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

Bakalářská práce

Hash funkce a jejich experimentální porovnání.

Plzeň, 2011 **Peter Citriak**

Prohlášení

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

V Plzni dne _____ _____

podpis

Abstrakt

Při práci s daty je častým problémem rychlé dohledávání dat (ověřování duplicit, hledání hodnot na základě vyhledávacího klíče atd.). Pro tyto účely se zdá být vhodnou volbou právě struktura nazývaná hashovací tabulka. Její vlastnosti závisí na vhodné volbě hashovací funkce.

Cílem této práce je navrhnout modulární systém pro získání dát potřebných pro porovnání vlastností různých hashovacích funkcí s ohledem na použití v hashovacích tabulkách. Na netriviálních datech bude ověřena funkcionalita a vlastnosti několika vybraných hashovacích funkcí.

Abstract

When working with data there is a common problem of fast data tracing (checking for data duplicates, search for values based on the "search key", etc.). For this purpose appears to be an appropriate choice to use a structure called a hash table. Its properties depend on appropriate choice of the hash function.

The main goal of this work is to design a modular system for gathering data needed for comparing properties of different hash functions with respect to the use of the functions in hash tables. The functionality and qualities of several selected hash functions will be verified on non-trivial data sets.

Obsah

Kapitola 1

Úvod

Při zpracování velkého počtu dat vznikají problémy související s rychlým dohledáváním ve zpracovaných datech. Částečně se tyto problémy dají řešit technologicky (vývojem rychlejších procesorů, zkracováním přístupové doby k paměti atd.), ale i algoritmicky (například sofistikovaným způsobem ukládání dat). Jedním ze způsobů ukládání dat, které zároveň urychlují následné vyhledávání, je používání tzv. hashovacích tabulek. Jejich vlastnosti se odvíjejí od správné volby hashovací funkce. Tato práce se bude věnovat porovnáním několika vybraných hashovacích funkcí. K tomuto účelu je nutné navrhnout jednoduchý modulární systém sloužící k získávání dat potřebných pro porovnání vlastností těchto funkcí. V teoretické části této práce budou vysvětleny základní pojmy související s danou problematikou. Samostatná část práce bude věnovaná analýze vlastností několika vybraných hashovacích funkcí, které se dají použít pro implementaci hashovací tabulky. V realizační části bude navrhnut modulární systém pro získávání informací o vlastnostech daných funkcí. Data získaná z tohoto systému budou použita pro porovnání vlastností vybraných hashovacích funkcí. Závěr této práce se bude věnovat především vyhodnocení získaných výsledků a celkovému zhodnocení vybraných hashovacích funkcí.

Kapitola 2

Teoretická část

2.1 Hashovací funkce

2.1.1 Definice

Existuje několik běžně používaných definic hashovacích funkcí. Jednotlivé definice se od sebe většinou odlišují tím, jak autor na zpracovaná data nahlíží. Všeobecně platí, že na datové struktury v počítačích je možné nahlížet jako na bitové sekvence. V daném případě je libovolná struktura jistým druhem bitového vektoru. S ohledem na danou skutečnost je možné použít následující definici hashovací funkce:

„Hashovací funkce jsou funkce, které mapují jeden bitový vektor na jiný bitový vektor, z pravidla kratší než je ten původní a obvykle s pevnou délkou bitového vektoru“[1].

Z matematického hlediska je to libovolná, dobře definovaná, matematická funkce $h(x)$, pro kterou platí:

$$\forall x \in M \exists y \in N : y = h(x)$$

kde M je množina mapovaných hodnot a N je množina výsledných hodnot (takzvaných hashů). Pokud se nejedná o speciální hashovací funkci (např. kryptografickou), je obvykle množina výsledných hodnot N konečná. V obecném případě může být množina hodnot M větší (má větší počet prvků) než množina N, a proto běžně není hashovací funkce prostá.

$$\exists x_1, x_2 \in M, x_1 \neq x_2 : h(x_1) = h(x_2)$$

Pokud by ale množina mapovaných hodnot měla menší nebo stejný počet různých prvků jako množina hodnot, je možné najít hashovací funkci, která bude zároveň i prostá.

$$\forall x_1, x_2 \in M : x_1 \neq x_2 \Leftrightarrow h(x_1) \neq h(x_2)$$

Takovéto funkci se říká „perfektní hashovací funkce“. Význam této funkce bude zřejmý po vysvětlení pojmu hashovací tabulka. V praxi však nalezení perfektní hashovací funkce, která by zároveň nevedla na nějaký složitý výpočet hashu, bývá velmi obtížné. Spíše se používají funkce, které se vlastnostmi perfektní hashovací funkci blíží.

2.1.2 Použití

Používání hashovacích funkcí pro různé účely se značně rozšířilo. I tak je možné rozdělit použití do dvou základních kategorií [1]:

1. „Fast table lookup“ – rychlé dohledávání dat.
2. „Encryption“ – kryptografie dat.

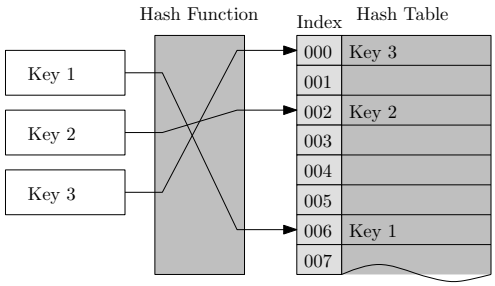
Daná dvě různá použití kladou zároveň důraz na jiné vlastnosti hashovacích funkcí. Kupříkladu u funkcí používaných v kryptografii je kladen veliký důraz na to, aby výsledná hodnota nebyla čitelná, aby drobná změna vstupního vektoru způsobila velkou změnu ve výsledném hashu a také aby bylo velmi obtížné najít dva různé vektory se stejným hashem. Co se týče funkcí spadajících do první kategorie, jejich vlastnosti budou blíže rozebrány po zavedení pojmu „hashovací tabulka“. Obecně mají tyto funkce vlastností generátorů náhodných čísel (požaduje se, aby výsledné hodnoty byly co nejrovnoměrněji rozložené v nějakém předem daném intervalu hodnot). Proto je možné porovnání hashovacích funkcí realizovat jako porovnaní funkcí generujících pseudonáhodná čísla (detaily o funkcech generujících náhodné čísla viz. [2]).

2.2 Hashovací tabulka

2.2.1 Definice

Hashovací tabulka je obecná datová struktura[3]. Její primární účel je rychlé vyhledávání položek. V ideálním případě je tato datová struktura složena ze dvou základních částí. Tou první částí je pole konstantní délky, ve kterém jsou položky nebo reference na dané položky. Říká se mu stejně jako celé datové struktuře a to „hashovací tabulka“ (viz. obrázek 2.1). Druhou částí je mapování klíčů na index v hashovací tabulce. To je realizováno vhodnou hashovací funkcí, která nám převede klíč na index v daném poli. Vyhledání

[1] Na stránkách Breta Mulveye [1] je uvedena ještě jedna kategorie (tzv. „message digest“). Po úvaze ovšem dospějeme k tomu, že se jedná o speciální případ kategorie „fast table lookup“, při kterém se používají hashovací funkce s vlastnostmi spíše spadajícími do kategorie kryptografických funkcí.

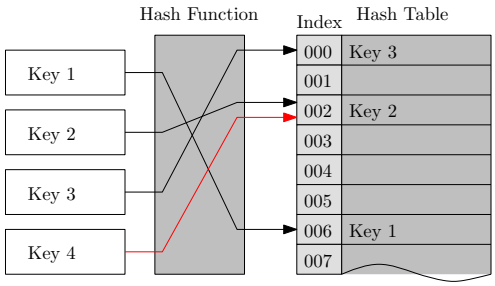


Obrázek 2.1: Hashovací tabulka.

položky v hashovací tabulce je pak velmi rychlá operace a její složitost (výpočetní náročnost) je závislá pouze na složitosti výpočtu indexu. Teoreticky se jedná o dohledávání v konstantním čase. To ale neplatí obecně. Vlastnosti hashovací tabulky jsou silně závislé na volbě hashovací funkce a také na volbě velikosti hashovací tabulky. Obsazenost tabulky je také často sledovaným parametrem. V literatuře se označuje pojmem "load factor" (f) a vyjadřuje poměr mezi počtem obsazených indexů a velikostí hashovací tabulky (H_SIZE).

2.2.2 Kolize

Protože universum klíčů může být mnohem větší než celkový počet indexů v daném poli, může se stát, že hashovací funkce vrátí pro dva různé klíče stejný index. Tomuto jevu se říká „kolize“. Nalezení hodnoty se pak zkom-



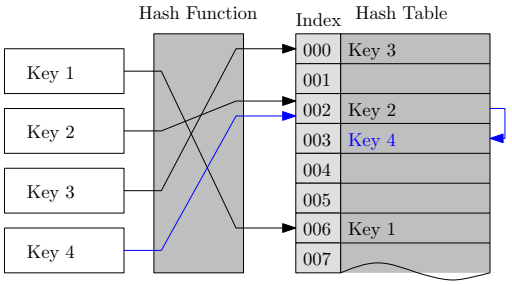
Obrázek 2.2: Kolize v hashovací tabulce.

plikuje a bude také záležet na zvolené strategii, kterou se kolize v hashovací tabulce bude řešit. Klíče se stejným indexem mohou pak být sdružované v takzvaných bucketech (řetězení), nebo se dopočte dodatečný index například za pomoci jiné hashovací funkce (otevřené adresování). Tyto dvě strategie patří mezi nejzákladnější[4].

2.2.3 Otevřené adresování

Název dané strategie vychází z toho, že hashovací funkce neurčuje přesnou adresu v hashovací tabulce. Pokud je index již obsazen jiným klíčem se stejným hashem, musí se nalézt nějaké volné místo, kam nový klíč umístit.

Základní strategie bývá takzvaný „linear probing“. V této strategii se kontroluje následující index v hashovací tabulce (viz. obrázek 2.3). Pokud je ná-



Obrázek 2.3: Linear probing.

sledující index volný, může se tam klíč vložit. Pokud ale volný není musí se nejprve zkontrolovat hash hodnoty na obsazeném indexu. Při shodě je ještě nutná kontrola shody klíčů. Pokud klíče nejsou shodné, je nutné daný postup opakovat, dokud se nenalezne stejný klíč, nebo není nalezen neobsazený index. To znamená, že je nutné zpracovávat i prvky, které vůbec nemusí mít stejný hash. Vyřazení prvků bývá také do jisté míry složité. Buď se vyřazený prvek nahradí speciálním prvkem označujícím, že daný index již nemá být používán[2], nebo se místo něj použije naposledy vkládaný prvek se stejným hashem[3]. V obou případech to vede ke zbytečným operacím, které se projeví zejména u téměř plně obsazených hashovacích tabulek. Tyto problémy se mohou také objevit při nevhodné volbě hashovací funkce, která klíče v hashovací tabulce nedistribuuje rovnoměrně. To vede ke vzniku takzvaných „shluků“.

Další strategii spadající do této skupiny je takzvaný „quadratic probing“ (viz. obrázek 2.4). Je podobná předešlé strategii, ale kolidující prvky nejsou obsazovány do hashovací tabulky následně za sebou. Nechť n je číslo vyjadřující počet neúspěšných porovnání klíčů v hashovací tabulce. Následující kontrolovaný index bude vyjádřen vztahem:

$$i_{n+1} = h(x) + n^2 \tag{2.1}$$

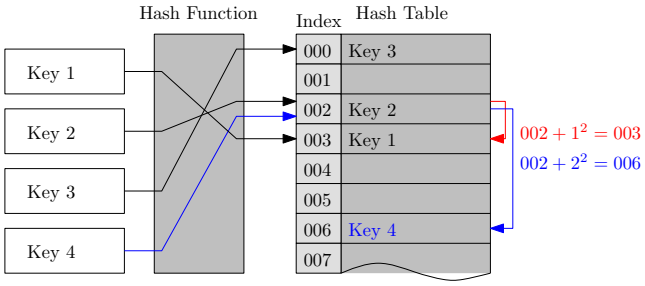
[2] Úplné vyřazení není možné, protože existuje možnost, že za vyřazeným prvkem existuje další prvek se stejným hashem a ten by již nebyl při prohledávání uvažován.

[3] To může v některých případech vést k úplnému prohledání celé hashovací tabulky.

Protože následující index může být větší než je velikost tabulky, je nutné daný vztah doplnit o modulární dělení velikostí tabulky, to znamená:

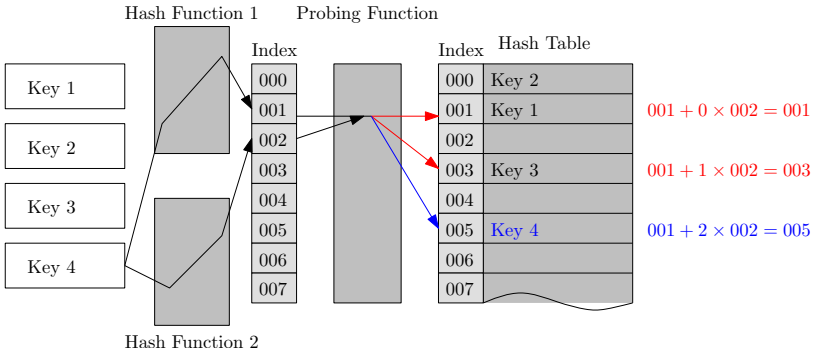
$$i_{n+1} = (h(x) + n^2) \div H\_SIZE \tag{2.2}$$

To jistě vede k lepší distribuci klíčů a menšímu počtu výskytů „shluků“.



Obrázek 2.4: Quadratic probing.

Co se týče zaplnění tabulky je situace ještě horší. Není zaručeno, že pro nový klíč je možné nalézt volné místo, když hashovací tabulka je napůl plná (nebo ještě dříve) pokud není velikost tabulky prvočíslo[4]. Problém výskytů



Obrázek 2.5: Double hashing.

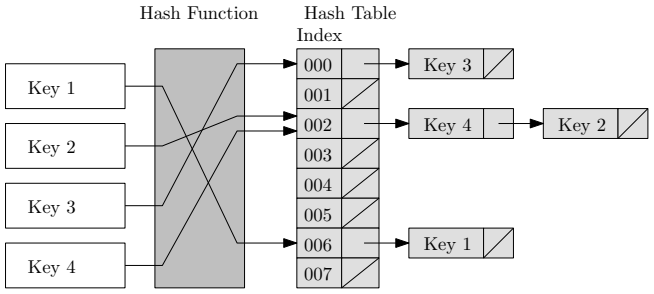
shluku řeší také strategie „double hashing“. Následující kontrolovaný index bude vycházet ze vztahu

$$i_{n+1} = (h_1(x) + n * h_2(x)) \div H\_SIZE \tag{2.3}$$

kde $h_1(x)$ a $h_2(x)$ jsou hashovací funkce. Je zřejmé, že $h2(x)$ nesmí vracet 0 pro žádné x. Ale ze stejného důvodu jako u „quadratic probe“ je také nutné, aby velikost tabulky byla prvočíslo. Jinak by snadno došlo k tomu, že nebude možné nalézt volný index, i když tabulka nebude plně obsazena.

2.2.4 Oddělené řetězení

Jedná se o alternativní metodu řešení kolizí k otevřenému adresování. Na rozdíl od „open adressing“ není tento způsob řešení kolizí omezen velikostí hashovací tabulky. Hashovací tabulka neobsahuje klíče, ale reference na klíče. Klíče se stejným hashem jsou sdružovány v takzvaném bucketu. Ten je implementován buď jako lineární spojový seznam, nebo jako vyváženy binární strom. Pokud dojde ke kolizi, prvek se přidá do spojového seznamu pokud se tam ještě nenachází (viz. obrázek 2.6). Tento způsob řešení kolizí odstra-



Obrázek 2.6: Řetězení.

ňuje některé nedostatky otevřeného adresování. Jestliže se v bucketu nacházejí jakékoliv prvky, není nutné opakovaně dopočítávat jejich hash. Z principu návrhu vyplývá, že všechny prvky v bucketu mají stejný hash. Nedostatkem zůstává časově náročná alokace paměti pro nově vkládaný prvek. U odděleného řetězení se doporučuje používat bucket ve formě lineárního spojového seznamu nebo seřazeného lineárního spojového seznamu. U neseřazeného spojového seznamu bude průměrná složitost dohledání prvku $\Theta(I_a)$, kde I_a je průměrná délka obsazených bucketů. V nejhorším případě to bude $O(I_m)$, kde I_m je maximální délka obsazených bucketů. Složitější struktury, jako například vyvážený binární vyhledávací strom, jsou doporučovány v případě, kdy load factor f je větší než 10 (to znamená, že v hashovací tabulce je 10 krát více prvků než je velikost hashovací tabulky), nebo když distribuce klíčů není rovnoměrná. Průměrná složitost dohledání prvku v obsazeném bucketu v daném případe bude $\Theta(log(I_a))$ a v nejhorším případě to bude $O(log(I_m))$.

2.3 Vlastnosti hashovacích funkcí v hashovacích tabulkách.

2.3.1 Obecné požadavky na hashovací funkce.

V ideálním případě by se při volbě hashovací tabulky mělo jednat o perfektní hashovací funkci, nebo funkci která distribuuje klíče v hashovací tabulce dokonale rovnoměrně. V praxi se však taková funkce hledá obtížně. Bez ohledu na to, jestli taková funkce existuje nebo ne, mohou nám vlastnosti dané funkce dobře posloužit jako porovnávací kritérium pro funkce existující.

Dobrá hashovací funkce by měla mít následující vlastnosti[3]:

1. Výpočet indexu by měla být velmi rýchlá operace.

2. Měla by obsahovat minimum kolizí.

První podmínka se ťežko posuzuje. Je závislá na architektuře procesorů, která vede k různým výpočetním rychlostem. Počet instrukcí a počet procesorových taktů jednotlivých instrukcí pro výpočet hashe za pomoci té samé hashovací funkce může být různý. To samé platí pro použitý kompilátor, který může nebo nemusí proces výpočtu optimalizovat. Problematikou výkonnosti hashovací funkce s ohledem na optimalizace překladu či použitého procesoru se zabývá například Paul Hsies [4]. Ve svém článku poukazuje na rozdíly několika vybraných hashovacích funkcí, které byly kompilovány a testovány na různých platformách.

Druhou podmínku je možné testovat programově. Mohou nastat dvě situace. Velikost hashovací tabulky H_SIZE je větší než počet vkládaných klíčů. V ideálním případě je každý klíč vložen na samostatný index, a proto průměrná délka obsazených bucketů je $I_a = 1$. Pokud je počet vkládaných klíčů větší než velikost hashovací tabulky, musí nutně dojít ke kolizím. Pro průměrnou délku obsazených bucketů bude v ideálním případě platit:

$$I_a = n/H\_SIZE \tag{2.4}$$

kde n je počet vložených klíčů.

[4] Článek je možné najít na stránkach `http://www.azillionmonkeys.com/qed/hash.html`

2.3.2 Hashovací funkce pro zpracování textů

Obecná hashovací funkce musí pracovat bez ohledu na to, jaká data přijdou na vstup dané funkce. Pro textove účely je ale možné předpokládat, že každý vstupní vektor je sekvence bitů sdružená po bytech (1 byte je 8 bitov) a že každý byte je z omezeného rozsahu reprezentovatelných čísel[5]. To znamená, že je víceméně přesně dán rozsah jednotlivých bytů, ale není určené, kolik bytů bude mít vstupní vektor. Obecně doporučený tvar hashovací funkce pro tyto učely má tvar [6]

$$h(x) = f(x) \bmod p \tag{2.5}$$

2.3.3 HF-Aditivní hash

Aditivní hashovací funkce je jednou ze základních hashovacích funkcí[3][8]. Každý byte daného řetězce se připočítává k výslednému hashu a na závěr se výsledný hash modulárně dělí vybraným prvočíslem. Pokud požadujeme, aby výsledek byl z nějakého rozsahu čísel, je vhodné nahradit dané prvočíslo požadovaným rozsahem. Volba prvočísla výchází z velikosti použité hashovací tabulky s otevřeným adresováním. Pokud jsou kolize řešené řetězením, není nutné, aby se jednalo o prvočíslo. Zdrojový kód přizpůsoben pro použití v programovacím jazyku C# můžeme vidět v následující části.

```
ulong prime = somePrimeNumber;
ulong additive_hash(string key)
{
  ulong hash=key.Length;
  for (int i=0; i<key.Length; ++i)
    hash += key[i];
  return (hash % prime);
}
```

Poznámka:

V základním kódování [6] obsadí jeden znak v paměti jeden byte. Návratová hodnota je ale ulong (u 32 bitových systemů jsou to 4 byty), a proto velká část návratové hodnoty zůstane nevyužita.

[5]Například použitelné (viditelné) znaky z ASCII tabulky začínají až od čísla 65.

[6]Například ASCII kódování.

2.3.4 HF-XOR hash

Opět jedna ze základních hashovacích funkcí. Pro výpočet hashu se používá bitový OR.

```
ulong prime = somePrimeNumber;
ulong xor_hash(string key)
{
  ulong hash=0;
  for (int i=0; i<key.Length; ++i)
    hash ^= key[i];
  return (hash % prime);
}
```

2.3.5 HF-Rotační (shift) hash

Poslední základní hashovací funkce je takzvaná rotační hashovací funkce. Pro výpočet hashů je použita takzvaná rotace bitů („left-shift“ nebo „right-shift“).

```
ulong prime = somePrimeNumber;
ulong rotating_hash(string key)
{
  ulong hash=(ulong)key.Len;
  for (int i=0; i<key.Len; ++i)
    hash = (hash<<4)^(hash>>28)^key[i];
  return (hash % prime);
}
```

2.3.6 HF-Václav Skala, Jan Hrádek, Martin Kuchař

Princip následující hashovací funkce byl popsán v dokumentu [6]. Výchází právě z předpokladů, že pracujeme s vektorem neznámé délky, ale známeho rozsahu použitých hodnot. Základem dané funkce je aditivní hash, který můžeme popsat následujícím vztahem:

$$h(x) = \left(\sum_{i=1}^{N} x_i \right) \bmod p \tag{2.6}$$

Aby bylo zabráněno „přetečení“, byl navrhnut následujíci vztah:

$$h(x) = \left(C * \sum_{i=1}^{N} q^i x_i \right) \bmod p \tag{2.7}$$

kde $q \in (0; 1)$ je koeficient zabezpečující konvergenci dané řady. Aby byl výsledný hash z rozsahu čísel 0 až h_{max}, je nutné dopočítat konstantu C. Nechť ϕ_i je největší možné písmeno na pozici i v množině hashovaných hodnot. Poté je možné udělat odhad maximální hash-hodnoty pomocí nasledujícího vztahu:

$$h_{max} = \left(C * \sum_{i=1}^{L} q^i \phi_i \right) \bmod p \tag{2.8}$$

Kde L je maximální délka řetězce z množiny hashovaných hodnot. Ve většině případů bude ϕ_i reprezentováno znakem „z“. Pokud je požadováno, aby hashovací hodnota byla reprezentována 64-bitovým číslem, bude $h_{max} = 2^{64} - 1$. Je ale nutné si uvědomit, že mantisa v čísle typu double má jenom 52 bitu a podle toho volit maximalní hodnotu h_{max}. Jinak může dojít naopak k velkému počtu kolizí. Z daného vztahu a po drobných úpravach je již jednoduché odvodit hledanou konstantu C (viz. [7]):

$$C = \frac{(2^{64} - 1) * (1 - q)}{L} \tag{2.9}$$

Koeficient q je nutné odvodit experimentálně. Ukazuje se, že i relativně krátka simulace před pevným nastavením hodnoty koeficientu q vede k dobrým výsledkům. Výslední zdrojový kód bude vypadat následovně:

```
double coefficientC = computedC;
double coefficientQ = simulatedQ;
ulong powerOf2SizeMask = poverOf2HashTableSize - 1;

ulong skala_hash(string key)
{
  double hash = 0;
  double qValue = coefficientC * coefficientQ;
  for (int i=0; i<key.Len; ++i)
  {
    hash += hashedKey[i] * qValue;
    qValue *= coefficientQ;
  }
  return
    (ulong)System.BitConverter.DoubleToInt64Bits(hash)
    & base.PowerOf2Mask;
}
```

2.3.7 HF-Brian Kernighan, Dennis Ritchie

Tato hashovací funkce byla navrhnuta Brianom Kernighanom a Dennisom Ritchiem v jejich známé knize „The C Programming Language“. Často krát je uvaděna v různých zdrojích pod zkratkou BKDR. Samotná implementace v jazyku C# bude vypadat následovně:

```
ulong prime = somePrimeNumber;
ulong seed = 131; //31 131 1313 13131 131313 etc..
ulong bkdr_hash(string key)
{
  ulong hash = 0;
  for (int i = 0; i < key.Length; i++)
    hash = (hash * seed) + key[i];
  return hash % prime;
}
```

2.3.8 HF-Donald E. Knuth

Donald Erwin Knuth ve své knize „The Art Of Computer Programming – Sorting And Searching“ navrhuje pro hashovací tabulky jako jednu z variant použití následovné hashovací funkce [7]:

```
ulong powerOf2SizeMask = poverOf2HashTableSize - 1;
ulong dek_hash(string key)
{
  ulong hash = (ulong)key.Length;
  for (int i = 0; i < key.Length; i++)
    hash = ((hash << 5) ^ (hash >> 27)) ^ key[i];
  return hash & powerOf2SizeMask;
}
```

2.3.9 HF-Arash Partow

Ve svem článku [9] analyzuje Arash Partow několik existujících hashovacích funkcí a na základě jejich vlastností vytváří vlastní variantu hashovací funkce pro textové účely. Samotný algoritmus vychází částečně z hashovací funkce uvedené v knize Donalda Erwina Knutha, avšak samotný výpočet mixovacích kroků se liší podle toho, zda je aktuální index v řetězci sudé nebo liché číslo. Algoritmus je možné vidět v následujícím kódu:

[7] Původní návrh předpokládal, že se bude jednat o 32 bitový hash, a proto je v posledním kroku algoritmus doplněn o logický součin s maskou velikosti hashovací tabulky.

```
ulong prime = somePrimeNumber;
ulong ap_hash(string key)
{
  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 % prime;
}
```

2.3.10 HF-Glen Fowler, Landon Curt Noll, Phong Vo

Jedna z nejčastěji používaných hashovacích funkcí, která vznikla na základě příspěvku Glena Fowlera a Phonga Vo z roku 1991 [10]. Její dobré vlastnosti se projevují zejména u rozlišení téměř identických textových řetězců. Proto je používána pravě v mailových serverech pro filtrování spamu, nebo v databázích pro indexaci. Implementace bude vypadat následovně:

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

ulong fnv_hash(string hashedKey)
{
  ulong hash = fnv_offset;
  for (int i = 0; i < hashedKey.Length; i++)
  {
    hash ^= hashedKey[i];
    hash *= fnv_prime;
  }
  return hash & base.PowerOf2Mask;
}
```

Na stránkach [10] je zdůrazněno, že fnv_prime bylo získano experimentálně a pro různé velikosti návratové hodnoty je vhodné volit jiné prvočíslo a jiný offset. Přehled těchto hodnot je teké uveden na citovaných stránkach.

2.3.11 HF-Dan J. Bernstein

Algoritmus byl navržen Danem Juliusem Bernsteinem a je považován za jeden z nejefektívnějších publikovaných algoritmů [9]. Zdrojový kod dané funkce bude vypadat následovně:

```
public ulong djb_hash(string key)
{
  ulong hash = 5381;
  //This was modified. The original algorithm is witten in C
  //and testing condition was on end of string.
  for (int i = 0; i < key.Length; i++)
    hash = ((hash << 5) + hash) + key[i]; /* hash * 33 + c */
  return hash;
}
```

Kapitola 3

Realizační část

3.1 Požadavky na testovací systém

Testovací aplikace musí být schopná testovat vybrané vlastnosti hashovacích funkcí použitých v hashovací tabulce s odděleným řetězením. Mezi testované vlastnosti bude patřit:

1. Lineární průměr obsazenosti bucketu
2. Kvadratický průměr obsazenosti bucketu
3. Relativní kritérium
4. Počty bucketů různých velikostí

Testovací aplikace bude schopná nasimulovat chování ideální hashovací funkce a jaké vlastnosti by měla tato funkce. Výsledky testování této funkce budou sloužit k porovnání vůči testovaným funkcím i to, jak se blíží k tomuto ideálnímu stavu.

Jednotlivé hashovací funkce nebudou součástí aplikace. Bude možné přidávat nové funkce bez nutnosti zásahu do kódu testované aplikace (pomocí samostatných modulů).

Zdroje dat, pomocí kterých se budou vlastnosti funkcí testovat, také nebudou součástí aplikace. To znamená, že zdroje dat bude možné volit samostatně.

Výsledky testů se budou ukládat do samostatných souborů tak, aby bylo možné jednoduše spravovat výsledky v jiných aplikacích (například v nějakém CSV[1] formátu souboru).

[1] Z anglického „Comma-Separated Values“

3.2 Návrh testovací aplikace

3.2.1 Volba programovacího jazyka

Většina dnes běžně používaných vyšších programovacích jazyků umožňuje relativně jednoduchý vývoj aplikací na základě objektového návrhu. Zároveň je součástí vývojových prostředí i množství připravených tříd pro zjednodušení práce (například pro práci se soubory či adresáři). Existuje několik běžně používaných způsobů, jak zabezpečit modularitu vyvíjené aplikace. Mezi nejznámější určitě patří následující tři koncepce:

1. COM objekty – Velmi rozšířený způsob, ale víceméně již na ústupu. Důvodem tohoto stavu je zřejmě to, že se musí striktně dodržovat metodika vývoje.

2. Scriptovací jazyky – Princip spočívá v tom, že kód je ukládaný jako textový řetězec (script) a ten se vyhodnocuje či aplikuje až za běhu aplikace (například Java-script, SQL dotazy, atd.).

3. Reflexe – Rozšířila se hlavně po vzniku koncepce psaní aplikací pro virtuální stroje (Java, .NET, Pyton). Využívá k tomu popisy tříd ukládané v tzv. metadatech.

Protože v našem případě nejde ani tak o rychlost jako o názornost a správnost výsledků, zvolíme si pro vyvíjenou aplikaci třetí metodu a to za použití programovacího jazyka C#.

3.2.2 Návrh a implementace

Celkové řešení testovacího modulu můžeme rozdělit do tří částí. Za prvé je to testovací aplikace, která bude mít na starosti právě vyhodnocování jednotlivých funkcí při použití v hashovací tabulce. Další částí jsou moduly obsahující testované hashovací funkce. Poslední částí je rozhraní, které bude definovat jednotný tvar testovacích funkcí.

3.2.3 Implementace rozhraní

Jelikož budeme testovat vlastnosti hashovacích funkcí použitých v hashovacích tabulkách, navrhneme nejprve strukturu samotné tabulky, která nám bude sloužit jako rozhraní pro implementaci hashovacích funkcí. V teoretické části jsme se seznámili se základními návrhy této struktury a s různými způsoby řešení kolizí. Každá hashovací tabulka se skládá ze dvou základních částí a to pole, ve kterém se budou položky nacházet a samotné hashovací

funkce. Jelikož v dané chvíli nevíme, jak bude hashovací funkce počítat výsledný hash, označíme metodu pro výpočet identifikátorem „virtual“. Tím umožníme dodatečné předefinování této metody[2]. Základní kostra hashovací tabulky bude následovná:

```
public class VHashTable
{
  private object[] _HashTable;
  public virtual ulong GetHash(string key)
  {
    throw
      new NotImplementedException("Hash method not defined!");
  }
}
```

Víme, že takto definovaná třída ještě není aplikovatelná. Zatím není definovaný objekt v hashovací tabulce. Ten je určitě závislý na zvolené strategii řešení kolize. V našem případě, kde je použito oddělené řetězení, by mohla položka hashovací tabulky vypadat následovně:

```
public class HashTableEntity
{
  public string Value; //No need for encapsulation now
  public HashTableEntity Next;

  public HashTableEntity(string value, HashTableEntity next)
  {
    this.Value = value;
    this.Next = next;
  }
}
```

Takto definovaná struktura nám zabezpečí vytvoření jednoduchého lineárního spojového seznamu. Pro optimalizaci vyhledávání položek je také možné vytvořit i složitější datové struktury jako například binární vyhledávací strom a udržovat daný strom ve vyváženém stavu. Tento postup se doporučuje v případě, že často dochází ke kolizím (v průměru 10 a více). Protože se v tuto chvíli neumíme rozhodnout zda má smysl použít strom[3], ponecháme tuto pomocnou strukturu a budeme používat lineární spojový seznam. Po úpravě bude naše třída vypadat následovně:

[2]Nebo můžeme dodefinováni těla metody vynutit vyhozením výjimky v těle definované virtuální metody.

[3]V danou chvíli nám není známo, jaké vlastnosti bude mít použitá hashovací funkce.

```
public class VHashTable
{
  private HashTableEntity[] _HashTable;
  public virtual ulong GetHash(string key)
  {
    throw
      new NotImplementedException("Hash method not defined!");
  }
}
```

Máme základní strukturu, která však nemá inicializované hodnoty. Nejprve je nutné určit velikost tabulky. Existují dvě možnosti. V prvním případě se určuje velikost hashovací tabulky jako nejbližší vyšší prvočíslo k předpokládané velikosti tabulky. Umístění v hashovací tabulce se určí modulárním dělením hashovacích hodnot. V druhém případě se určí velikost tabulky jako nejbližší vyšší mocnina dvou. Výhodou je pak nahrazení modulárního dělení operací bitového součinu (viz. například „HF-Václav Skala, Jan Hrádek, Martin Kuchař“). Jelikož nemůžeme předem určit, kterou z těchto dvou hodnot bude hashovací funkce využívat, bude inicializační metoda hashovací tabulky obsahovat parametr „předpokládaná velikost tabulky“ a budou se inicializovat obě hodnoty. Nejprve se určí nejbližší mocnina dvou a potom nejbližší prvočíslo k dané mocnině. Samotná velikost hashovací tabulky (pole entit) bude učená hodnotou prvočísla. Pro určení nejbližšího prvočísla k dané mocnině dvou vytvoříme pomocné pole, kde index bude určovat mocnitele a hodnota bude reprezentovat nejbližší prvočíslo. Implementace inicializační metody bude následovná:

```
public class VHashTable
{
  private static ulong[] _PrimeNumbers =
    new ulong[]
     {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};
  private HashTableEntity[] _HashTable;

  public ulong PowerOfTwoSize = 0;
  public ulong PrimeNumberSize = 0;

  public void InitTable(ulong preferedTableSize)
  {
```

```
    this.PrimeNumberSize = 0;
    this.PowerOfTwoSize = 1;
    double logValue = System.Math.Log(preferedTableSize, 2);
    int exponent = (int)System.Math.Ceiling(logValue);
    this.PowerOfTwoSize = (ulong)System.Math.Pow(2, exponent);
    this.PrimeNumberSize = _PrimeNumbers[exponent];
    _HashTable =
      new HashTableEntity[this.PrimeNumberSize];
  }

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

Některé hashovací funkce obsahují dopočitávané hodnoty závislé například na velikosti hashovací tabulky. Aby bylo možné tyto hodnoty inicializovat, vytvoříme dodatečnou virtuální inicializační metodu[4], která bude volaná na závěr inicializace hashovací tabulky.

```
public class VHashTable
{
  private static ulong[] _PrimeNumbers =
    new ulong[]
     {1, 2, 5, 11, 17, 37, 67, 131, 257, 521, 1031, 2053, 4099,
      8209, 16411, 32771, 65537, 131101, 262147, 524309,
      1048583, 2097169, 4194319, 8388617, 16777259, 33554467,
      67108879, 134217757, 268435459, 536870923, 1073741827,
      2147483693};
  private HashTableEntity[] _HashTable;

  public ulong PowerOfTwoSize = 0;
  public ulong PrimeNumberSize = 0;

  public void InitTable(ulong preferedTableSize)
  {
    this.PrimeNumberSize = 0;
    this.PowerOfTwoSize = 1;
    double logValue = System.Math.Log(preferedTableSize, 2);
    int exponent = (int)System.Math.Ceiling(logValue);
```

[4]Předefinování této metody není nutné, pokud nejsou inicializovány další proměnné.

```
    this.PowerOfTwoSize = (ulong)System.Math.Pow(2, exponent);
    this.PrimeNumberSize = _PrimeNumbers[exponent];
    _HashTable =
      new HashTableEntity[this.PrimeNumberSize];
    InitValues();
  }

  public virtual void InitValues()
  {
    ;//Nothing to do.
  }

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

Danou třídu musíme ještě doplnit o metodu pro přidávání nových položek. Nejprve určíme hashovací hodnotu přidávaného klíče. Pokud se v hashovací tabulce klíč nenachází, přidáme klíč na počátek lineárního spojového seznamu, jinak klíč nevkládáme. Viz následující kód[5]:

```
public class VHashTable
{
  ...
  public bool AddValue(string value)
  {
    ulong hashValue = GetHash(value);
    HashTableEntity current =
      _HashTable[hashValue];
    while (current != null)
    {
      if (current.Value == value)
        return false; //Already contains specified value.
      current = current.Next;
    }
    _HashTable[hashValue] =
      new HashTableEntity(value, _HashTable[hashValue]);
    return true;
  }
}
```

[5]Pro přehlednost vynecháme v kódu již implementované části

Tímto jsme vytvořili funkční hashovací tabulku. Ještě nám ale chybí metody pro určení testovaných vlastností naplněné hashovací tabulky. Pro dané vlastnosti vytvoříme veřejné proměnné a naplníme je metodou `CalculateComparsionValues`. Pro výpočet některých hodnot je nutné znát celkový počet položek v hashovací tabulce, a proto upravíme i metodu pro přidávání nových položek. Aby bylo jednotlivé implementace možné identifikovat, doplníme také virtuální jméno hashovací funkce, které je nutné v děděné třídě implementovat.

```
public class VHashTable
{
  ...

  public long ValueCount = 0;
  public double LinearAvarageBucketSize = 0;
  public double QuadraticAvarageBucketSize = 0;
  public double RelativeCriterion = 0;
  public Dictionary<long, long> BucketSizeDistribution =
    new Dictionary<long, long>();

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

  public void InitTable(ulong preferedTableSize)
  {
    this.ValueCount = 0;
    this.LinearAvarageBucketSize = 0;
    this.QuadraticAvarageBucketSize = 0;
    this.RelativeCriterion = 0;
    ...
  }

  public bool AddValue(string value)
  {
    ...
```

```
  _HashTable[hashValue] =
    new HashTableEntity(value, _HashTable[hashValue]);
  //increase the value count after adding new value
  this.ValueCount++;
  return true;
}

public void CalculateComparsionValues()
{
  long bucketSize = 0;
  long occupiedBucketCount = 0;
  //linear bucket size summary is same like element count
  long quadraticSummary = 0;
  long relativeSummary = 0;
  this.BucketSizeDistribution.Clear();
  for(ulong index = 0;
      index < this.PrimeNumberSize;
      index++)
  {
    bucketSize = GetBucketSize(index);
    quadraticSummary += bucketSize * bucketSize;
    if(!BucketSizeDistribution.ContainsKey(bucketSize))
      BucketSizeDistribution.Add(bucketSize, 0);
    BucketSizeDistribution[bucketSize]++;
    if (bucketSize > 0)
      occupiedBucketCount++;
  }
  foreach(KeyValuePair<long, long> dist
          in BucketSizeDistribution)
    relativeSummary += dist.Key * dist.Key * dist.Value;
  this.LinearAvarageBucketSize =
    (double)this.ValueCount
    /(double)occupiedBucketCount;
  this.QuadraticAvarageBucketSize =
    System.Math.Sqrt((double)quadraticSummary
    /(double)occupiedBucketCount);
  this.RelativeCriterion =
    (1.5d) * relativeSummary
    /(double)this.ValueCount;
}

private long GetBucketSize(long index)
{
```

```
    long result = 0;
    HashTableEntity current = _HashTable[index];
    while(current != null)
    {
        result++;
        current = current.Next;
    }
    return result;
  }
}
```

Tímto máme hotovu implementaci rozhraní, která splňuje požadavky na testovaná data.

3.2.4 Příklad použití rozhraní

Na příkladu hashovací funkce popsané v dokumentech [6] a [7] si ukážeme, jak postupovat při implementaci samotných hashovacích funkcí pro testovací účely. Vytvoříme si novou třídu, která dědí vlastnosti právě navrhnuté třídy `VHashTable`.

```
public class VSTestTable: VHashTable
{
}
```

Takto prázdná třída by ovšem způsobovala chyby při běhu testovací aplikace. Některé části kódu, které byly označeny jako virtuální, způsobují vyhození výjimky, a proto je nutné je implementovat. Mezi ně patří veřejná „property“ `HashFunctionName` a metoda `GetHash`. Protože implementovaná funkce musí mít inicializované některé proměnné, předefinujeme také metodu `InitData`. Výsledná implementace této funkce bude:

```
public class VSTestTable: VHashTable
{
  private double koeficientC = 0;
  private double koeficientQ = 0;

  public override string HashFunctionName
  {
    get
    {
      return "Vaclav Skala";
    }
  }
```

```
  public override void InitValues()
  {
    //max word length (approx)
    int maxWordLength = 28;
    //experimentaly found value for english dictionary
    this.koeficientQ = 0.20000000003453966d;
    this.koeficientC =
      (System.Math.Pow(2, 64) - 1)
      *(1 - this.koeficientQ)
      /maxWordLength;
  }

  public override ulong GetHash(string hashedKey)
  {
    double hash = 0;
    double qValue = koeficientC;
    for (int i = 0; i < hashedKey.Length; i++)
    {
      hash += hashedKey[i] * qValue;
      qValue *= koeficientQ;
    }
    return
      (ulong)System.BitConverter.DoubleToInt64Bits(hash)
      &(base.PowerOfTwoSize - 1);
  }
}
```

Z předešlého příkladu je vidět, že implementace nové funkce pro testování je jednoduchá, zejména když není nutná inicializace dat. Je vhodné testovací třídy implementovat zvlášť do samostatných knihoven. Důvod bude zřejmý v další části implementace.

3.2.5 Implementace testovací aplikace

Jak jsme již na začátku návrhu zdůraznili, celá implementace se dá rozdělit do tří částí. Jak bude vypadat rozhraní a jak se mají implementovat jednotlivé testované funkce jsme již ukázali. Zbýva ještě návrh a implementace testovací aplikace. Protože aplikace má fungovat jako modulární, musíme navrhnout, jak se budou jednotlivé testované moduly a testovací data načítat a jak a kam se budou výsledky testu ukládat. Tento problém se dá vhodně řešit použitím konfiguračního souboru. Testovací aplikace implmentujeme jako konzolovou aplikaci. Na základě konfiguračního souboru načteme

jednotlivé moduly a slovníky a do samostatných vystupních souborů uložíme výsledky testu. Pro práci s konfiguračními soubory poskytuje vyvojové prostředí *.Net* hned několik tříd. Pro jednoduchost použijeme třídu `ConfigurationManager` a její statickou property `AppSettings`. Ta nám dovolí jednoduše přistupovat do konfiguračního souboru aplikace a získavat hodnoty na základě klíčů. Aby bylo možné odlišit, zda se jedná o modul obsahující testované funkce, nebo se jedná o slovník, zavedeme jmennou konvenci pro klíče v konfiguračním souboru. Pro moduly budeme užívat jména klíču ve formátu „TestedDll_#“, kde # nahradíme pořadovým číslem. Hodnotou takového záznamu bude relativní adresa k testovanému modulu. Příklad těla konfigurace s jedním modulem bude:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="TestedDll_1" value="HashFunctionTest.dll"/>
  </appSettings>
</configuration>
```

Pro soubor s testovanými hodnotami zase zavedeme konvenci klíču ve formátu „TestedKeyCollection_#“, kde # nahradíme pořadovým číslem. Výsledný konfigurační soubor bude vypadat následovně:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="TestedDll_1" value="HashFunctionTest.dll"/>
    <add key="TestedKeyCollection_1" value="en.txt"/>
  </appSettings>
</configuration>
```

Teď si ukážeme, jak budeme načítat jednotlivé soubory za pomoci konfiguračního souboru. Do hlavní třídy nové konzolové aplikace implementujeme dvě statické metody pro načtení testovaných slovníků a modulů. Načtení slovníku bude relativně jednoduché. Využijeme třídy `ConfigurationManager`, která má v sobě definovanou kolekci pro snadný přístup k sekci `<appSettings>`. Zkontrolujeme všechny klíče, a pokud bude klíč obsahovat řetězec „TestedKeyCollection“, pokusíme se načíst data ze souboru definovaném pro daný klíč. Nejprve si ovšem zadefinujeme metodu pro načtení informací o souboru na základě konfiguračních hodnot. Pokud není soubor definován, vráti metóda hodnotu false. Viz následující kód:

```
class Program
```

```
{
  static void Main(string[] args) { ... }

  private static bool TryLoadFileInfo(
    string appSettingKey,
    string identification,
    out string fileName,
    out string filePath)
  {
    filePath = string.Empty;
    fileName = string.Empty;
    if (!appSettingKey.ToUpper().Contains(
           identification.ToUpper()
       ))
      return false;
    fileName =
      ConfigurationManager.AppSettings[appSettingKey];
    if (string.IsNullOrEmpty(fileName)||
      !File.Exists(fileName))
      return false;
    filePath = Path.GetFullPath(fileName);
    return true;
  }
}
```

Pro množinu klíčů použijeme generickou třidu `List<T>`. Budeme předpokládat, že v množině nejsou žádné duplicity. Je možné upravit načítaní s kontrolou na duplicity, ale je to za použití této třídy zbytečně časově náročná operace[6]. Vytvoříme také pomocnou metodu pro načtení cesty k souboru.

[6]Proto je kontrola v kódě sice naznačena, ale zakomentovaná.

```
class Program
{
  static void Main(string[] args) { ... }
  private static bool TryLoadFileInfo(...) { ... }

  private static
    Dictionary<string, List<string>> LoadDictionaries()
  {
    Dictionary<string, List<string>> result =
      new Dictionary<string, List<string>>();
    string filePath;
    string fileName;
    foreach (string settingKey
      in ConfigurationManager.AppSettings.Keys)
    {
      if (!TryLoadFileInfo(settingKey,
                           "TestedKeyCollection",
                            out fileName,
                            out filePath))
        continue;
      if (result.ContainsKey(fileName))
        continue;
      Console.WriteLine(string.Format(
           "Loading key collection from file: {0}"
           ,fileName
        ));
      result.Add(fileName, new List<string>());
      StreamReader file =
        new StreamReader(filePath);
      string line = null;
      int maxlength = 0;
      while ((line = file.ReadLine()) != null)
      {
        //if (!result[fileName].Contains(line))
        result[fileName].Add(line);
        if (line.Length > maxlength)
        maxlength = line.Length;
      }
      Console.WriteLine("Key collection loaded");
    }
    return result;
  }
}
```

Obdobním způsobem vytvoříme metodu pro načtení modulů, tentokrát ale využijeme reflexe pro vytvoření instancí testovaných tříd. Kód dané metody bude vypadat následovně:

```
private static List<VHashTable> LoadTestingClasses()
{
  List<VHashTable> result = new List<VHashTable>();
  Type hashTableType = typeof(VHashTable);
  string fileName;
  string filePath;
  foreach (string settingKey
    in ConfigurationManager.AppSettings.Keys)
  {
    if (!TryLoadFileInfo(settingKey,
                         "TestedDll",
                         out fileName,
                         out filePath))
      continue;
    Assembly dllAssembly = null;
    Console.WriteLine(
      string.Format(
         "Loading test modules from file: {0}",
         fileName));
    try
    {
      dllAssembly = Assembly.LoadFile(filePath);
    }
    catch (Exception exc)
    {
      Console.WriteLine(
        string.Format(
          "Unable to load the test modules!\r\n {0}",
          exc.Message));
        continue;
    }
    foreach (Module module in dllAssembly.GetModules())
    {
      foreach (Type type in module.GetTypes())
      {
        if (!hashTableType.IsAssignableFrom(type))
          continue;
        ConstructorInfo defaultConstructor =
          type.GetConstructor(Type.EmptyTypes);
        if (defaultConstructor == null)
```

```
        {
          ConstructorInfo[] constructors =
            type.GetConstructors();
          if (constructors == null || constructors.Length < 1)
            continue; //this type has no public constructor!
          defaultConstructor = constructors[0];
        }
        try
        {
          VHashTable newHashFunctionTestClass =
            (VHashTable)defaultConstructor.Invoke(
                                new object[] { }
                               );
          result.Add(newHashFunctionTestClass);
        }
        catch (Exception exc)
        {
          //could not create instance of this module type!
        }
      }
    }
    Console.WriteLine("Modules loaded!");
  }
  return result;
}
```

Pro každý načtený slovník budeme testovat všechny testovací tabulky, abychom získali data pro porovnání vlastností hashovacích funkcí. Protože nás bude zajímat také jak se bude chovat hashovací funkce v případě, že hashovací tabulka nemá dostatečný počet položek[7], budeme testy opakovat vícekrát a při každé iteraci zmenšíme inicializační velikost tabulky o polovinu. Pro přehlednost kódu si vytvoříme samostatnou funkci, která bude jednotlivé slovníky testovat a vytvářet adekvátní reporty. Vytvářet se boudou dva typy reportů. Prvním bude obecný report obsahující data pro porovnání hashovacích funkcí. Druhý typ výstupu bude statistika obsazenosti bucketů jednotlivých testovacích tabulek. Ten se bude generovat a ukládat pro každou hashovací tabulku zvlášť. Názvy souborů jednotlivých reportů budou generovány následovně:

- Obecní report – `_generalReport_<dictionaryName>_<tableSize>.txt`

[7] Velikost hashovací tabulky bude menší než počet vkládaných hodnot.

- Report obsazenosti – _specificReport_<dictionaryName>_<tableSize>_<hfName>.txt

, kde <dictionaryName> je jméno slovníku, <tableSize> je inicializační velikost tabulky a <hfName> je jméno hashovací funkce použité v hashovací tabulce. Kód implementované metody CreateReports je možné vidět na následujících řádcích.

```
private static string comparsionFileRowPattern =
  "{0}\t{1}\t{2}\t{3}\t{4}\t{5}\t{6}\r\n";
private static string bucketSizeStatisticsRowPattern =
  "{0}\t{1}\r\n";
private static void CreateReports(
  string dictName,
  List<string> valueCollection,
  List<VHashTable> hashTableCollection,
  ulong preferedHashTableSize
)
{
  string generalReportFileName =
    "_generalReport" +
    "_" + dictName +
    "_" + preferedHashTableSize.ToString() +
    ".txt";
  StreamWriter generalReport =
    new StreamWriter(generalReportFileName, false);
  //create report header
  generalReport.Write(comparsionFileRowPattern,
    "Hash Function Name",
    "Value Count",
    "Prime Number Size",
    "Power Of Two Size",
    "Linear Avarage Bucket Size",
    "Quadratic Avarage Bucket Size",
    "Relative Criterion");
  foreach (VHashTable testedTable
    in hashTableCollection)
  {
    testedTable.InitTable(preferedHashTableSize);
    foreach (string value in valueCollection)
      testedTable.AddValue(value);
    testedTable.CalculateComparsionValues();
    generalReport.Write(comparsionFileRowPattern,
      testedTable.HashFunctionName,
```

```
      testedTable.ValueCount,
      testedTable.PrimeNumberSize,
      testedTable.PowerOfTwoSize,
      testedTable.LinearAvarageBucketSize,
      testedTable.QuadraticAvarageBucketSize,
      testedTable.RelativeCriterion);
    string specificReportFileName =
      "_specificReport" +
      "_" + dictName +
      "_" + preferedHashTableSize.ToString() +
      "_" + testedTable.HashFunctionName +
      ".txt";
    StreamWriter specificReport =
      new StreamWriter(specificReportFileName, false);
    specificReport.WriteLine(
      "Bucket size statistics for " +
      testedTable.HashFunctionName + "\r\n" +
      "Prime number size = " +
      testedTable.PrimeNumberSize + "\r\n" +
      "Power of 2 size = " +
      testedTable.PowerOfTwoSize);
    specificReport.Write(bucketSizeStatisticsRowPattern,
      "Bucket Size",
      "Bucket Count");
    foreach (KeyValuePair<long, long> distribution
      in testedTable.BucketSizeDistribution)
      specificReport.Write(bucketSizeStatisticsRowPattern,
        distribution.Key,
        distribution.Value);
    specificReport.Close();
  }
  generalReport.Close();
}
```

Na závěr doplníme tělo hlavní metody o kód, kterým budeme pro jednotlivé testované slovníky provádět testy na načtených hashovacích funkcích.

```
class Program
{
  static void Main(string[] args)
  {
    List<VHashTable> testedHashFunctions =
      LoadTestingClasses();
    Dictionary<string, List<string>> testedDictionaries =
```

```
    LoadDictionaries();
  if (testedHashFunctions.Count == 0)
  {
    Console.WriteLine("There is no hash funkcion" +
      " for testing in defined dll files!");
    return;
  }
  if (testedDictionaries.Count == 0)
  {
    Console.WriteLine("There are no testing data" +
      " defined to run the test!");
    return;
  }
  ulong preferdSize = 0;
  foreach (
    KeyValuePair<string, List<string>> testedData
    in testedDictionaries
  )
  {
    if (testedData.Value.Count < 16)
    {
      Console.WriteLine("Tested data in " + testedData.Key +
        " contain less than 16 values. This test is ignored!");
      continue;
    }
    preferdSize = (ulong)testedData.Value.Count;
    for (int i = 0; i < 4; i++)
    {
      CreateReports(
        testedData.Key,
        testedData.Value,
        testedHashFunctions,
        preferdSize
      );
      preferdSize = preferdSize / 2;
    }
  }
}
private static bool TryLoadFileInfo(...) { ... }
private static List<VHashTable> LoadTestingClasses()
{...}
private static
  Dictionary<string, List<string>> LoadDictionaries()
```

```
  {...}
  private static void CreateReports(...) {...}
}
```

Tímto jsme realizovali implementační část úlohy. Zbývá nám danou aplikaci použít na reálná data a výsledky testu zhodnotit.

3.3 Interpretace výsledků získaných z testování

3.3.1 Testovaná data

V teoretické části tohoto dokumentu bylo popsáno několik hashovacích funkcí. Úroveň jejich využití je různá. Například aditivní hashovací funkce se nepoužívá vůbec kvůli jejím špatným vlastnostem. Naopak hashovací funkce FNV[8] je často využívána například pro indexaci v databázích. Pro naše porovnání naimplementujeme všechny hashovací funkce zmíněné v teoretické části, vyjma rotační hashovací funkce[9]. Pro porovnání s ideálním případem implementujeme také testovací hashovací tabulku, která bude simulovat ideální rozložení. To zabezpečíme pseudo-hashovací funkcí, která bude vracet lineárně se zvyšující index. Z návrhu společného rozhraní plyne, že testovanými hodnotami budou následující hodnoty:

- Průměrná délka bucketů – tento parametr určuje průměrné zaplnění bucketů a pro nás poslouží zejména pro hrubý odhad toho, které funkce jsou použitelné a které vůbec nevyhovují pro použití v praxi.
- Kvadratický průměr délky bucketů – obdobně jako v předešlém případě, a však u kvadratického průměru dochází k lepšímu odlišení podobných funkcí.
- Relativní kritérium – metoda porovnání pomocí relativního kritéria vychází z dokumentu [7]. Zahrnuje v sobě i vliv vyhledávaní položek když dojde ke kolizi, a proto je velmi vhodné pro porovnání vlastností hashovacích funkcí.

Hashovací funkce budeme testovat vůči anglickému a českému slovníku[10]. Obecně je ale možné test rozšířit o další slovníky nebo dokonce o kombinace slovníků. Kombinace slovníků by mohly sloužit například pro situace, kdy

[8] Zkratka vychází z jmen autorů Glen Fowler, Landon Curt Noll a Phong Vo.

[9] Principielně se shoduje s funkcí navrženou Donaldem Ervinem Knuthem

[10] Slovníky byli staženy ze stránek OpenOffice.org [11].

se pracuje s více jazyky současně (e-mailová komunikace v mezinárodním call-centru).

3.3.2 Získane výsledky

V tabulce 3.1 vidíme výsledky získané při testování hashovacích funkcí vůči anglickému slovníku. Jak jsme již upozorňovali v předchozí kapitole, budeme

Hash Function Name	Linear Avarage Bucket Size	Quadratic Avarage Bucket Size	Relative Criterion
Ideal Case	1	1	1.5
Vaclav Skala (en)	1.38503666	1.528667648	2.530790173
Donald Ervin Knuth	1.389068972	1.534572664	2.542976599
Dan Julius Bernstein	1.393880071	1.540867745	2.555033383
Kernighan-Ritchie	1.394215996	1.541502955	2.556524276
Fowler-Noll-Vo	1.397415381	1.547100622	2.569229273
Arash Partow	1.400756659	1.550199159	2.573377844
Aditive hash	26.13269339	39.49256768	89.52365982
XOR hash	216.2663551	317.7011248	700.067317
Počet položek ve slovníku			46281
Požadovaná velikost hashovací tabulky			46281
Inicializovaná velikost hashovací tabulky (mocnina dvou)			65536
Inicializovaná velikost hashovací tabulky (prvočíslo)			65537

Tabulka 3.1: Data získaná pro anglický slovník.

průměrnou velikost hashovací tabulky, využívat k základnímu rozdělení hashovacích funkcí na použitelné a nepoužitelné. Z tabulky je na první pohled zřejmé, že aditivní hashovací funkce a XOR hashovací funkce jsou pro hashovací tabulky naprosto nepoužitelné. Z tohoto důvodu je při dalších simulacích již nebudeme brát v potaz. U ostatních hashovacích funkcí vycházejí průměrné velikosti bucketů přibližně stejně. Při návrhu jsme požadovali, aby simulace proběhla i pro zmenšující se velikost hashovací tabulky. Výsledky daných simulací můžeme vidět v tabulkách 3.2, 3.3 a 3.4.

Hash Function Name	Linear Avarage Bucket Size	Quadratic Avarage Bucket Size	Relative Criterion
Ideal Case	1.412384033	1.495711235	2.375931808
Vaclav Skala (en)	1.859272055	2.107574168	3.583554807
Dan Julius Bernstein	1.87455952	2.123576974	3.608511052
Fowler-Noll-Vo	1.868504986	2.125242076	3.625883192
Arash Partow	1.870166081	2.126908614	3.628346406
Kernighan-Ritchie	1.870090512	2.127188592	3.62944837
Donald Ervin Knuth	1.874407679	2.132797097	3.640208725
Počet položek ve slovníku			46281
Požadovaná velikost hashovací tabulky			23140
Inicializovaná velikost hashovací tabulky (mocnina dvou)			32768
Inicializovaná velikost hashovací tabulky (prvočíslo)			32771

Tabulka 3.2: Data získaná pro anglický slovník.

Hash Function Name	Linear Avarage Bucket Size	Quadratic Avarage Bucket Size	Relative Criterion
Ideal Case	2.824768066	2.850235136	4.313897712
Vaclav Skala (en)	2.98779858	3.364522687	5.683120503
Arash Partow	3.002530167	3.385095059	5.72460621
Dan Julius Bernstein	2.998250842	3.385228112	5.733227458
Kernighan-Ritchie	3	3.389315131	5.743728528
Fowler-Noll-Vo	3.005650084	3.394935602	5.751960848
Donald Ervin Knuth	3.010146341	3.40097547	5.763823167
Počet položek ve slovníku			46281
Požadovaná velikost hashovací tabulky			11570
Inicializovaná velikost hashovací tabulky (mocnina dvou)			16384
Inicializovaná velikost hashovací tabulky (prvočíslo)			16411

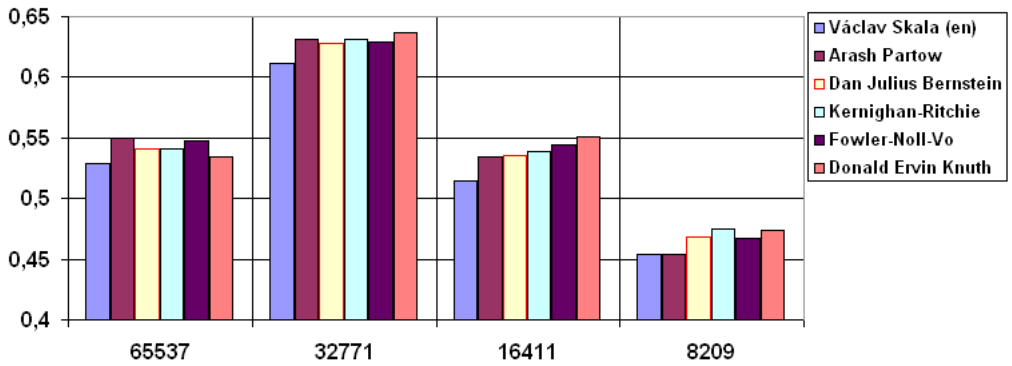
Tabulka 3.3: Data získaná pro anglický slovník.

Bližším zkoumáním hodnot zjistíme, že různé hashovací funkce reagují na změnu velikosti hashovací tabulky (bez změny vstupních dat) různě. Abychom danou skutečnost vyjádřili názorně, vytvořili jsme grafy kvadratických průměrů a relativních kritérií závislých na velikosti tabulky. Protože

Hash Function Name	Linear Avarage Bucket Size	Quadratic Avarage Bucket Size	Relative Criterion
Ideal Case	5.649536133	5.669647031	8.53474428
Vaclav Skala (en)	5.671691176	6.123714351	9.91764439
Arash Partow	5.659899719	6.12343977	9.937414922
Fowler-Noll-Vo	5.665442527	6.136439866	9.969890452
Dan Julius Bernstein	5.66752388	6.137846283	9.970797952
Kernighan-Ritchie	5.669606762	6.144539002	9.988883127
Donald Ervin Knuth	5.661977	6.143231668	9.998087768
Počet položek ve slovníku			46281
Požadovaná velikost hashovací tabulky			5785
Inicializovaná velikost hashovací tabulky (mocnina dvou)			8192
Inicializovaná velikost hashovací tabulky (prvočíslo)			8209

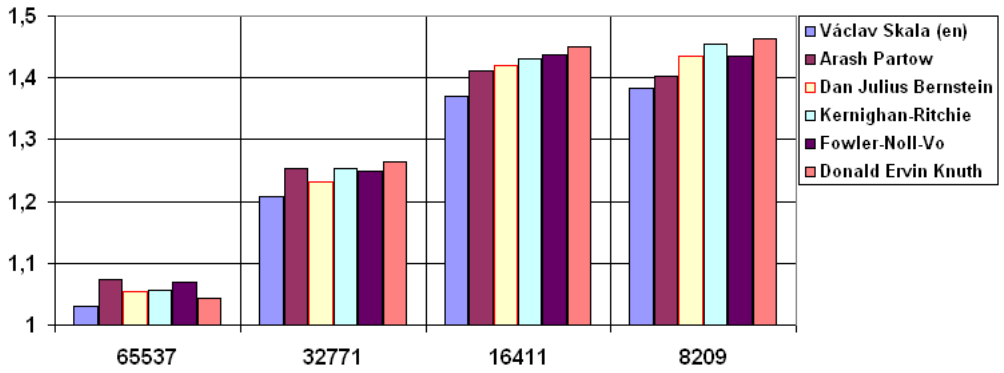
Tabulka 3.4: Data získaná pro anglický slovník.

hodnoty jednotlivých funkcí se od sebe moc neliší, zanesli jsme do grafu rozdíly získaných hodnot vůči ideálnímu případu. Při vytváření jsme pro jednoduchost použili velikost hashovací tabulky vyjádřenou prvočíslem. Některé funkce sice využívají velikost vyjádřenou mocninou dvou, ale dané prvočíslo se příliš neliší od této velikosti. Oba grafy můžeme vidět na obrázcích 3.1 a 3.2.



Obrázek 3.1: Rozdíl kvadratických průměrných délek vůči ideální hodnotě pro anglický slovník.

Z grafů je zřejmé, že nejlepší výsledky jsou získany při použití hasho-



Obrázek 3.2: Rozdíl relatívního kritéria vůči ideální hodnotě pro anglický slovník.

vací funkce navrhnuté v dokumentu [6]. Při testování hashovacích funkcí na českém slovníku se však pořadí hashovacích funkcí změní. To je vidět již v následující tabulce (3.5):

Hash Function Name	Linear Avarage Bucket Size	Quadratic Avarage Bucket Size	Relative Criterion
Ideal Case	1	1	1.5
Fowler-Noll-Vo	1.315863413	1.440531432	2.365516193
Kernighan-Ritchie	1.316435978	1.441032027	2.366130983
Arash Partow	1.316201166	1.441286899	2.367390313
Vaclav Skala (en)	1.317439167	1.442474838	2.369066113
Dan Julius Bernstein	1.317892536	1.443387116	2.371247628
Donald Ervin Knuth	1.319985515	1.446322755	2.377127804
Počet položek ve slovníku			302542
Požadovaná velikost hashovací tabulky			302542
Inicializovaná velikost hashovací tabulky (mocnina dvou)			524288
Inicializovaná velikost hashovací tabulky (prvočíslo)			524309

Tabulka 3.5: Data získaná pro český slovník.

Jak jsme již uváděli v teoretické části této práce, je funkce navrhnuta v dokumentu [6] závislá na koeficientu Q , který se získává experimentálně. Pro český slovník byla experimentálně zjištěna hodnota Q = 0.51234567891376492 Po dosazení do hashovací funkce získáváme následující data:

Hash Function Name	Linear Avarage Bucket Size	Quadratic Avarage Bucket Size	Relative Criterion
Ideal Case	1	1	1.5
Vaclav Skala (cs)	1.313840528	1.437440234	2.359001395
Fowler-Noll-Vo	1.315863413	1.440531432	2.365516193
Kernighan-Ritchie	1.316435978	1.441032027	2.366130983
Arash Partow	1.316201166	1.441286899	2.367390313
Dan Julius Bernstein	1.317892536	1.443387116	2.371247628
Donald Ervin Knuth	1.319985515	1.446322755	2.377127804
Počet položek ve slovníku			302542
Požadovaná velikost hashovací tabulky			302542
Inicializovaná velikost hashovací tabulky (mocnina dvou)			524288
Inicializovaná velikost hashovací tabulky (prvočíslo)			524309

Tabulka 3.6: Data získaná pro český slovník.

Hash Function Name	Linear Avarage Bucket Size	Quadratic Avarage Bucket Size	Relative Criterion
Ideal Case	1.15410614	1.209263586	1.900585704
Vaclav Skala (cs)	1.682331026	1.901125918	3.22256414
Kernighan-Ritchie	1.685114015	1.904734075	3.229465661
Fowler-Noll-Vo	1.684888785	1.904861152	3.230328351
Dan Julius Bernstein	1.684860635	1.904947563	3.23067541
Arash Partow	1.684813722	1.905160746	3.231488521
Donald Ervin Knuth	1.69326259	1.917579273	3.257418805
Počet položek ve slovníku			302542
Požadovaná velikost hashovací tabulky			151271
Inicializovaná velikost hashovací tabulky (mocnina dvou)			262144
Inicializovaná velikost hashovací tabulky (prvočíslo)			262147

Tabulka 3.7: Data získaná pro český slovník.

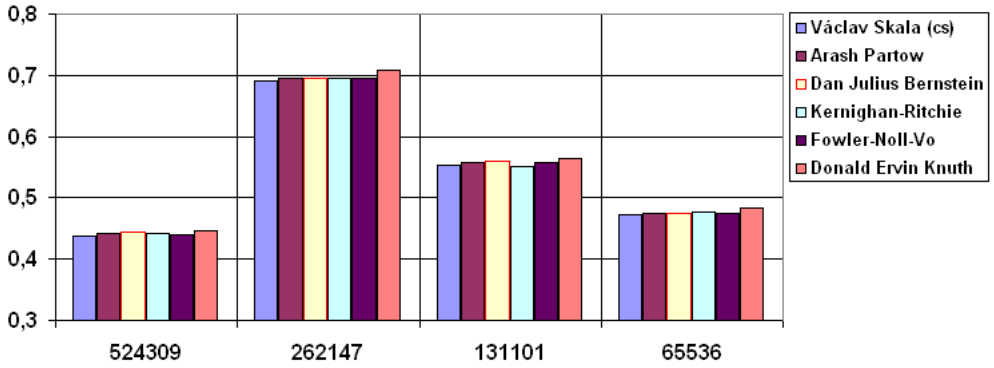
Na následujících obrázcích 3.3 a 3.4 je opět znázorněna závislost kvadratického průměru délky bucketu a relativního kritéria na zvolené velikosti hashovací tabulky. Zde opět vidíme, že nejlepší výsledky jsou pro hashovací funkci navrhnutou Prof. Václavem Skalou.

Hash Function Name	Linear Avarage Bucket Size	Quadratic Avarage Bucket Size	Relative Criterion
Ideal Case	2.30821228	2.353945922	3.600878556
Kernighan-Ritchie	2.558992447	2.905526072	4.94847988
Vaclav Skala (cs)	2.559641954	2.907347245	4.953427954
Arash Partow	2.56287273	2.910517728	4.957979388
Fowler-Noll-Vo	2.561852746	2.910511702	4.959932836
Dan Julius Bernstein	2.564328155	2.913779816	4.966279062
Donald Ervin Knuth	2.565567654	2.917801923	4.977593194
Počet položek ve slovníku			302542
Požadovaná velikost hashovací tabulky			75635
Inicializovaná velikost hashovací tabulky (mocnina dvou)			131072
Inicializovaná velikost hashovací tabulky (prvočíslo)			131101

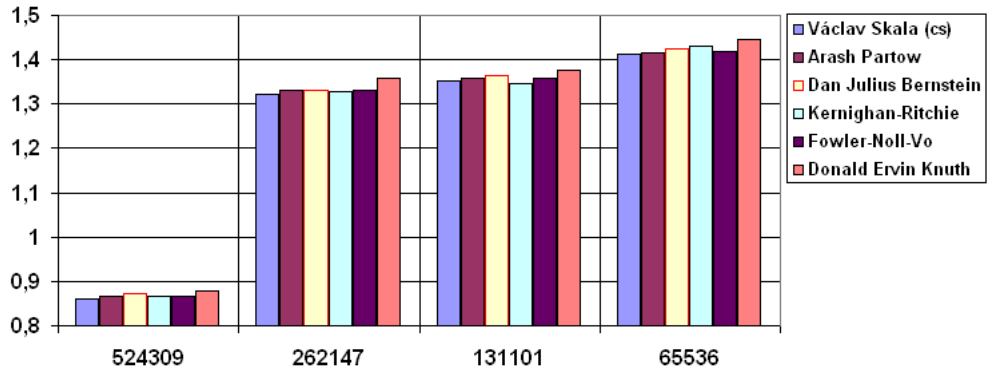
Tabulka 3.8: Data získaná pro český slovník.

Hash Function Name	Linear Avarage Bucket Size	Quadratic Avarage Bucket Size	Relative Criterion
Ideal Case	4.616424561	4.641963059	7.00146426
Vaclav Skala (cs)	4.662238797	5.113592857	8.41296415
Arash Partow	4.663963742	5.115759318	8.416980122
Fowler-Noll-Vo	4.662741774	5.115821217	8.419389705
Dan Julius Bernstein	4.662526199	5.117621571	8.425706183
Kernighan-Ritchie	4.66180776	5.119232315	8.432310225
Donald Ervin Knuth	4.665474116	5.126036505	8.448096463
Počet položek ve slovníku			302542
Požadovaná velikost hashovací tabulky			37817
Inicializovaná velikost hashovací tabulky (mocnina dvou)			65536
Inicializovaná velikost hashovací tabulky (prvočíslo)			65537

Tabulka 3.9: Data získaná pro český slovník.



Obrázek 3.3: Rozdíl kvadratických průměrných délek vůči ideální hodnotě pro český slovník.



Obrázek 3.4: Rozdíl relativního kritéria vůči ideální hodnotě pro český slovník.

Kapitola 4

Závěr

Výsledky práce poukazují na možné využití nového přístupu k volbě, popřípadě návrhu, hashovacích funkcí.

Výsledky měření ukazují, že při znalosti vzorku zpracovaných dat, je velmi vhodné využívat hashovací funkce řízené parametry. Tyto funkce dokážou hodnoty v hashovacích tabulkách rozložit rovnoměrněji a tím zkrátit dobu získávání dat. Tradičně používané hashovací funkce se při volbě různých parametrů omezují jen na parametry související s velikostí návratové hodnoty (viz. např. FNV hash). U hashovací funkce navrhnuté Prof. Václavem Skalou, je důležitá volba parametru (jedná se o koeficient Q). Tento koeficient přímo souvisí s rozložením hodnot ve výsledné hashovací tabulce a je možné ho získat simulací. U testovaných funkcí vedl tento přístup k nejlepším výsledkům. Přitom koeficient Q byl získán krátkou simulací, která zdaleka nepokrývala celý rozsah použitelných koeficientů.

V případě, že by nebyl předem znám vzorek vstupních dat, se naopak jeví vhodnější použití tradičních hashovacích funkcí nebo například funkce navrhnutá Arashem Partowem. Tato hashovací funkce má obecně dobré vlastnosti, pokud je velikost tabulky menší, než je počet vkládaných dat.

Literatura

[1] MULVEY, Bret. Pluto Scarab [online]. 2007 [cit. 2011-05-01]. Hash Functions. Dostupné z WWW: <http://home.comcast.net/~bretm/hash/>.

[2] KNUTH, Donald Ervin. *The Art Of Computer Programming : Seminumerical Algorithms.* 3rd edition. Massachusetts : Addison-Wesley Profesional, 1998. 762 s. ISBN 0-201-89684-2.

[3] KNUTH, Donald Ervin. *The Art Of Computer Programming : Sorting and Searching.* 2nd edition. Massachusetts : Addison-Wesley Profesional, 1998. 780 s. ISBN 0-201-89685-0.

[4] WEISS, Mark Allen. *Data Structures and Algorithm Analysis in C.* 2nd edition.: Addison-Wesley, 1998. 600 s. ISBN 0-201-49840-5.

[5] KERNIGHAN, Brian W., RITCHIE Dennis. *The C Programming Language.* 2nd edition.: Prentice Hall, 1988. 274 s. ISBN 0-13-110362-8.

[6] SKALA, Václav; HRÁDEK, Jan; KUCHAŘ, Martin. *New Hash Function Construction for Textual and Geometric Data Retrieval.*: 2010, [cit. 2011-05-10]. Dostupný z WWW: <http://herakles.zcu.cz/~skala/PUBL/PUBL_2010/2010_Corfu-NAUN-Hash.pdf>.

[7] SKALA, Václav; HRÁDEK, Jan. *Efficient hash function for duplicate elimination in dictionaries.*: 2009, [cit. 2011-05-10]. Dostupný z WWW: <http://herakles.zcu.cz/~skala/PUBL/PUBL_2009/2009_HASH_Dictionary-Algoritmy.pdf>.

[8] JENKINS, Bob. Bob Jenkins Web Site [online]. [cit. 2011-05-01] Hash Functions. Dostupné z WWW: <http://burtleburtle.net/bob/hash/examhash.html>

[9] PARTOW, Arash. General Purpose Hash Function Algorithms [online]. ? [cit. 2011-05-30] Dostupné z WWW: <http://www.partow.net/programming/hashfunctions/>

[10] NOLL, Landon Curt. Fowler / Noll / Vo (FNV) Hash [online]. [cit. 2011-05-30] Dostupné z WWW: <http://www.isthe.com/chongo/tech/comp/fnv/>

[11] Dictionaries - OpenOffice.org Wiki [online]. [cit. 2011-08-05] Dostupné z WWW: <http://wiki.services.openoffice.org/wiki/Dictionaries>