným balíkem knihoven (starší verze knihovny Microsoft .NET) není zaručena.

1.2 Práce s aplikací

Vlastní aplikace disponuje hlavní oknem a několika dialogovými okny. Ukázka hlavního okna viz Obrázek 1.2-1 Hlavní okno. Hlavní okno aplikace obsahuje progress bar, který zobrazuje postup při zpracování vstupních dat, a několik tlačítek.

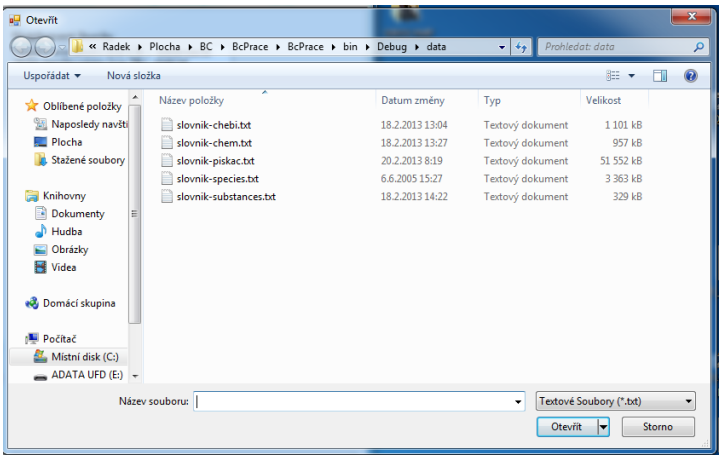


Obrázek 1.2-1 Hlavní okno

[3] balík knihoven je volně dostupný na http://www.microsoft.com nebo na CD ve složce software

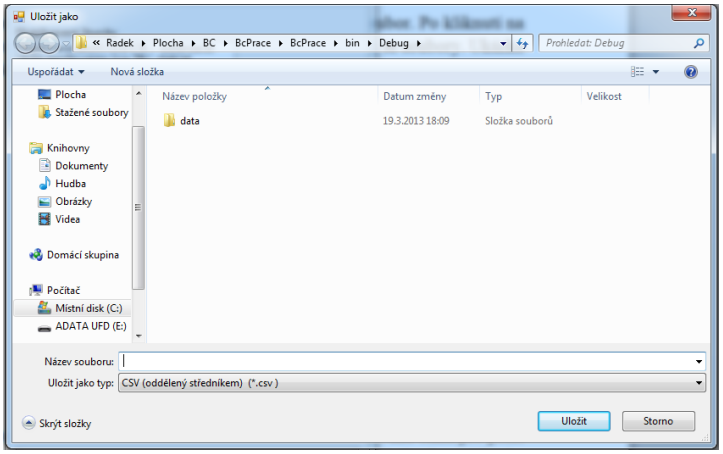
1.2.1 Práce s předpřipravenými slovníky

Při výběru předpřipraveného slovníku dojde k automatickému vyplnění polí vstupního souboru, výstupního souboru a parametru q. Pokud chcete s předpřipravenými slovníky experimentovat, musíte je načíst sami pomocí tlačítka Vstupní Soubor. Po kliknutí na tlačítko Vstupní Soubor se otevře klasické dialogové okno pro práci se soubory. Ukázka dialogového okna viz Obrázek 1.2-2.



Obrázek 1.2-2 Dialogové okno Vstupní Soubor

Umístění výstupního souboru vyberete pomocí dialogového okna pro práci se soubory, které se objeví po kliknutí na tlačítko Výstupní Soubor. Ukázka dialogového okna viz Obrázek 1.2-3.



Obrázek 1.2-3 Dialogové okno Výstupní Soubor

Přednastavené ukládání výstupních souborů v rámci předpřipravených slovníku je nastaveno do složky vlastní aplikace.

Pomocí zaškrtávacích políček (radion buttons) volíte strategii, která je použita při načítání vstupního řetězce. Tato strategie určuje, zda bude vstupní řetězec pro hashovací funkce načítán v normální nebo reverzním pořadí např. slovo *velryba:*

- normální pořadí – *velryba*
- reverzní pořadí – *abyrlev*.

1.2.2 Práce s obecnými slovníky

Při použití obecného slovníku je vyžadováno dodržení pravidel uvedených v podkapitole 1.2.4 Požadavky na vstupní soubor. Obecný vstupní soubor se načte pomocí dialogového okna Vstupní Soubor, viz Obrázek 1.2-2.

Umístění výstupního souboru se zvolí pomocí dialogového okna Výstupní Soubor, viz Obrázek 1.2-3.

Hodnota parametru q musí být z intervalu $(0; 1)$. Pro textová data bylo zjištěno, že se optimální parametr q nachází na intervalu $(0{,}8; 1)$. Parametr q zapisujte pomocí desetinné čárky.

1.2.3 Použité hashovací funkce

V aplikaci je porovnáváno celkem 12 hashovacích funkcí podrobný popis spolu s ukázkami zdrojových kódů naleznete v bakalářské práci. Zde je uveden pouze jejich seznam.

1. aditivní
2. XOR
3. rotační
4. Shash
5. BKDR
6. DEK
7. AP
8. FNV
9. DJB
10. JAVA
11. SDMB
12. ELF

1.2.4 Požadavky na vstupní soubor

Základní požadavky pro vstupní soubory:

1. znaková sada vstupní souboru musí být totožná s přednastavenou znakovou sadou operačního systému např. Windows 7 Professional CZE používají Windows - 1250 (CP - 1250)
2. struktura vstupního souboru musí splňovat následující podmínku: každý záznam (vstupní řetězec) je na samostatném řádku
3. každý řádek vstupního souboru je ukončen CR nebo CR+LF

1.2.5 Struktura výstupního souboru

Výstupní soubor je ve formátu CSV s přednastavenou znakovou sadou daného operačního systému. Začátek výstupního souboru je tvořen hlavičkou. Hlavička výstupního souboru obsahuje obecné informace o vstupních datech. Těmito informacemi jsou:

5. absolutní cesta ke vstupním datům např. C:\...\ slovnik-substances.txt
6. postup při načítání vstupního řetězce (normální / reverzní)
7. nejdelší vstupní vektor ze vstupních dat (nejdelší slovo)
8. počet nalezených duplicitních vstupních vektorů

Další část výstupního souboru tvoří vlastní tabulka, která obsahuje naměřené hodnoty, viz Tabulka 1.2.5-1. Definice a vzorce pro výpočet jednotlivých hodnot jsou podrobně popsány v bakalářské práci.

Název hashovací funkce	Maximální bucket	Kritérium Q	Relativní kritérium Q'	Lineární obsazenost bucketu I_a	Kvadratická obsazenost bucketu I_{a2}
Aditivní funkce	8	17743,5	2,967636729	1,522148676	1,73535478

Tabulka 1.2.5-1 Ukázka struktury výstupního souboru

Hodnoty uložené v tabulce jsou uloženy na výchozí počet desetinných míst, který je umožněn programovacím jazykem C#. Tento přístup byl zvolen pro maximální flexibilitu dat tak, aby výstupní data vyhovovala co nejvíce požadavkům. O zaokrouhlení výstupních dat potom rozhoduje ten, kdo je zpracovává.

1.2.6 Čtení výstupního souboru

Během testování bylo zjištěno, že starší verze sady Microsoft Office (verze 2000) špatně načítají výstupní soubor. Proto byl na CD přidán volně dostupný textový editor PSPAD[4]. Novější sady Microsoft Office s načtením CSV souboru neměly problém (testováno na verzi 2010).

[4] volně dostupný na http://www.pspad.com/cz/ nebo na CD ve složce software