Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE

Petr Kucka

Externí hašování

Katedra softwarového inženýrství

Vedoucí bakalářské práce: Mgr. Jan Lánský
Studijní program: Informatika, správa počítačových systémů

2007

Na tomto místě bych rád poděkoval Mgr. Janu Lánskému za trpělivé zodpovídání otázek ohledně hašovacích algoritmů a podnětné připomínky k práci. Také bych chtěl poděkovat kamarádu Janu Taušovi za velmi užitečné rady o programovacím jazyku Java. A v neposlední řadě bych rád poděkoval své přítelkyni a rodině, za příkladnou péči při psaní této práce.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 8. srpna 2007 Petr Kucka

Obsah

1 Úvod **5**

2 Popis algoritmů **6**
2.1 Cormack 6
2.2 Larson a Kajla 10
2.3 Fagin 11
2.4 Litwin 15
2.5 Skupinové štěpení stránek 16

3 Uživatelská dokumentace **21**
3.1 Instalace 21
3.2 Ovládání 21

4 Programátorská dokumentace **24**
4.1 GUI 24
4.2 Logování 25
4.3 Hašování 26
4.4 Zpoždění 26
4.5 Pársování hašovací funkce 28

Literatura **29**

Seznam programů **30**

Seznam obrázků **31**

A Obsah CD **32**

Název práce: Externí hašování
Autor: Petr Kucka
Katedra: Katedra softwarového inženýrství
Vedoucí bakalářské práce: Mgr. Jan Lánský
e-mail vedoucího: zizelevak@gmail.com

Abstrakt: Hlavním cílem předložené práce je usnadnit výuku předmětu Organizace a zpracování dat a to metod externího hašování. Jedná se konkrétně o Cormackovo perfektní hašování, perfektní hašování Larsona a Kajli, Faginovo rozšiřitelné hašování, Litwinovo lineární hašování a Skupinové štěpení stránek. Výstupem této práce je 5 programů v programovacím jazyce java, které simulují chování výše uvedených metod externího hašování. Programy umožňují záznamy vkládat a vyhledávat. Je také možné zadat hašovací funkce, kterou metody používají.

Klíčová slova: externí hašování, java

Title: External Hashing
Author: Petr Kucka
Department: Department of Software Engineering
Supervisor: Mgr. Jan Lánský
Supervisor's e-mail address: zizelevak@gmail.com

Abstract: In this work we describe five methods of external hashing: Cormack Perfect Hashing, Perfect Hashing from Larson and Kajla, Fagin's Extendible hashing, Litwin's Linear Hashing and Breaking of Pages by Groups. Our main goal is to enhance the lecturer experience of the subject „Data organization and processing". We created five programs in the Java programming language, one for each of these hashing methods. All of these programs enable the user to insert and find arbitrary keys and let him set the hash function, used by the respective method.

Keywords: external hashing, java

Kapitola 1

Úvod

Hlavním cílem této práce je vytvořit programy, které by pomohly při výuce předmětu „Organizace a zpracování dat“ vyučovaném na MFF UK. Jde o výuku metod externího hašování. Těchto metod je v předmětu vyučováno pět. Cormackovo perfektní hašování, perfektní hašování Larsona a Kajli, Faginovo rozšiřitelné hašování, Litwinovo lineární hašování a skupinové štěpení stránek.

Programů je tedy pět a každý simuluje chování jedné hašovací metody. Vzhledem k tomu, že hlavní úloha programů je tyto metody vizualizovat, čili graficky ukázat, zvolil jsem si programovací jazyk Java. Programy jsou zabalené v jar archivech a je možné je spouštět jako samostatnou aplikaci, nebo je vložit do webové stránky jako Java applet. Na spuštění programů je nutné mít nainstalovanou javu verze aspoň 6.0.

Programy jsou určeny pro samostudium při běžné výuce a umožňují prvky vkládat a vyhledávat. Je možné nastavit jak rychle výpočet poběží (s jak dlouhými pauzami), případně ho pozastavit a krokovat po jednotlivých krocích. Součástí programů je log toho, co zrovna program vykonává a také podrobný popis celých hašovacích algoritmů. Hašovací metody používají různé hašovací funkce pro svou činnost a tyto hašovací funkce je možné zadávat. Také je možné nastavovat různé jiné parametry konkrétních metod jako například počet záznamů ve stránce, po kolika vloženích se bude štěpit a podobně.

Kapitola 2

Popis algoritmů

2.1 Cormack

Metoda perfektního hašování, kterou si nyní vysvětlíme, byla vymyšlena G. V. Cormackem v roce 1985 [2]. Je vhodná i pro větší objemy dat a vyznačuje se konstantním přístupovým časem. K vyzvednutí libovolného záznamu jsou třeba nejvýše dva přístupy na disk. Pokud je adresář uložen v paměti, pak pouze jeden. Je to metoda *statická*.

Pomocná datová struktura, kterou budeme používat, se nazývá adresář a je velikosti *O(n)*, kde n je velikost souboru. Pro zkonstruování této hašovací tabulky budeme dále potřebovat **hlavní hašovací funkci** $h(k)$ a **posloupnost nezávislých hašovacích funkcí** $h_i(k, r)$. Parametrem každé funkce z posloupnosti je k – klíč, r – přirozené číslo a vrací hodnotu z intervalu $< 0, r - 1 >$.

Hašovací schéma sestává ze dvou částí. Z *primárního souboru*, ve kterém se nacházejí data a je uložen v pevném paměťovém prostoru a z *adresáře*, který obsahuje informace potřebné pro získání přesné adresy záznamu v *primárním souboru*. Algoritmus získání záznamu potom pracuje tak, že na základě hodnoty *klíče* zjistíme příslušný řádek *adresáře* a pomocí něj spočítáme přesnou adresu záznamu v *primárním souboru*.

Každý řádek adresáře obsahuje tři položky:

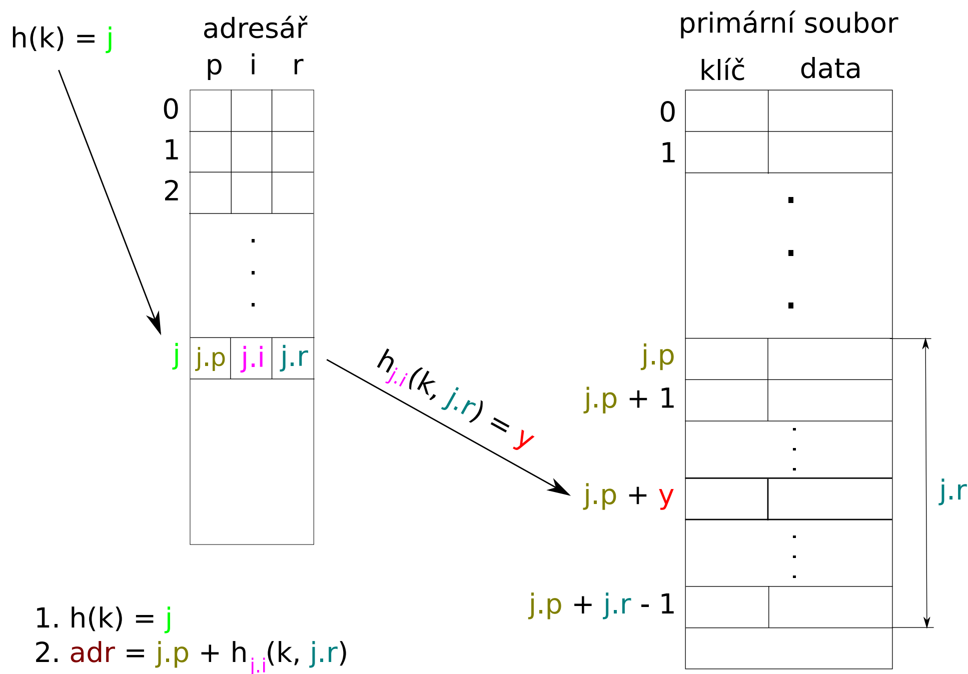
- **p** index do *primárního souboru*,
- **i** index hašovací funkce,
- **r** počet míst na kolik hašovací funkce h_i prvky hašuje.

Pro jednoduchost budeme *primární soubor* simulovat polem záznamů. Hlavní hašovací funkce může být například: $h(k) = k\ mod\ s$, kde s je velikost adresáře a je voleno jako prvočíslo, aby byla zajištěno rozumné rozložení. Posloupnost pomocných hašovacích funkcí nechť je:

$$h_i(k, r) = (k\ mod\ (2i + 100r + 1))\ mod\ r$$

Předpokládáme, že $k >> 2i + 100r + 1$. Pokud tomu tak není můžeme celý prostor klíčů posunout přičtením velkého čísla ke k.

$h(k)$ potom ke každému klíči přiřadí jeden řádek adresáře. Všechny kolidující záznamy (se stejnou hodnotou $h(k)$) jsou uloženy v *primárním souboru* v souvislém úseku délky r, počínaje adresou p a v rámci něho jsou rozmístěny funkcí $h_i(k, r)$. Schéma si můžete prohlédnout na Obrázku 2.1.



Obrázek 2.1: Schéma cormackova hašovaní

Algoritmus přístupu k záznamu je tedy velice jednoduchý a popisuje ho program 2.1 na straně 8.

Také je již dobře vidět, že algoritmus získání záznamu potřebuje pro libovolný záznam nejvýše dva přístupy na disk. Jeden do *adresáře* a jeden

do *primárního souboru*. Každý přístup sestává z přečtení souvislého úseku pevně dané délky a jednoduchého výpočtu. Proto celý přístup k prvku trvá konstantní čas.

Program 2.1: ACCESS (Cormack)

```
/* metoda vrátí pozici záznamu v souboru, nebo -1
* pokud se záznam v souboru nenachází.*/
int access(int klic, Soubor soubor, Adresar adresar) {
  int adr = h(klic); // zjistíme řádek adresáře
  radek = adresar[adr]; // zkratka
  if (radek.r == 0) {
    return =-1; // záznam v souboru není
  } else {
    // pomocí hašovací funkce z posloupnosti, zjistíme
    // přesnou pozici, kde by měl záznam být
    int y = radek.p + h_radek.i(klic, radek.r);
    if (soubor[y].klic == klic){
      return y; // záznam jsme našli
    } else {
      return -1; // záznam v souboru není
    }
  }
}
```

Podívejme se teď na algoritmus vkládání prvků. Všechny hodnoty *adresáře* jsou na začátku nastaveny na 0. Dále budeme potřebovat několik pomocných funkcí. **int getFree(int r)**, která vrátí adresu v *primárním souboru* od které následuje volné místo pro r záznamů. **void setFree(int p, int r)**, která označí r záznamů od adresy p jako volné. **int[] getZaznamy(int p, int r)**, která vrátí pole s r záznamy, uloženými od adresy p. **najdiHash(int klic, int[] kol)**, která najde index i a rozsah r pro funkce $h_i(k, r)$, tak aby funkce h_i byla na prostoru r pro záznamy z pole *kol* a klíč *klic* perfektní hašovací funkcí (tzn. aby hašovala všechny prvky na navzájem různá místa). Funkce zkouší různé indexy i a pokud se jí nepovede prvky perfektně zahašovat, zkusí zvětšit r o jedna (zahašovat prvky do o jedna většího prostoru) a znova přezkouší indexy i.

Algoritmus insert pak popisuje program 2.2 na straně 9.

Program 2.2: INSERT (Cormack)

```
boolean insert(int klic, Soubor soubor, Adresar adresar)
{
  int adr = h(klic); // zjistíme řádek adresáře
  if (adresar[adr].r == 0) { // záznam je první
    int y = getFree(1); // najdeme pro něj volné místo
    soubor[y] = klic; // uložíme ho
    // a zaktualizujeme adresář
    adresar[adr].p = y;
    adresar[adr].i = 0;
    adresar[adr].r = 1;
  } else {
    radek = adresar[adr];
    //vyzvedneme kolidující záznamy
    int[] k = getZaznamy(radek.p, radek.r);
    if (klic je v k) {
      return false; // záznam již v souboru je
    }
    setFree(radek.p, radek.r); //smažeme starou pozici
    // najdeme index hašovací fce a její obor hodnot
    vysl = najdiHash(klic, k);
    int y = getFree(vysl.r); // najdeme volné místo
    // umístíme nový záznam
    soubor[y+h_vysl.m(klic, vysl.r)] = klic;
    // umístíme staré
    for (int zaznam : k){ // pro všechny záznamy v k
      soubor[y+h_vysl.m(zaznam, vysl.r)] = zaznam;
    }
    // zaktualizujeme adresář
    adresar[adr].p = y;
    adresar[adr].i = vysl.m;
    adresar[adr].r = vysl.r;
  }
  return true; // všechno se podařilo
}
```

2.2 Larson a Kajla

Metoda perfektního hašování od Larson a Kajli pochází z roku 1984 [3]. Tato metoda je zástupcem statických metod a hašuje záznamy do pevně daného počtu stránek. Pomocná datová struktura, kterou budeme potřebovat je podstatně menší než u Cormackova algoritmu. Struktura (tabulka) vyžaduje $M * d$ bitů, kde M je počet stránek a d je počet bitů pomocné datové struktury (posloupnost 0 a 1) nazývané **separátor stránky**. Metoda opět zaručuje přístup ke všem prvkům v jednom přístupu na disk.

Dále budeme potřebovat **posloupnost nezávislých hašovacích funkcí** $h_i(k)$ a $s_i(k)$, kde $i \in< 0, \ldots, M-1 >$. Pokud hledáme pro nějaký klíč stránku, do které patří, vytvoříme pomocí funkcí $h_i(k)$ **prohledávací posloupnost stránek**. Prohledávací posloupnost je jednoznačně určená klíčem k a každá taková posloupnost představuje nějakou permutaci množiny adres stránek $\{0, \ldots, \text{M-1}\}$.

Pro každý klíč je také nutné spočítat posloupnost **signatur**. Tu získáme pomocí funkcí $s_i(k)$. Posloupnost je opět jednoznačně určena klíčem k a každá signatura je řetězec 0 a 1 dlouhý d bitů (Pozn. signatury jsou stejně dlouhé jako separátory stránek).

Když se rozhodujeme zdá záznam k patří do stránky $h_j(k)$, použijeme **signaturu** $s_j(k)$ a **separátor stránky j**. Pokud je signatura **menší** než separátor, záznam patří do této stránky. Pokud ne, zkusíme další stránku z prohledávací posloupnosti.

Záznamy jsou ve stránce setříděny podle svých signatur. Při vkládání záznamu do stránky, může nastat situace, že záznam do stránky patří, ale stránka je již plná. Potom dojde k **vytlačení** některých záznamů. Všechny záznamy patřící do stránky jsou setříděny vzestupně podle svých signatur a záznam (nebo záznamy - pokud mají stejné signatury) s největší signaturou bude vytlačen. Separátor stránky je potom změněn na nejnižší signaturu vytlačených záznamů (Pozn. separátor bude ostře větší, než všechny signatury ve stránce). Přetečený záznam (záznamy) jsou umístěny do jiných stránek dle svých prohledávacích posloupností. To vede k řetězení operací INSERT.

Nyní nám již zbývá jen vyřešit jakou počáteční hodnotu by měli mí separátory stránek. Na začátku algoritmu je všechny nastavíme na $2^d - 1$, kde d je délka signatur a separátorů. Protože ale není možné, aby signatury potom nabývaly této hodnoty, je nutno při výpočtu s_i testovat, jestli není signatura rovna $2^d - 1$ a pokud ano tak jí v tomto případě nahradit nulou.

Algoritmus nalezení adresy záznamu popisuje program 2.3 na stránce 11.

Program 2.3: GetAdr (Larson)

```
/* metoda vrátí stránku do které záznam
 * patří a k ní odpovídající signaturu.
 * Nebo -1, -1 pokud taková stránka není. */
getAdr(int klic, int[] sep) {
  // projdeme prohledávací posloupnost
  for (int i=0; i<pocetStranek-1; i++)
  {
    int adr = h_i(k);
    int sig = s_i(k);
    if (sig < sep[adr]) // je stránka správná?
    {
      return (adr, sig);
    }
  }
  return (-1, -1);
}
```

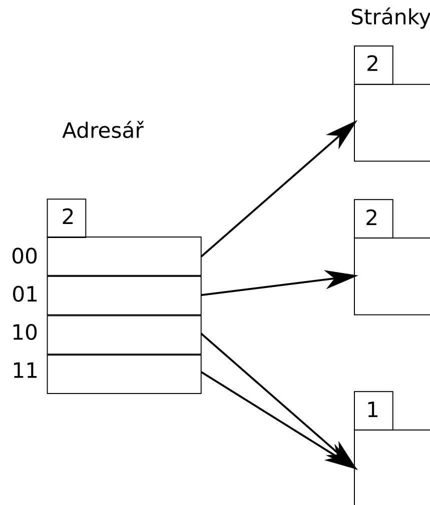
Algoritmus ACCESS potom jen získá stránku ve které by měl záznam být metodou getAdr, prohledá stránku a ohlásí, jestli tam záznam je, nebo není. Pokud není v této stránce, nikde jinde se již nacházet nemůže.

INSERT získá stránku od metody getAdr, vloží do ní záznam a pokud je plná, tak případné přetečené záznamy znovu vloží. Velká nevýhoda této metody je, že při vkládání prvků časem dojde k tomu, že se všechny stránky naplní. Potom již další záznamy vkládat nelze.

2.3 Fagin

Konečně se dostáváme k zástupcům dynamických hašovacích metod a to konkrétně k rozšiřitelnému hašování Ronalda Fagina [4]. Hašování sestává z **adresáře** a stránek v **primárním souboru**. Adresář obsahuje ukazatele(ne nutně různé) na stránky s daty v primárním souboru. Schéma tedy funguje tak, že dle výsledku **hašovací funkce h** přečteme řádek z adresáře a tak zjistíme stránku do které záznam patří.

Nechť máme hašovací funkci h, potom nazveme $k' = h(k)$ pseudoklíč



Obrázek 2.2: Faginovo hašování pro hloubku adresáře 2

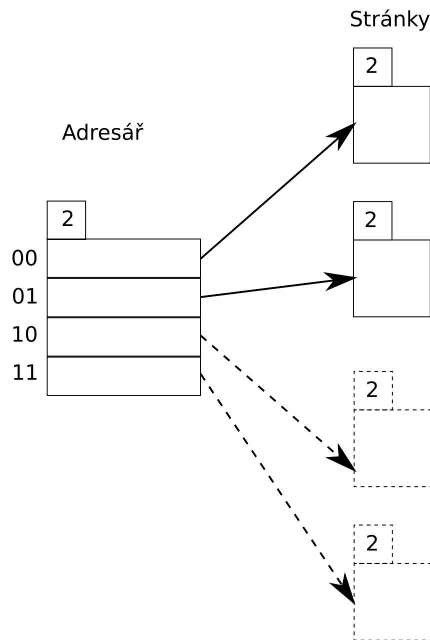
odpovídající klíči k. Pseudoklíče mají pevnou délku r (Pozn. klíče mít nemusejí). O funkci h předpokládáme, že pseudoklíče jsou na jejím oboru hodnot rozděleny rovnoměrně. To znamená, že přibližně polovina pseudoklíčů začíná 0, čtvrtina 01 atd.

Součástí adresáře je i hlavička, ve které je uložena *hloubka adresáře* d, $0 < d \leq r$, která říká, kolik prvních bitů z pseudoklíče použijeme pro zjištění řádku adresáře. Po ní následují odkazy na stránky. Nejdříve je ukazatel na stránku, která obsahuje všechny klíče k takové, že jejich pseudoklíč *k'* začíná d po sobě jdoucími nulami. Následuje ukazatel na stránku pro všechny klíče, jejichž pseudoklíč začíná d bitovou sekvencí 0. . . 01, potom ukazatel pro pseudoklíč začínající d bity 0. . . 010 a tak dále lexikograficky. Poslední ukazatel je tedy na stránku obsahující všechny klíče s pseudoklíčem začínajícím d po sobě jdoucími jedničkami. Například pro $d = 2$ může situace vypadat jak ukazuje obrázek 2.2

V každé stránce primárního souboru je také uložena *lokální hloubka d'*, $d' \leq d$, která říká, že pseudoklíče klíčů v ní obsažených mají prvních *d'* bitů shodných. Pokud je tedy $d' < d$ znamená to, že na tuto stránku vede odkaz z více řádek adresáře. Hloubka adresáře je potom maximum z lokálních hloubek všech stránek.

Pokud vkládáme záznam do stránky, může nastat několik případů.

- První a nejjednodušší, že je ve stránce místo. Potom záznam prostě



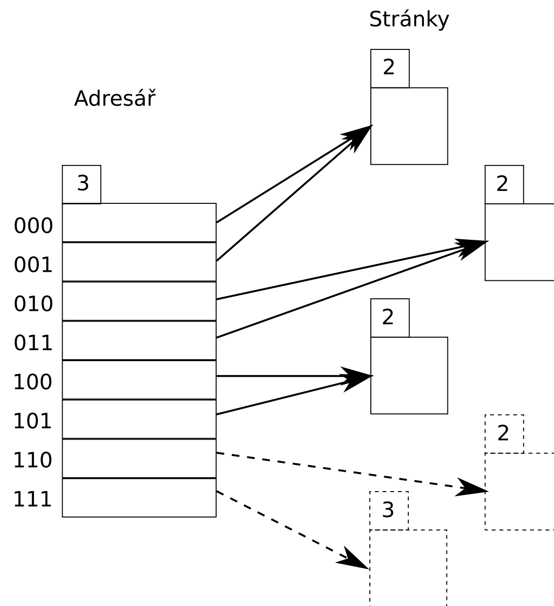
Obrázek 2.3: Faginovo hašování po štěpení stránky

přidáme do stránky.

- Druhý případ nastane, když stránka je již plná, ale platí, že lokální hloubka stránky je menší než hloubka adresáře ($d' < d$). Pak se stránka rozštěpí na dvě o lokálních hloubkách $d' + 1$ a záznamy se mezi nimi rozdělí podle $d' + 1$ bitu jejich pseudoklíče. Naše schéma se tedy změní a bude vypadat jako ukazuje obrázek 2.3

- Poslední případ nastane, když je stránka plná a zároveň lokální hloubka stránky je stejná jako hloubka adresáře ($d' = d$). Potom musíme zvětšit hloubku adresáře o jedna, čili zdvojnásobit adresář. Teprve pak rozštěpíme stránku na dvě o lokálních hloubkách $d + 1$ a záznamy v nich rozdělíme podle $d + 1$ bitu pseudoklíčů. Po rozštěpení adresáře a stránky bude situace vypadat jako na Obrázku 2.4 na stránce 14.

Algoritmus zdvojnásobení adresáře hloubky d provedeme jedním průchodem starého adresáře odspoda nahoru. Čas k tomu potřebný je $O(2^d)$, přičemž žádné stránky nemusí být dotčeny, samozřejmě až na tu, jejíž přetečení štěpení adresáře vyvolalo. Záznamy ve stránce je možné ukládat také pomocí



Obrázek 2.4: Faginovo hašování po štěpení adresáře

hašování, přirozené je použít například zbývajících $r - d$ bitů z pseudoklíče. Kolize je možné řešit libovolným způsobem, například řetězením. Jak je snadno vidět k nalezení záznamu potřebujeme maximálně dva přístupy na disk.

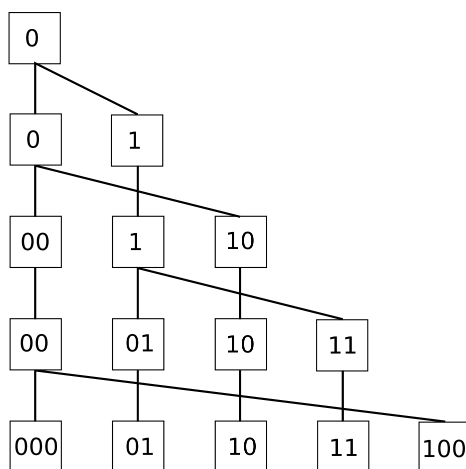
Algoritmus nalezení stránky pro záznam potom probíhá v několika krocích následujícím způsobem.

1. Spočítáme $k' = h(k)$.
2. Přečteme hloubku adresáře d.
3. Vezmeme prvních d bitů z k', interpretujme je jako binární číslo a nazveme jej p.
4. Vyzvedneme ukazatel, který je umístěn $p * v$ bytů od začátku adresáře, kde v je velikost každého ukazatele v bytech.
5. Tento ukazatel vede na hledanou stránku.

2.4 Litwin

Dalším zástupcem dynamických hašovacích metod, je lineární hašování [5]. Metoda tentokrát nevyužívá adresář, ale daní za to je, že potřebuje **oblast přetečení** a tedy není zaručeno, že se ke všem datům dostaneme v jednom přístupu na disk. Data jsou uložena ve stránkách.

Hlavní myšlenka spočívá v tom, že stránky jsou štěpeny nezávisle na tom, kde došlo ke kolizi, kruhovým schématem, který znázorňuje obrázek 2.5. Po každých L (L je parametr metody) vloženích je rozštěpena jedna stránka. Když dochází ke štěpení stránky, přidáme jednu stránku na konec **úložiště stránek**. Záznamy ze štěpené stránky (a případné z oblasti přetečení, které by patřily do štěpené stránky) jsou potom rozděleny mezi novou a štěpenou stránku. Úložiště stránek je tvořeno n stránkami očíslovanými $0, \dots, n-1$.



Obrázek 2.5: Schéma štěpení při lineárním hašování

Pro zjištění adresy stránky do které záznam patří, budeme potřebovat hašovací funkci $h(k)$. Potom nazveme $k' = h(k)$ pseudoklíč. Pokud hašovací funkce rozděluje záznamy do stránek rovnoměrně, tak potom díky kruhovému schématu štěpení stránek (na každou dříve nebo později dojde) bude počet přetečených záznamu „rozumně“ malý.

Na začátku algoritmu je pouze jedna stránka do které patří všechny záznamy. Po L operacích INSERT dojde k jejímu rozštěpení. Záznamy, které na konci pseudoklíče mají 0 zůstanou ve stránce a ty jejichž pseudoklíč končí na 1 se přesunou do nové stránky. Při rozhodování do které stránky záznam

patří potom spoužijeme poslední bit pseudoklíče.

Při dalším štěpení (Pozn. opět po L operacích INSERT) rozštěpíme stránku 0. Tentokrát použijeme již druhý bit od konce pseudoklíče a tedy budou záznamy, dle něj rozděleny na záznamy s 00 a 10 na konci pseudoklíče. Situace tedy bude vypadat jako ve třetím řádku Obrázku 2.5 na stránce 15. Na tomtéž obrázku je také vidět jak štěpení probíhá dále a také kolik posledních bitů pseudoklíče mají záznamy ve stránce shodné.

Algoritmy INSERT a ACCESS potom probíhají tak, že zjistíme adresu stránky, jak ukazuje funkce GetAdr v programu 2.4, a potom záznam do stránky vložíme (pokud se nevejde, tak do oblasti přetečení), případně jí prohledáme, a zjistíme zda se tam záznam nachází (pokud ne, a stránka je plná, může být záznam ještě v oblasti přetečení).

Program 2.4: GetAdr (Litwin)

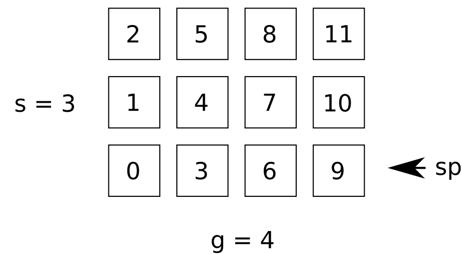
```
/* funkce vrátí číslo stránky, která odpovídá klíči. */
int GetAdr(int klic) {
  // zjistíme kolik posledních bitů nás zajímá
  // Pozn. Ceil = Horní celá část.
  int w = Ceil(log2(pocetStranek));
  // uřízneme z pseudoklíče potřebný počet bitů
  a = h(klic) mod 2^w;
  // pokud stránka v tomto kole ještě neprošla
  // štěpením zajímá nás ještě o bit míň
  if (a >= pocetStranek) a = a mod 2^(w-1);
  // máme číslo stránky
  return a;
}
```

2.5 Skupinové štěpení stránek

Další metoda externího hašování na kterou se podíváme je skupinové štěpení stránek [1]. Jedná se o jakési rozšíření lineárního litwinova hašování. U lineárního hašování jsou stránky většinou nerovnoměrně zaplněny, ty které jsou po štěpení mívají prvků málo, naopak ty které jsou těsně před štěpením jsou často plné. Metoda tedy opět nepotřebuje adresář ani žádnou jinou pomocnou strukturu, neobejde se však bez oblasti přetečení.

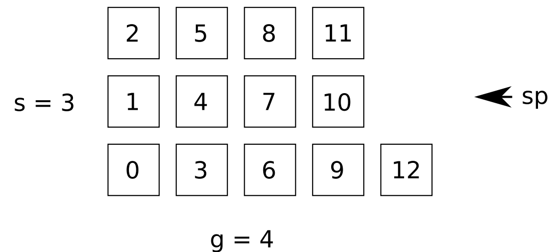
Skupinové štěpení stránek řeší problém s nerovnoměrným rozložením záznamů rozdělením stránek do několika skupin. Vždy když štěpíme skupinu, přibude do ní jedna stránka a záznamy z celé skupiny jsou rozděleny do o jedna většího počtu stránek, čímž dosáhneme rovnoměrnějšího rozložení záznamů ve stránkách. *Počet skupin s* se v průběhu algoritmu mění, naproti tomu *počet stránek ve skupině g* zůstává po celou dobu běhu stejný.

Nechť máme **s skupin** po **g stránkách** a **sp** ukazuje na skupinu, která je na řadě se štěpením. Potom situace na začátku algoritmu vypadá jako na obrázku 2.6. Všimněme si, že čísla stránek ve skupině nejdou za sebou, ale liší se právě o s.



Obrázek 2.6: Skupinové štěpení stránek - začátek algoritmu pro $s = 3$ a $g = 4$.

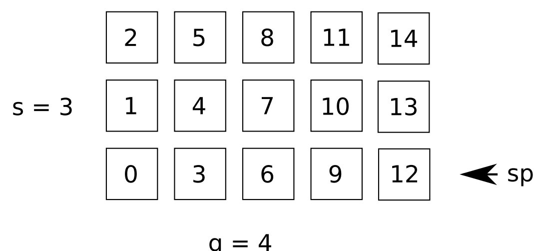
Po L operacích INSERT dojde k prvnímu štěpení. Do skupiny na kterou ukazuje *sp* přibude jedna stránka, záznamy z celé skupiny jsou přerozděleny z g do *g+1* stránek a *sp* se posune na další skupinu. Situaci po prvním štěpení ukazuje obrázek 2.7.



Obrázek 2.7: Skupinové štěpení stránek - po štěpení první skupiny.

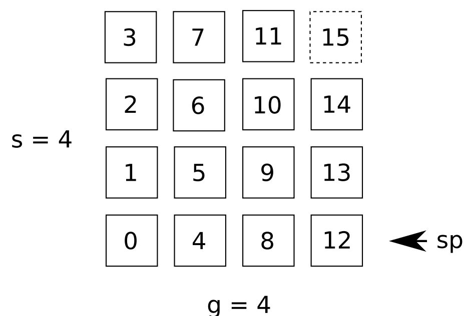
Po dalších L vloženích je rozšířena další skupina, záznamy z ní přerozděleny do *g+1* stránek a to se opakuje dokud neprojdou štěpením všechny

skupiny. Nyní máme tedy s skupin po g+1 stránkách. Situace po rozštěpení všech skupin vypadá jako na obrázku 2.8.



Obrázek 2.8: Skupinové štěpení stránek - po rozštěpení všech skupin.

Když jsou všechny skupiny rozštěpeny, dojde k „reorganizaci“ stránek. Reorganizace je to pouze virtuální, žádné záznamy při ní nejsou přesouvány. Stránky jsou znova rozděleny tak, aby byl počet prvků ve skupině opět g, jako na začátku algoritmu. Dojde tedy ke zvětšení počtu skupin stránek. Pokud je stránek málo na to, aby všechny skupiny měli g stránek, chybějící stránky doplníme. Situaci po reorganizaci znázorňuje obrázek 2.9. Doplnit jsme museli jednu stránku a nyní tedy máme *4* skupiny po *4* stránkách. Dále algoritmus pokračuje rozštěpením skupiny na kterou ukazuje *sp* atd.



Obrázek 2.9: Skupinové štěpení stránek - po první reorganizaci.

Pro nalezení stránky, kde by se měl záznam nacházet, budeme potřebovat posloupnost nezávislých hašovacích funkcí $h_i(k) : K^* \mapsto < 0, g >, i \in 1 \ldots$ Všimněme si, že obor hodnot hašovacích funkcí je $< 0, g >$ to proto, že s jejich pomocí přerozdělujeme záznamy ve skupině z g do **g+1** stránek. Dále budeme potřebovat základní hašovací funkci $h(k) : K^* \mapsto 0, \ldots, g * s_0 - 1$,

kde s_0 je počáteční počet skupin, která záznamy rozdělí do počátečního počtu stránek. Potom už stačí jenom dosavadní počet reorganizací d (Pozn. Na začátku algoritmu $d = 0$), sp ukazující na skupinu, která je na řadě se štěpením a můžeme napsat funkci GetAdr tak, jak jí popisuje program 2.5. Funkce vrátí pro klíč adresu stránku ve které by se měl nacházet (případně do ní patří).

Program 2.5: GetAdr(Skup)

```
/* Funkce vrátí adresu stránky pro záznam klic
 * d je dosavadní počet reorganizací
 * sp je číslo skupiny, která bude štěpena */
int GetAdr(int klic, int d, int sp) {
  // inicializace počtu skupin
  s = s0;
  // počáteční zahašování do prostoru s0*g stránek
  h = h(klic);
  // projdeme všechny dosavadní reorganizace
  for (int i = 0; i<d-1; i++) {
    // nová adresa záznamu po aktuální reorganizaci
    h = h mod s + hi(klic)*s;
    // nový počet skupin po aktuální reorganizaci
    s = Ceil(s*(g+1)/g);
  }
  // pokud již v tomto cyklu skupina prošla štěpením
  // musíme ještě jednou prehašovat do g+1 stránek
  if (h mod s < sp){
    h = h mod s + hd(klic)*s;
  }
  // vrátíme spočítanou adresu stránky
  return h;
}
```

Algoritmus hledání záznamu funguje tak, že prochází celou historii kde byl záznam uložen. Nejdříve ho zahašuje do původního počtu stránek základní hašovací funkcí a potom ho postupně přesouvá pomocí hašovacích funkcí $h_i(k)$ tak jak záznam v průběhu vkládání prvků cestoval. Nakonec ještě musíme zjistit, zda v tomto cyklu již došlo v této skupině ke štěpení (a

tedy je $h \ mod \ s < sp$) a pokud ano, tak je potřeba ještě jednou přehašovat za to poslední štěpení.

Důležité je nezapomenout, že skupinové štěpení stránek se neobejde bez oblasti přetečení a tedy pokud záznam nelze vložit do jemu určené stránky (protože je plná) vložíme ho do oblasti přetečení. Když potom při štěpení dochází k přerozdělování prvků z g do $g+1$ stránek, přidáme do skupiny i záznamy z oblasti přetečení, pro které se případně uvolnilo místo. Při hledání záznamu je také třeba prohledat oblast přetečení, pokud je stránka, kde by se měl záznam nacházet plná.

Kapitola 3

Uživatelská dokumentace

3.1 Instalace

Každá metoda potřebuje ke své činnosti tři .jar archivy. Jeden speciální pro každdou metodu a potom archivy: utils.jar –obsahující pomocné třídy a vectorvisuals.jar –obsahující externí knihovnu pro práci s grafikou. Pokud chceme některou metodu spustit, musí se tyto tři archívy nacházet ve stejném adresáři.

Programy jsou napsané zároveň jako Java aplikace a zároveň jako Java applet. Pokud chceme metodu spustit jako samostatnou aplikaci můžeme tak učinit příkazem „java -jar <jmeno-archivu>“. V případě že bychom chtěli programy umístit do webové stránky jako applety umožní nám to html tag applet. Například pro Cormackovo perfektní hašování stačí tedy, pokud jsou jar archivy ve stejném adresáři jako html soubor, do html stránky vložit:

```
<applet code="cormack.AppletCormack.class"
  archive="cormack.jar, utils.jar, vectorvisuals.jar"
  width="800" height="700"
  alt="Potřebujete javu verze 6, nebo vyšší,
  abyste mohli spustit applet znázorňující
  cormackovo hašování" />
```

3.2 Ovládání

Programy jsou rozděleny na tři na první pohled patrné části. Nahoře se nachází textová část, kde se vypisuje co metoda právě vykonává (pokud něco).

Textová část má záložky (tabs). V druhé záložce je podrobněji popsaná hašovací metoda a ve třetí záložce je Javascriptová syntaxe, která se vám bude hodit při zadávání hašovacích funkcí. Zadávají se v Javascriptové syntaxi. Některé řádky v první záložce (výpisu toho co program, právě vykonává) jsou podtržené, na ty je možné kliknout a program automaticky otevře druhou záložku a zaskroluje na konkrétní místo, které se týká kliknuté řádky.

Střední část je zabraná samotnou metodou. Zde jsou nakresleny stránky, případně jiné struktury, které daná hašovací metoda používá a také se zde odehrává přidávání prvků, stránek a podobně. Pokud grafika vyleze z viditelné části, objeví se skrollbary a je tedy možné si prohlédnout celou situaci.

Ve spodní části programu se nachází dva panely. Ovládací a nastavovací. Ovládací panel je pro všechny metody stejný, kdežto nastavovací je různý, neboť každá metoda má trochu jiné možnosti nastavování. U každé metody je ovšem na konci nastavovacího panelu tlačítko „Nové hašování“, které smaže vložené prvky a z nastavených hodnot vytvoří nové hašování.

Ovládací panel začíná místem pro napsání klíče, který chcete vložit, případně vyhledat. Je možné zadávat pouze čísla z intervalu $< 1,999 >$. Potom jsou tlačítka „Insert“ které tento klíč vloží a „Access“, které ho vyhledá. Dále je tlačítko „Random“, které vygeneruje náhodné (alespoň tak náhodné jako je javová třída Random) číslo a připraví ho k vložení. Dále je možné změnit delay v milisekundách se kterým se program vykonává. Po změně je třeba stisknout tlačítko „Změň delay“. Pokud program běží, je možné ho zastavit tlačítkem „Pause“ a potom ho znova spustit tlačítkem „Resume“, nebo ho nechat vykonat jen jeden krok tlačítkem „Krok“.

V nastavovacím panelu se nastavují převážně hašovací funkce specifické pro každou metodu. Většinou je u těchto metod předpokládán nějaký definiční obor a tak je výsledek funkce na závěr zmodulován (operací mod) tak, aby funkce nevracela moc velká čísla a nedocházelo k nestandardním situacím (jako že bychom četli z adresáře 20. řádek, když jich má jenom pět a podobně). Pokud je tedy funkce zadaná správně nemá na její výsledek toto modulování vliv, pokud je však funkce zadaná špatně bude vracet jiné výsledky.

3.2.1 Cormack

U Cormackova perfektního hašování je možné nastavit primární funkci **h(k)**, která určuje řádek adresáře, potom také sekundární hašovací funkce **h(i,k,r)**, které rozdělují záznamy z jednoho řádku adresáře. Dále můžeme zadat hod-

notu **maxI** což je maximální hodnota indexu i, která se má zkoušet při hledání perfektní hašovací funkce h(i,k,r). Pokud se nepovede najít perfektní hašovací funkci pro i z intervalu $< 0, maxI >$, zvedne se o jedna r (Pozn. r se zvedá do dvojnásobku počtu prvků které hašujeme, když ani potom není nalezena perfektní hašovací funkce, záznam je prohlášen za nevložitelný.) Poslední hodnota kterou můžeme u cormacka změnit je **velikost adresáře s**, čili kolik řádků bude mít adresář.

3.2.2 Larson a Kajla

U perfektního hašování Larsona a Kajli můžeme zadávat hašovací fce **h(i,k)**, které generují prohledávací posloupnost. Také funkce **s(i,k)**, které vytvářejí posloupnost signatur. Potom je možné zadat kolik **bitů** budou mít signatury a separátory. Dále také **počet stránek** a nakonec **počet záznamů na stránce p**.

3.2.3 Fagin

U Faginova rozšiřitelného hašování je situace velice jednoduchá. Stačí nastavit hlavní hašovací funkci **h(k)**, potom **počet bitů**, které tato hašovací funkce vrací. Z hašovací funkce totiž využíváme pouze několik bitů od začátku a tak je nutné vědět kolik funkce vrací bitů, abychom věděli kde je začátek. Poslední nastavitelná hodnota je **počet prvků na stránce p**.

3.2.4 Litwin

U Litwinova lineárního hašování je opět možné zadat hlavní hašovací funkci $h(k)$, která určuje do které stránky bude záznam patřit. Potom také **parametr L**, který udává po kolika vloženích dojde ke štěpení stránky. Nakonec je také možné zadat **počet prvků na stránce p**.

3.2.5 Skupinové štěpení stránek

U skupinového štěpení stránek je možné zadat hlavní hašovací funkci **h(k)**, která provádí základní zahašování do počátečního počtu stránek. Potom také posloupnost nezávislých hašovacích funkcí **h(i,k)**, které prvky přerozdělují v rámci skupiny. Dále **počáteční počet skupin s0** a **počet stránek ve skupině g**. Také **parametr L**, čili po kolika vloženích dojde ke štěpení skupiny a nakonec **počet záznamů na stránku p**.

Kapitola 4

Programátorská dokumentace

Celá bakalářská práce se skládá ze šesti Java balíčků (package). Každé hašování tvoří jeden balíček („cormack“, „larson“, „fagin“, „litwin“, „skup“) a navíc je ještě balíček „utils“, který obsahuje pomocné a abstraktní třídy ze kterých jsou odvozeny třídy konkrétní pro každé hašování.

Další balíček, je balíček „vectorvisuals“, pomocí kterého jsem kreslil grafickou část aplikace. Balíček pochází z www.vectorvisuals.com a poskytuje objektové API pro kreslení a manipulaci s 2D grafickými tvary a obrázky. Děkuji tímto jeho tvůrcům.

Kompletní dokumentaci ke všem třídám, vytvořenou pomocí nástroje Javadoc naleznete na přiloženém CD. Podívejme se proto jen na programátorsky nejzajímavější části mé práce.

4.1 GUI

K vytvoření grafického uživatelského rozhraní (GUI) byla použita grafická knihovna „Swing“. Jako nejvyšší třída, do které je celé GUI vloženo, jsem použil *JPanel*. Díky tomu je možné tento panel vložit do třídy *JFrame* a tedy spustit jako samostatnou aplikaci, nebo zabalit do třídy *JApplet* a spustit jako Java applet ve webové stránce.

Společná část grafického rozhraní je popsána v abstraktní třídě *utils.Panylek*. Zde je zavedena celá horní textová část aplikace (kromě konkrétního popisu hašovací metody) ve střední části je připraveno plátno na kreslení a také je zde vytvořen celý ovládací panel. Tato třída obsahuje dvě abstraktní metody *void initGUI()* a *void initializeHash()*, které musí každý potomek této třídy implementovat. V metodě *initGUI* potomek vytvoří svůj panel s nastavením

(který je pro každou metodu různý) a metoda *initializeHash* přečte po stisku tlačítka „Nové hašování“ hodnoty z tohoto panelu s nastavením a pokud jsou správně zadané, vytvoří nové hašovaní s těmito hodnotami a vrátí hašovací algoritmus na začátek.

Každá hašovací metoda potom obsahuje třídu *PanelXyz* (například pro Faginovo rozšiřitelné hašování se tato třída jmenuje *PanelFagin*). Kde tyto metody implementuje a dodefinuje si tak část grafického rozhraní, které je specifické jen pro ní.

4.2 Logování

Každá metoda v horní textové části loguje co právě vykonává. Protože výpisy občas obsahují horní nebo dolní index a zároveň jsem chtěl aby bylo možné z programu výpisy barevně odlišovat, případně občas zvýraznit jen některou část výpisu. Rozhodl jsem se použít jazyk html. Textová část je tedy tvořena swingovou komponentou „JEditorPane“. Tato komponenta umí interpretovat html značky, pokud je nastaven správný content-type. Log je tedy vlastně html stránka, do které jen přidáváme řádky, v nichž tedy můžeme používat html značky pro zvýrazňování, ale také horní nebo dolní index. Pomocí stylů je také možné jednotlivé řádky odlišit barevně.

Popis metod je tedy potom možné napsat jako html stránku se všemi z toho plynoucími výhodami i nevýhodami. Každý hašovací balíček, obsahuje html stránku stejného jména (například v balíčku (adresáři) *cormack* je soubor *cormack.html*), ve které je popsána celá metoda a zároveň na některých zajímavých místech nadefinovány kotvy (pomocí). Tento soubor je v metodě *initGUI* načten do druhé záložky textové části.

Podtržené klikatelné řádky, jak jste si již jistě domysleli, nejsou nic jiného než obyčejné html odkazy. Aby na ně však bylo možné klikat je nutné, aby každý *PanelXyz* implementoval interface *HyperlinkListener* a tedy definoval metodu *void hyperlinkUpdate(HyperlinkEvent e)*, která je zavolána, když se stane nějaká událost z odkazem. Tato metoda potom říká, že pokud je kliknuto na odkaz, je otevřena druhá záložka a zaskrolováno na konkrétní kotvu.

4.3 Hašování

V každém z balíčků obsahujících některé hašování také najdeme třídu *XyzHash* (například u Faginova hašování je to třída *FaginHash*). Jelikož jednotlivé hašovací metody jsou dosti různé, jsou i tyto třídy dosti různé, společné však mají to, že obsahují metody *void insert(int klic, Program program)* a *void access(int klic, Program program)*. Smysl parametru *Program* bude osvětlen v sekci „Zpoždění" této kapitoly. Tyto metody jsou volány po stisku tlačítka „Insert" nebo „Access" a parametr *klic* do souboru vloží respektive ho v souboru vyhledají.

4.4 Zpoždění

Aby byly programy k výuce užitečné, bylo nutné nějak vyřešit otázku zpoždění programu respektive jeho pomalého vykonávání. Protože prováděné výpočty jsou velice jednoduché a netrvají tedy prakticky žádný čas, bylo nutné programy zpomalit uměle.

Grafická knihovna „Swing" není threadsafe a tedy je nutné veškeré grafické operace provádět v hlavním swingovém vlákně. Zároveň ale není možné toto hlavní vlákno uspat (pozastavit) neboť obsluhuje celé grafické uživatelské rozhraní. Ve chvíli, kdy bychom uspali toto hlavní swingové vlákno aplikace by vůbec na nic nereagovala (zvětšení okna, kliknutí myší...), prostě by celá zamrzla.

4.4.1 Myšlenka

Problém jsem tedy vyřešil tak, že jsem vytvořil vlastní třídu *utils.Program*, které se spouští ve zvláštním vlákně. Tato třída obsahuje frontu událostí, které je třeba vykonat. Události jsou dvojího druhu. Za prvé *grafické*, ty jsou spouštěny v kontextu hlavního swingového vlákna a za druhé *čekací*, které jsou spouštěny ve zvláštním vlákně.

Práce s *programem* tedy probíhá tak, že do jeho fronty přidáváme akce které potřebujeme vykonat. Akce jsou typu: zobraz stránku, posuň stránku, změň barvu stránky, vypiš zprávu na log, počkej 1000ms... Když jsou všechny akce, které chceme, ve frontě. Spustíme *program* (vlákno). Ten bere ze své fronty událostí naplánované akce a postupně, jak byly přidávány, je vykonává. Když vykoná poslední akci skončí a vlákno zanikne. Pokud program v

průběhu jeho práce přerušíme, přestane vykonávat *čekací* akce a tedy velice rychle dokončí svou frontu naplánovaných akcí.

Pokud tedy při hašování někdo zmáčkne například tlačítko „Insert“ děje se následující. Vytvoříme novou instanci třídy *Program*. Potom provádíme výpočet dle konkrétní hašovací metody a vždy když by se měla v tomto výpočtu stát nějaká grafická (viditelná) událost přidáme tuto událost do fronty tohoto nově vzniklého objekt. Poté co výpočet skončí, prvek je vložen na svém místě (my však stále nic nevidíme), tak již máme ve frontě naplánované všechny viditelné události potřebné pro vložení tohoto prvku. Pokud nějaký *program* zrovna běží (právě dochází například ke vkládání předchozího prvku) tento běžící program ukončíme (necháme ho dokončit rychle, bez čekání), počkáme až doběhne a těsně po něm spustíme náš nový *program*. Ten potom, dle nastaveného zpoždění (delay), začne vykonávat naplánované akce a když všechny vykoná skončí.

Běžící *program* je také možné zastavit tlačítkem „Pause“ tím vlákno které právě vykonávaný *program* představuje pozastavíme funkcí „wait()“. Vlákno čeká do té doby, než je znovu spuštěno tlačítkem „Resume“, nebo ukončeno (donuceno rychle dokončit svou frontu bez čekání) spuštěním nového *programu*. Pokud je stisknuto tlačítko „Krok“, *program* ze své fronty vyjme jednu událost, tu vykoná a znovu čeká.

4.4.2 Implementace

Nyní se podíváme jak je třída *utils.Program* řešena technicky. Třída obsahuje interface *Action* a abstraktní třídu *SwingAction*, která tento interface implementuje. Fronta naplánovaných událostí je potom *List<Action>*. Pokud chceme aby byla akce vykonána ve swingovém vlákně (tedy je grafická) odvodíme jí od třídy *SwingAction*. Pokud má být vykonána ve speciálním vlákně, necháme jí implementovat interface *Action*.

Jednotlivé akce jsou tedy třídy vnořené ve třídě *Program*. *Program* dále obsahuje metody, které tyto akce přidají do fronty. Ukažme si to na příkladu. Jedna z akcí je *LogFormatAction*, která na log vypíše zprávu dané důležitosti. K ní existuje několik metod, které konkrétně důležitou akci přidají do fronty. Jako třeba *infof, warningf. . .* Zavoláním *program.infof(„Toto je info akce.“)* tedy přidáme do fronty akci vypisující na log informativní zprávu.

4.5 Pársování hašovací funkce

Část, která se z počátku jevila jako nejobtížnější, a to pársování hašovací funkce, se nakonec ukázala být velice jednoduchá. Neboť hašovací metody mnou naprogramované používají různé hašovací funkce a součástí zadání bylo, aby tyto funkce bylo možné zadávat. Bylo tedy nutné vyřešit, jak tyto funkce zadávat a jak je potom vyhodnocovat, čili pársovat.

4.5.1 Myšlenka

Java od verze 6 (to je také důvod, proč je na spuštění programů nutná Java verze aspoň 6) totiž obsahuje velice zajímavou vlastnost a tou je takzvaný JavaScripting. Jde o interpretry různých skriptovacích jazyků, které jsou kompletně přepsány pomocí Javy. Díku tomu je možné v Javě volat kód těchto skriptovacích jazyků a také naopak ve skriptech používat třídy vytvořené v Javě. Vybral jsem si Javascript, neboť je přímo součásti Javové distribuce, nebylo tedy nutné zprovozňovat engine pro jiný skriptovací jazyk a našemu úkolů posloužil Javascript stejně dobře jako jakýkoliv jiný jazyk.

První vlastnost, totiž možnost volat v Javě kód některého skriptovacího jazyku, je pro tento úkol jako stvořená. Pokud je tedy hašovací funkce zadaná jako výraz v Javascriptové syntaxi, stačí z tohoto výrazu vytvořit Javascriptovou funkci a máme vyhráno. Toho docílíme tak, že vytvoříme funkci pomocí klíčového slova *function* s určitým počtem parametrů a zadaný výraz přiřadíme jako její návratovou hodnotu.

4.5.2 Implementace

Pro vyhodnocování hašovací funkce jsem v balíčku *utils* vytvořil třídu *Scripting*. Parametrem konstuktoru této třídy je výraz, který chceme vyhodnotit a počet parametrů, které jsou ve výrazu použity. Když vytváříme instanci této třídy, je výraz zabalen do Javascriptové funkce s odpovídajícím počtem parametrů. Potom konstuktor zkusí funkci zavolat s hodnotou všech parametrů rovnou jedné. Pokud všechno proběhne správně je výraz syntakticky i sémanticky správně, pokud ne konstuktor vyhodí výjimku.

Literatura

[1] Pokorný Jaroslav, Žemlička Michal: *Základy implementace souborů a databází*, Karolinum, Praha, 2004, 41–45 a 51–58.

[2] Cormack G. V.: *Practical perfect hashing*, School of Computer Science, McGill University, 1985.

[3] Larson P., Kajla A.: *File organization: Implementation of a method guaranteeing retrieval in one access*, ACM, 1984.

[4] Fagin R., Nievergelt J., Pippenger N., Strong H. R.: *Extendible hashing – A fast access method for dynamic files*, ACM, 1979.

[5] Litwin W.: *Linear Hashing: A new tool for file and table addressing*, IEEE, 1980.

Seznam programů

2.1 ACCESS (Cormack) 8
2.2 INSERT (Cormack) 9
2.3 GetAdr (Larson) 11
2.4 GetAdr (Litwin) 16
2.5 GetAdr(Skup) 19

Seznam obrázků

2.1 Schéma cormackova hašovaní 7
2.2 Faginovo hašování pro hloubku adresáře 2 12
2.3 Faginovo hašování po štěpení stránky 13
2.4 Faginovo hašování po štěpení adresáře 14
2.5 Schéma štěpení při lineárním hašování 15
2.6 Skupinové štěpení stránek - začátek algoritmu pro $s = 3$ a $g = 4$. 17
2.7 Skupinové štěpení stránek - po štěpení první skupiny. 17
2.8 Skupinové štěpení stránek - po rozštěpení všech skupin. . . . 18
2.9 Skupinové štěpení stránek - po první reorganizaci. 18

Dodatek A

Obsah CD

Na přiloženém CD se nacházejí tyto adresáře:

jar - spustitelné .jar archívy obsahující přeložené programy,

html - html stránky spouštějící jar archívy jako Java applety,

dokumentace - dokumentace ve formátu html vytvořená dokumentačním nástrojem Javadoc,

source - zdrojové soubory v jazyce Java,

bcprace - pdf verze této bakalářské práce,

pdf - pdf soubory popisující naprogramované hašovací metody.