VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

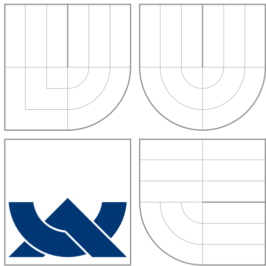
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

DATOVÁ STRUKTURA BLOOMŮV FILTR A JEJÍ VLASTNOSTI

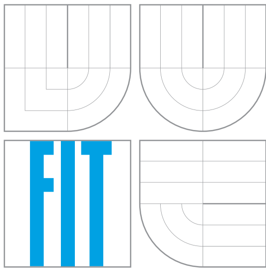
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE TOMÁŠ PROKOP
AUTHOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

DATOVÁ STRUKTURA BLOOMŮV FILTR A JEJÍ VLASTNOSTI

BLOOM FILTERS AND THEIR PROPERTIES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — TOMÁŠ PROKOP
AUTHOR

VEDOUCÍ PRÁCE — Ing. VIKTOR PUŠ
SUPERVISOR

BRNO 2010

Abstrakt

Bakalářská práce se zabývá popisem a konstrukcí Bloomova filtru. Autorem tohoto filtru je Burton H. Bloom. Bloomův filtr představuje efektivní nástroj ukládání prvků do univerzální množiny v podobě datové struktury. Zpracovává velký objem dat při zaplnění menšího paměťového prostoru. Datová struktura umožňuje vkládání prvků a jejich opětovné vyhledání v množině s nenulovou pravděpodobností chyby. Součástí práce je vysvětlení vlastností a způsobů využití datové struktury včetně možností snížení přípustných chyb. Rozšířením obecného Bloomova filtru je Počítaný Bloomův filtr, který umožňuje širší uplatnění této datové struktury.

Abstract

A dissertation is engaged in a description and a construction of the Bloom filter. The inventor of this filter is Burton H. Bloom. The Bloom filter represents an efficient tool for storing elements into the universal aggregate in the form of a data structure. It processes large volume of data while filling less of store space. The data structure makes it possible to insert elements and their time after time selecting in the aggregate with zero probability of errors. Part of the dissertation is explanation of properties and usage methods of the data structure including possibility of a reduction acceptable errors. An expansion of the Bloom filter is computed Bloom filter, which allows broader usage of the data structure.

Klíčová slova

Python, Bloomův filtr, Počítaný Bloomův filtr, Linux, hashovací funkce, knihovna, bitové pole

Keywords

Python, Bloom filters, Counting Bloom filters, Linux, hash funcions, liblary, bit array

Citace

Tomáš Prokop: Datová struktura Bloomův filtr
a její vlastnosti, bakalářská práce, Brno, FIT VUT v Brně, 2010

Datová struktura Bloomův filtr a její vlastnosti

Prohlášení

Čestně prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Viktora Puše. Také prohlašuji, že jsem uvedl všechny literární prameny, ze kterých bylo čerpáno.

....
Tomáš Prokop
16. května 2010

Poděkování

Tímto bych chtěl poděkovat Ing. Viktoru Puši za odbornou pomoc, za poskytnuté materiály a užitečné nápady při tvorbě bakalářské práce.

© Tomáš Prokop, 2010.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **3**

2 Teoretická část **5**
2.1 Hashovací funkce 5
2.2 Kryptografické hashovací funkce 7
2.3 Konvenční hashovací metody 8
2.3.1 Metoda 1 8
2.3.2 Metoda 2 9
2.4 Bloomův filtr 10
2.5 Pravděpodobnost chyby 11
2.6 Počítaný Bloomův filtr 14

3 Praktické využití Bloomova filtru **17**

4 Návrh řešení **18**
4.1 Zpracování parametrů 19
4.2 Kontrola činností programu 20
4.3 Zpracování vláken 20
4.3.1 Vlákno pro data z konzole 20
4.3.2 Vlákno pro zpracování dat 20
4.4 Zpracování Bloomova filtru a rozšíření 20
4.4.1 Vytvoření hashovací funkce 20
4.4.2 Vytvoření bitového pole 21

5 Implementace **22**
5.1 Modul `params` 22
5.2 Modul `debug` 24
5.3 Modul `start` 24
5.4 Modul `twoThread` 24
5.4.1 Vlákno pro data z konzole 25
5.4.2 Vlákno pro zpracování dat 25
5.5 Modul `bloom` 26
5.6 Modul `hashBitArray` 27

6 Experimenty a výsledky **29**
6.1 Testy pro Bloomův filtr 29
6.1.1 Test různého typu a počtu hashovacích funkcí 29
6.1.2 Test různě velkého bitového pole 33

6.2 Testy pro Počítaný Bloomův filtr 34
6.2.1 Test různého počtu hashovacích funkcí bez odebíráním dat a s ním . 34
6.2.2 Test různě velkého počítaného bitového pole 37
6.3 Souhrný závěr z experimentů 39

7 Závěr **40**

A Obsah CD **43**

B Manuál **44**

Kapitola 1

Úvod

V dnešní době je trend většinu informací získávat a ukládat v elektronické podobě do počítače. Elektronická data velmi rychle nabývají a neustále kladou větší nároky na množství fyzické paměti. Během ukládání je nutné vždy vyhledat vhodnou pozici (neboli volnou pozici) pro uložení malého či velkého objemu dat. Při procesu vyhledávání je zapotřebí rychlý přístup k uloženým informacím. Z tohoto důvodu si v dnešní době nemůžeme dovolit velké časové prodlevy, a proto existují snahy pro rychlejší ukládání a vyhledávání informací. Další podmínkou je: „Jak ideálně uspořit paměťový prostor, místo jeho neustálého rozšiřování?“. Toho lze dosáhnout za použití vhodných speciálních prostředků, které v informatice nazýváme „algoritmy“. Algoritmy slouží k dosažení snazšího ukládání a vyhledávání. Tyto problémy z větší míry řeší Bloomův filtr. Více v Kapitole 2.

Bloomův filtr byl poprvé uveřejněn v roce 1970 [1] B. H. Bloomem. Poté byl aplikován na mnoha místech s nedostatkem paměti. V současné době se uplatňuje pro nalezení slov v aplikacích, takzvaně jestli byly, nebo nebyly uloženy. Jako příklad můžeme uvést vyhledání v databázi (nebo firewall, streaming a mnoho dalších). Více v Kapitole 3 Praktické využití Bloomova filtru.

V Bloomově filtru se využívá bitové pole pro uložení dat a přístupu k nim soubor rozdílných hashovacích funkcí. Bloomův filtr neukládá fyzická data, ale nahrazuje je logickou jedničkou. To je příčina, proč se Bloomův filtr využívá všude tam, kde není potřeba mít fyzická data (fyzicky viditelná).

Struktura bakalářské práce

V Kapitole 2 je vysvětlen účel hashovací funkce, její základní vlastnosti, co nás nejvíce u těchto funkcí zajímá, předpoklady využití hashovacích funkcí (více v podkapitole 2.1 Hashovací funkce). Z praktického hlediska se hashovací funkce neuplatňují jen pro základní aspekty, ale také se využívají funkce se speciálními vlastnostmi, kryptografické hashovací funkce. V následující podkapitole 2.2 (Kryptografické hashovací funkce) jsou popsány základní vlastnosti a předpoklady použití těchto hashovacích funkcí. V průběhu hashování hashovacími funkcemi se tvoří pozitivní chyby. Existují jisté metody pro zohlednění, upravení a dokonce i částečné využití pozitivních chyb. Při výskytu těchto chyb nás zajímá matematický způsob výpočtu četnosti, pravděpodobnosti a důkazu chyby. Různé metody konvenčního hashování najdeme v podkapitole 2.3 (Konvenční hashovací metody). Informace ohledně hashovací funkce zjistíme v prvních třech podkapitolách Kapitoly 2. Další podkapitola se bude věnovat Bloomovu filtru, kde jsou hashovací funkce využity.

V podkapitole 2.4 (Bloomův filtr) nalezneme jakým způsobem se hashovací funkce využívají pro datovou strukturu Bloomův filtr. Podkapitole 2.5 (Pravděpodobnost chyby) pojednává o připustných chybách, kterými je Bloomův filtr zatížen. V poslední podkapitole 2.6 této části je popsána optimalizace na Počítaný Bloomův filtr, který dokáže kvalitnějšího výsledku jiným způsobem.

V Kapitole 3 bakalářské práce je uvedeno praktické využití Bloomova filtru. Zaměříme se zde zejména na síťové prostředí (firewall, streaming, databáze, atd.). Kapitola popisuje různé reálné postupy a metody pro využití Bloomova filtru.

Návrh řešení Bloomova filtru nalezneme v Kapitole 4. Podkapitola 4.1 (Zpracování parametrů) ukazuje jak správně navrhnout a zkonstruovat Bloomův filtr. Druhou částí této kapitoly je podkapitola 4.2 (Kontrola činnosti programu). Následující podkapitola 4.3 (Zpracování vláken), informuje o konstrukci a využití vláken. V poslední podkapitole 4.4 (Zpracování Bloomova filtru a rozšíření) je vytvořena knihovna, která informuje o způsobu návrhu obecného Bloomova filtru a následně i návrhu rozšíření na Počítaný Bloomův filtr.

Kapitola 5 vysvětluje popis a postup implementace do jednotlivých modulů. Moduly obsahují třídu ladící, třídy vláken, Bloomův filtr, Počítaný Bloomův filtr, hashovací funkce a bitové pole.

O všech implementovaných experimentech v podobě testů, dosažených výsledcích a celkovém vyhodnocení se dozvíme v Kapitole 6.

Kapitola 7 obsahuje Závěr, který je zaměřen na cíl bakalářské práce.

Kapitola 2

Teoretická část

Součástí této kapitoly je vysvětlení, jak lze jednoduchým způsobem dosáhnout implementace Bloomova filtru pomocí hashovacích funkcí, včetně pojmu hashovací funkce, významu, účelu i dalšího použití. V kapitole je kladen důraz na vysvětlení pozitivní (přípustné) chyby a vzorců pro její výpočet. Zde se budeme také zabývat metodami dosažení redukce pozitivních chyb. Obecný Bloomův filtr může být jednoduše upraven na rozšíření nazývané Počítaný Bloomův filtr. Na konci kapitoly se budeme více věnovat jeho vlastnostem a širšímu uplatnění.

2.1 Hashovací funkce

Úvodem

Pro pochopení Bloomova filtru bude velmi důležité porozumět jak funguje hashovací funkce a jaké má primární vlastnosti [1, 5, 8]. Hashovací funkce je základním stavebním prvkem Bloomova filtru, užívá se pro výpočet správného umístění logických jedniček v bitovém poli podle vypočteného hash kódu ze vstupních dat.

Vlastnosti

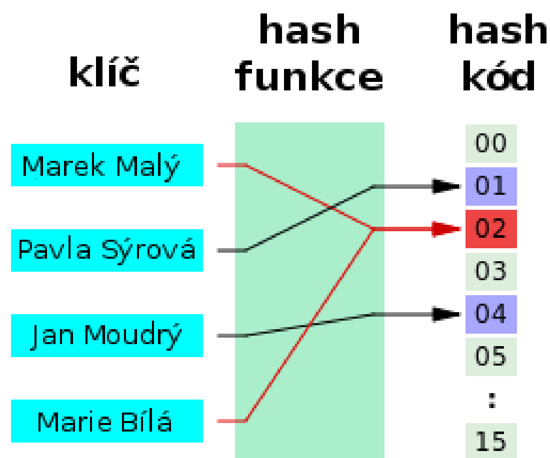
Hashovací funkce slouží pro převod jakýchkoliv vstupních dat do hash kódů, díky získanému matematickému výpočtu. Vypočtený hash kód, který získáme ze vstupních dat, obecně libovolné délky, skládající se například ze slov, je stejně dlouhý. Tento výstup šetří paměť a je vhodný pro další reprezentaci dat. Sebemenší změna vstupních dat se projeví s velkou změnou hash kódu. Po zahashování vstupních dat již není možné z výstupních hash kódů získat původní data zpátky.

Při zahashování vstupních dat x_1 není pravděpodobné, že by jiná data x_2 zpracovaná přes totožnou funkci měla stejný hash kód. Pokud data x_1 a x_2 získají naprosto shodný výsledek z hashovací funkce, označíme tento pojem kolize. Příkladem kvality hashovací funkce je pravděpodobnost náchylnosti na kolizi, s jejím růstem zjišťujeme nekvalitnost hashovací funkce. Obrázek 2.1 znázorňuje kolizi.

Účel hashovací funkce

Do základního využití rozhodně patří nalezení hodnoty podle klíče v databázi, kdy hashovací funkce se využívají v hashovacích (rozptylových) tabulkách. V tabulkách zkoumáme asociativní pole (abstraktní datový typ složený z kolekce unikátních klíčů) a dynamické sady (soubor údajů, který je dynamicky spojen s databází), proto aby byly záznamy velmi

rychle nalezeny, tak jako ve slovnících. Obrázek 2.1 ukazuje výpočet hashovací funkce, kde hash kód k se přiřazuje do hashovací tabulky.



Obrázek 2.1: *Hashovací funkce, která mapuje jména na celá čísla* $0-15$, *ukazuje kolizi mezi jmény „John Smith" a „Sandra Dee". Zdroj [12].*

Formální popis

Hashovací funkce generuje pseudonáhodný hash kód (číslo), který budeme nazývat k. Hash kód k musí být v rozsahu $(0 \leqslant k \leqslant h - 1)$.

Konvenční hashovací tabulka

Hashovací tabulka potom používá hashovací funkci k ukládání. Poslední buňka v hashovací tabulce musí ukazovat na *null*. Pokud daná buňka bude prázdná, tak se uloží *k-tý* hash kód, ale pokud je tato buňka zaplněna, tak hashovací funkce musí generovat nové k dokud nebude odpovídající buňka prázdná, pak se uloží hash kód k.

Pro ověření jestli vstupní data patří do vybraných buněk nebo nepatří, použijeme stejnou hashovací funkci. Pokud při testovaní hash kódu nalezneme obsazenou buňku, pak splňuje podmínky ověření, v opačném případě podmínky nesplňuje. Více v [1].

Za předpokladu uložení n vstupních dat, z nichž se každé skládá z b bitové délky. Probíhá vkládání dat n na hashovací oblast o N bitech organizovaných do buněk pro uložení $b+1$ bitové délky. Daná hashovací funkce vypočte hash kód k, kam umístí data n. Jeden bit buňky tabulky slouží pro kontrolu prázdnosti nebo obsazenosti. Buňky v tabulce musejí být určeny poměrem $h > n$.

Faktor ϕ představuje reprezentaci prázdných buněk, kterému odpovídá rovnice

$$\phi = \frac{h-n}{h} = \frac{N - n \cdot (b+1)}{N}. \qquad (2.1)$$

Pro hashovací oblast N platí

$$N = \frac{n \cdot (b+1)}{1-\phi}. \qquad (2.2)$$

Podíl buněk, které jsou prázdné, značí ϕ. Pravděpodobnost prázdné buňky je ϕ. Přístup do neprázdné buňky je rozdíl pravděpodobností $(1 - \phi)$. Pokud je buňka neprázdná, tak se procedura vyhledání prázdné buňky zopakuje.

Specializované hashovací funkce se využívají k vyhledávání míst na mapě podle hash kódu (klíče) k, který určuje hledanou pozici na mapě. K dalším použití patří určení zda jsou dva objekty identické, nebo pro kontrolní součty s velkým množstvím dat takzvaně CRC. Více v [11, 12]

2.2 Kryptografické hashovací funkce

Hashovací funkce neslouží jen k obvyklému použití. Užívají se i Kryptografické hashovací funkce, které mají velmi užitečné vlastnosti v zabezpečení autentizace uživatelů do různých systému, například přihlášení běžného počítače, zabezpečení přihlášení a připojení na server, zabezpečení zpráv, atd.. Nejvíce se ale Kryptografické hashovacích funkce používají k zabezpečení hesla uživatelů na daném počítačovém prostředí.

Citace z [14]:

> Nejdůležitější je následující trojice vlastností, která určuje obtížnost napadení hashovací funkce. Obtížností se v tomto kontextu myslí výpočetní složitost, která by měla být za současných technologických možností mimo možnosti reálného použití:
>
> - Odolnost vůči získání předlohy. Pro daný hash c je obtížné spočítat x takové, že $h(x) = c$ (hashovací funkce je jednosměrná.)
> - Odolnost vůči získání jiné předlohy. Pro daný vstup x je obtížné spočítat y takové, že $h(x) = h(y)$.
> - Odolnost vůči nalezení kolize. Je obtížné systematicky najít dvojici vstupů (x, y), pro které $h(x) = h(y)$.
>
> Další obvyklé požadavky zahrnují:
>
> - Nekorelovatelnost vstupních a výstupních bitů, kvůli znemožnění statistické kryptoanalýzy.
> - Odolnost vůči skoro-kolizím. Je obtížné nalézt x a y taková, že $h(x)$ a $h(y)$ se liší jen v malém počtu bitů.
> - Lokální odolnost vůči získání předlohy. Je obtížné najít i jen část vstupu x ze znalosti $h(x)$.

Některé šifrovací metodiky:

Message-Digest známé pod názvy MD4, MD5 obě tyto kryptografické hashhovací funkce jsou už prolomeny.

Secure Hash Algorithm známe SHA-0, SHA-1 jsou také prolomené, ale SHA-2 z množiny 4 hashovacích funkcí (SHA-224, SHA-256, SHA-384 a SHA-512) obsažené ve standardu FIPS 180-2 nejsou dodnes prolomeny.

Tiger hashovací funkce využití v peer-to-peer programech.

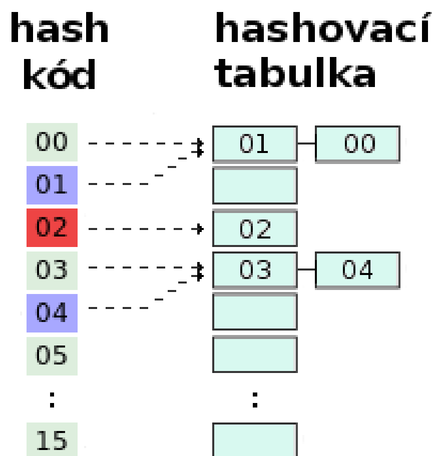
Více v [5, 8].

2.3 Konvenční hashovací metody

2.3.1 Metoda 1

Úvodem

Metoda 1 je odvozena od přirozených konvenčních hashovacích metod. Pro zpřehlednění tématu vyjdeme z obrázku 2.1, kde hashovací funkce podle klíče vytvoří hash kód. Tato metoda neprovádí nic jiného než podle hash kódu najde prázdnou buňku v hashovací tabulce a do dané buňky uloží onen hash kód, jak je vidět na obrázku 2.2.



Obrázek 2.2: *Do hashovací tabulky se ukládají položky hash kódu a podle nich se hledá volná pozice v tabulce.*

Formální popis

Hashovací prostor tvořící N' bitů opět organizovaný do buněk h, kde buňky o velikosti c bitů uchovávají pouze hash kód k, ale ne celá vstupní data.

V této metodě ϕ' značí podíl prázdných buněk

$$\phi' = \frac{N' - n \cdot c}{N'}. \tag{2.3}$$

$$N' = \frac{n \cdot c}{1 - \phi'}. \tag{2.4}$$

$$T' = \left(\frac{3}{\phi'}\right) - 2. \tag{2.5}$$

$$N' = n \cdot c \cdot \frac{T' + 2}{T' - 1}. \tag{2.6}$$

Hashovací funkce vypočítá hash kód k a jeho velikost závisí na pravděpodobnosti chyb. Velikost buněk h se zvýší, když pravděpodobnost chyb se sníží. Jestliže pravděpodobnost je dostatečně malá (asi 2^{-b}), tak buňky se stanou velké natolik, aby uchovaly vstupní data celá. Výsledkem je konečný výpočet bez chyb.

P' představuje pravděpodobnost chyb, kde se předpokládá $1 \gg P \gg 2^{-b}$. Velikost buněk nám určuje poměr $c < b$ a posléze vybere očekávanou pravděpodobnost, která bude v dostatečném okolí P', ale menší než P'. Všechna data budou zakódována v *c-bitovém* kódu, který není nezbytně jedinečný, z tohoto důvodu mohou vzniknout kolize. Potom tyto kódy budou uloženy a testovány podobným způsobem jako u konvenční hashovací tabulky s pravděpodobností daných chyb.

Očekávaná pravděpodobnost chyb P' testovaných dat, které byly chybně označeny, je

$$P' = \frac{\left(\frac{1}{2}\right)^{c-1}}{\phi'}. \tag{2.7}$$

$$c = -\log_2 P' + 1 + \log_2 \frac{T' + 2}{3}. \tag{2.8}$$

2.3.2 Metoda 2

Úvodem

Metoda 2 má odlišný koncept uspořádání hashovacích prostor buněk. Pro zpřehlednění tématu vyjdeme opět z obrázku 2.1, kde hashovací funkce podle klíče vytvoří hash kód. Předpokládá se, že všechny bity v hashovacím prostoru jsou nejprve nastaveny na 0. Metoda vyhledá volnou buňku v hashovací tabulce podle hash kódu. Do buňky neukládá hash kód ale jen bitovou 1, jak je vidět na obrázku 2.3.

Formální popis

Hashovací adresní prostor je složen z N'' bitů, s adresami od 0 do $N'' - 1$. Všechna data v souboru, která mají být uložena, jsou směrována hash kódem k na různý počet bitových adres $a_1, a_2, \ldots, a_d$. Všechny d bity řešené prostřednictvím a_1 až a_d jsou nastaveny na 1. Očekávaný podíl bitů v hashovacím prostoru N'' bitů představuje ϕ'', pro které platí

$$\phi'' = \left(1 - \frac{d}{N''}\right)^n. \tag{2.9}$$

Při testování nového vstupního data s posloupností d bitů adresy $a'_1, a'_2, \ldots, a'_d$ jsou data generována stejným způsobem jako při ukládání. Za podmínky nastavení všech d bitů na 1 jsou data obsažena. Je-li některý z d bitů v 0, hashovací prostor data nevlastní.

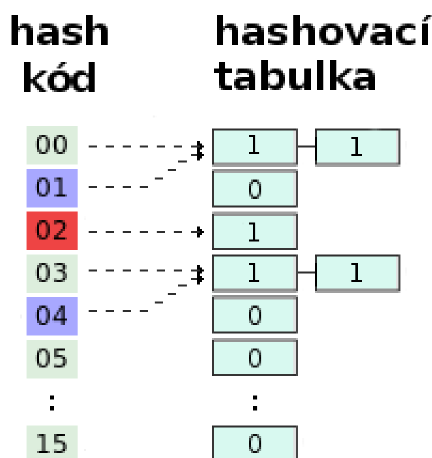
Data nebudou přidána do pole, budou-li chybně akceptována z důvodu nastavení všech bitů d na 1. Očekávaná pravděpodobnost chyb testovaných dat je

$$P'' = (1 - \phi'')^d. \tag{2.10}$$

Za předpokladu, že $d \ll N''$ přidáme $\log_2$ na obě strany rovnice (2.10) a získáme

$$\begin{aligned} \log_2 \phi'' &= \log_e \left(1 - \frac{d}{N''}\right)^n \cdot \log_2 e, \\ &= -n \cdot \left(\frac{d}{N''}\right)^n \cdot \log_2 e. \end{aligned} \tag{2.11}$$

$$N'' = n \cdot \frac{d}{P''} \cdot \frac{\log_2 e}{\log_2 \phi'' \cdot \log_2 (1 - \phi'')}. \tag{2.12}$$



Obrázek 2.3: *Hashovací tabulka, do které se ukládají bitové jedničky podle hash kódu, je vytvořená z klíčových vstupních dat.*

2.4 Bloomův filtr

Úvodem

Bloomův filtr je datová struktura umožnující velmi rychlé vkládání dat. V roce 1970 [1] byl zveřejněn Bloomův filtr autorem B. H. Bloomem. V Bloomově filtru neexistuje žádný způsob získání dat po jejich uložení. Máme zde ale možnost zjistit zda data byla do datové struktury uložena a to procesem vyhledání. Na mnoha místech v počítačovém prostředí postačí takto omezená aplikace.

Vkládání dat do datové struktury probíhá přes k odlišných a nezávislých hashovacích funkcí. Matematický výpočet hashovacích funkcí je ukazatel do bitového pole. Bity v poli se nastavují na logické jedničky.

Při vyhledání dat probíhá podobný postup. Ze stejných hashovacích funkcí se vypočítá ukazatel k do bitového pole. Dále na bitech v poli spočívá potvrzení požadavku vyhledání dat. Pokud data byla uložena, musejí být všechny bity v logické „1“. V případě, že bude jediný bit v logické „0“, data se neshodují a nebyla nikdy uložena.

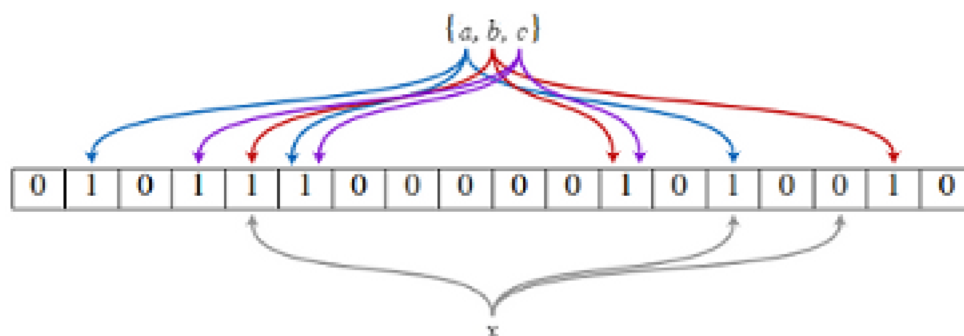
Formální popis
Univerzum U se skládá z N prvků a jeho velikost není stanovena. Nechť máme vektor v s velikosti m bitů pro Bloomův filtr [1, 3, 6, 10]. Vektor v se vztahuje k podskupině $A = \{a_1, a_2, \ldots, a_n\}$ z n možných prvků na univerzum U. Bloomův filtr je užitečný pouze v případě, že velikost $U \gg A$.

Z počátku je potřeba vektor v nastavit na 0. A nechť je dán soubor hashovacích funkcí $H = \{h_1, h_2, \ldots, h_k\}$, kde k označuje počet různých nezávislých hashovacích funkcí $h_1, h_2, \ldots, h_k$ každá s oborem hodnot $(1, \ldots, m)$.

Operace vkládání, kdy každý prvek a náleží množině A, představuje bity označené na pozicích $h_1(a), h_2(a) \ldots, h_k(a)$, kde jsou bity nastaveny na hodnotu 1. Obrázek 2.4 po výpočtu hashovacích funkcí ukazuje nastavení bitu jednou nebo vícekrát na hodnotu 1.

Účelem operace vyhledání je ověření, zda existující prvek např.: b univerza U patří do množiny A. Operace provede kontrolu nastavení všech bitů na 1 po zahashování z k funkcí $h_1(b), h_2(b), \ldots,$až $h_k(b)$. Je-li alespoň jeden bit nastaven na 0, jsme si jistí, že b nepatří do A (viz. obrázek 2.4 prvek x).

Pokud tedy všechny bity $H = \{h_1(b), h_2(b), \ldots, h_k(b)\}$ jsou nastaveny na 1, pro prvek b prohlásíme, že patří do množiny A.



Obrázek 2.4: *Ukázka Bloomůva filtru, kde vstupní data představují $\{a, b, c\}$, která projdou přes hashovací funkce h_1, h_2, h_3. Barevné šipky zobrazují pozice v bitové poli, každý prvek je mapován na* 1. *Prvek x značí test jestli v bitovém poli bude nebo nebude nalezen. Prvek x v bitovém poli o velikosti $m = 18$ a počtu hashovacích funkcí $k = 3$ není obsažen, protože u jednoho bitu daná pozice má* 0. *Zdroj [13].*

2.5 Pravděpodobnost chyby

Úvodem
Pozitivní chyby (nebo-li pravděpodobnost chyb, kterou si vysvětlíme později) v Bloomově filtru jsou nalezené při operaci vyhledání, když různé nezávislé hashovací funkce vypočítají ukazatel z dat na všechny neprázdné pozice v bitovém poli. Jednoduchým testem podle obrázku 2.5 zjistíme, jak lze tohoto stavu dosáhnout. Chyba se zdá být závažná, ale není

tomu tak. Operace vyhledání nikdy neřekne chybně, že zdrojová data nejsou obsažena, když byla uložena do datové struktury.

Vypočítaný stav \ Skutečný stav	Ano	Ne
Ano	OK	fp
Ne		OK

Obrázek 2.5: *Skutečný stav naznačuje opravdový obsah Bloomova filtru. Vypočítaný stav značí operaci vyhledání, kdy se musí vypočítat z testovaných dat hash kód k, který určuje umístění dat. Pokud se oba stavy shodují v Ano, testovaná data jsou opravdu v Bloomově filtru. Opačný stav, pro obě Ne, je také v pořádku. Data nejsou v Bloomově filtru a ani je nikdy neobsahoval. Stav f_p označuje výsledek, kdy operace vyhledání řekne, že data Bloomův filtr obsahuje, ale ve skutečnosti nebyla nikdy uložena. Takový stav je povolen a stanoví pravděpodobnost chyby. Do posledního proškrtnutého stavu se Bloomův filtr nemůže nikdy dostat díky jeho konstrukci.*

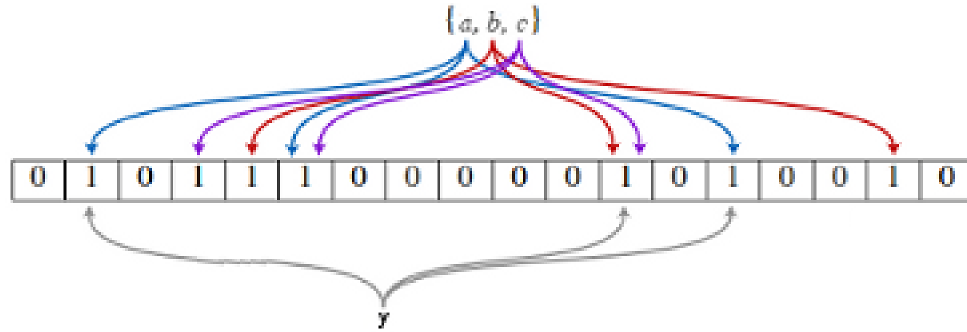
Formální popis

Operace vyhledání provede kontrolu m bitů pro prvek y pomocí různých nezávislých hashovacích funkcí k. Ukazatelé z k hashovacích funkcí musejí ukázat na všechny bity m nastavené na 1. Je-li ověřena existence prvku y z univerza U, pak prvek patří do množiny A (viz obrázek 2.6 prvek y). Ověřením skutečných dat v Bloomově filtru můžeme zjistit, že prvek y pravděpodobně nepatří do A. V důsledku toho vzniká riziko pozitivních chyb, anglicky nazývané „false positives“, na které Bloomův filtr odpovídá kladně, a tyto chyby patří do univerza U.

Pozitivní chybový rozsah můžeme alternativně definovat jako pravděpodobnost, že daný prvek nepatřící do množiny A a Bloomovým filtrem je chybně označen za obsažený. Při existenci pozitivní chyby podílu f_P, která značí skutečnost stejné hodnoty, pak pravděpodobnostní chybový rozsah označíme obdobnou notací f_P.

Soubor těchto chyb, který informuje o nesprávném obsahu Bloomůva filtru, označíme F_P. Pravděpodobnost chyb vyjadřuje definice f_P jako poměr počtu prvků $U - A$, které dávají kladnou odpověď na celkový počet prvků $U - A$

$$f_P = \frac{|F_P|}{|U - A|}. \tag{2.13}$$



Obrázek 2.6: *Vstupní data* $\{a, b, c\}$*, která projdou přes různé nezávislé hashovací funkce* h_1, h_2, h_3*, v Bloomově filtru jsou skutečně označena. Barevné šipky zobrazují pozice v bitovém poli, každý prvek je mapován na 1. Prvek* y *značí test jestli v bitovém poli bude prvek nalezen nebo ne. Operace vyhledání ověřuje označení všech pozic* $k = 3$ *na 1, které jsou zde označeny, tím je potvrzena existence prvku* y *v bitovém poli. V tomto případě nebyl prvek* y *nikdy do datové struktury vložen a jedná se o pravděpodobnost chyby při operaci vyhledání.*

Všechny hashovací funkce mapují každou položku univerza U na náhodné rovnoměrné rozložení v rozmezí $(1, \ldots, m)$. K určení pravděpodobnosti, zda je konkrétní bit nastaven na hodnotu 1, použijeme jednu hashovací funkci a jeden prvek z množiny A, pak pravděpodobnost je $\frac{1}{m}$. Pokud je specifický bit ponechán na 0, je pravděpodobnost $1 - \frac{1}{m}$. Bloomův filtr kóduje všechny prvky z množiny A s pravděpodobností pro bit roven nule, pak je

$$p_0 = \left(1 - \frac{1}{m}\right)^{kn}. \tag{2.14}$$

Čím víc vzroste m, tím víc se zlomek $\frac{1}{m}$ blíží k nule, pak je možné p_0 aproximovat

$$p_0 \approx e^{-\frac{kn}{m}}. \tag{2.15}$$

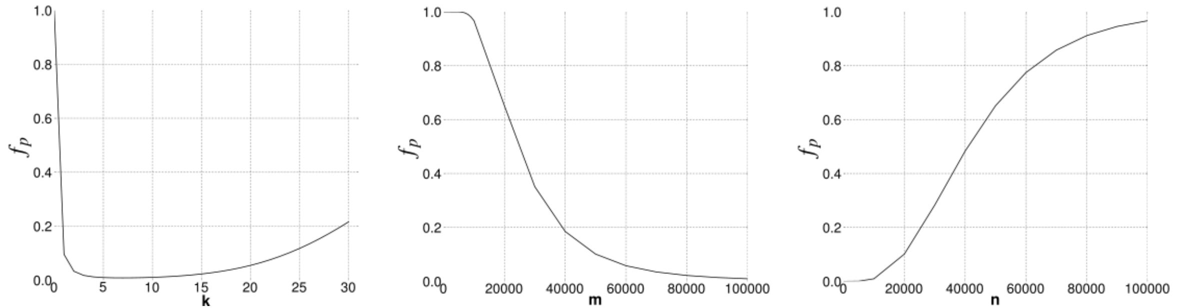
Pravděpodobnost, že konkrétní bit je nastaven na 1 může být vyjádřena jako

$$p_1 = 1 - p_0. \tag{2.16}$$

Pro pravděpodobnost chyb lze odhadnout, že každá z pozic k, vypočítaná ze souboru hashovacích funkcí, bude v 1. f_P je pak dáno

$$\begin{aligned} f_P &= p_1^k, \\ &= \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k, \\ &\approx \left(1 - e^{-\frac{kn}{m}}\right). \end{aligned} \tag{2.17}$$

Pravděpodobnost chyb f_P určují tři parametry v obrázku 2.7, n značí velikost podmnožiny A, m velikost filtru a k počet hashovacích funkcí. Pokud parametr m roste je f_P klesající funkce, s růstem n funkce f_P také roste. Zpočátku pro rostoucí k funkce f_P prudce klesá a dosahuje minima, až později se zvětšuje.



Obrázek 2.7: *Znázornění pravděpodobnosti f_p pomocí prametrů k, m a n. Zdroj [6].*

$$k = \frac{m \ln 2}{n}. \tag{2.18}$$

Při hashování dat do Bloomova filtru záleží na počtu hashovacích funkcí k. Čím více funkcí máme k dispozici, tím větší je šance najít bit 0, ale s velkým počtem funkcí vzrůstá obsazenost v bitovém poli. V důsledku toho záleží na vyváženosti velkého a malého počtu hashovacích funkcí.

f_P s ohledem na k je 0, kde k z rovnice (2.18) dokazuje globální minimum. Nejmenší pravděpodobnost chyb f_P určují dané hodnoty m a n v rovnici (2.19). Optimální počet hashovacích funkcí je dán pravděpodobností chyb, které musí být $\leqslant \frac{1}{2}$. Pak pravděpodobnost aproximace chyb je dána v poslední části rovnice (2.19).

$$f_P = \left(\frac{1}{2}\right)^{\frac{m \ln 2}{n}} \approx (0.6185)^{\frac{m}{n}}. \tag{2.19}$$

2.6 Počítaný Bloomův filtr

Úvodem

Základní vlastností Bloomova filtru je, že není možné odstranit uložený prvek. Smazání konkrétního vstup dat vyžaduje, aby odpovídající m bit v bitovém poli být nastaven na nulu, ale tato změna může porušit bity ostatních prvků zahashovaných ve filtru. Rozhodující otázkou bylo vyřešení tohoto problému, až nápad Počítaný Bloomův filtr anglicky zvaný „Counting Bloom filter“ navržený ve zdroji [3] tento problém objasnil.

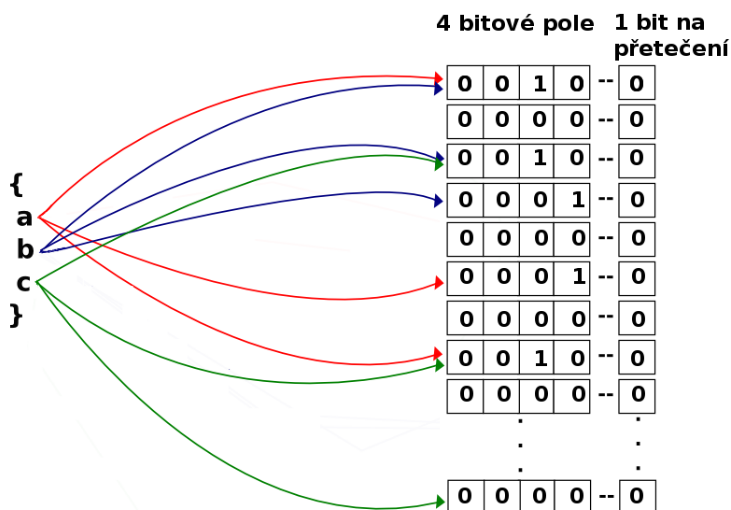
V Počítaném Bloomově filtru můžeme jak přidávat vstupní data, tak do určitého množství vložená data odebírat. Počítaný Bloomův filtr udržuje vektor čítačů, kterému odpovídá každý bit v bitovém poli. Kdykoliv je prvek přidán nebo odstraněn z filtru, čítač odpovídajícímu hash kódu k hodnotu zvýší nebo sníží. V čítači se změní hodnota z nuly do jedné a odpovídající bit v bitovém poli je nastaven. Jestliže se čítač změní z jedné do nuly, tak je vyloučen odpovídající bit v bitovém poli. Je podstatné si uvědomit, že čítače se změní pouze při přidávání a odebírání dat v Bloomově filtru. Hodnota čítačů musí být nastavena taková, aby nepřetekla, jinak by mohlo při změně na koncovou nulu dojít k porušení ostatních prvků v Počítaném Bloomově filtru.

Pro aplikace s nepovoleným přístupem do sítě se Počítaný Bloomův filtr velmi hodí, protože samotný proces by musel kontrolovat velké množství dat pro udržení bezpečnosti. Udržuje v sobě hashované záznamy, které se kontrolují s dalšími daty pomocí operace vyhledání. Data mohou být vyloučena pokud ověříme, že nepatří do zkoumaných hodnot Počítaného Bloomova filtru.

Formální popis

Obecný Bloomův filtr nalezneme v podkapitole 2.3 nebo v podrobnějších zdrojích [3, 6, 10]. Bloomův filtr tedy zastupuje vektor v o velikosti l bitů. Vektor v se vztahuje k podskupině $A = \{a_1, a_2, \dots, a_n\}$ o n prvcích z univerza U skládajících se z N prvků. Počítaný Bloomův filtr dále obsahuje bitové pole $B[1], \dots, B[N]$.

Z počátku je potřeba vektor v nastavit na 0. Nechť je dán soubor hashovacích funkcí $H = \{h_1, h_2, \dots, h_k\}$, kde se počet různých nezávislých hashovacích funkcí $h_1, h_2, \dots, h_k$ označuje k, každá funkce je s oborem hodnot $(1, \dots, l)$. Každý prvek $x \in v$, bity $B[h_i(x)]$ jsou nastaveny na hodnotu 1 pro $1 \leqslant i \leqslant k$. (Hodnota x v $B[h_i(x)]$ muže být nastavena na 1 vícekrát a tímto způsobem se vytváří Počítaný Bloomův filtr.) Pro větší přehlednost nám poslouží obrázek 2.8.



Obrázek 2.8: *Znázornění Počítaného Bloomova filtru. Kde vstupní položky $\{a, b, c\}$ se pomocí 3 různých nezávislích hashovacích funkcí zahashují do 4 bitového pole. Když nastane, že dané položky a, b jsou směrovány na shodnou pozici, tak se navýší 4 bitové pole o 1. Poslední bit označuje přetečení, kdyby se určitá část pole naplnila, změníme hodnotu bitu na 1. Bit přetečení slouží i pro značeni stavu při odstranění položek z 4 bitového pole. V případě kdy se v 4 bitovém poli nachází jediný prvek s bitem přetečení, pak ho z důvodu závislosti na ostatních prvcích v poli, nesmíme odstranit.*

Smazání prvku v obecném Bloomově filtru není možné pouhým nastavením bitů do 0, jinak by došlo ke konfrontaci s dalšími prvky a následně uložené změny by je poškodili. Aby bylo umožněno mazaní prvků, musíme provést jejich detekci. Povýšíme Bloomův filtr na Počítaný Bloomův filtr (PBF), který používá pole čítačů C místo jednorozměrného bitového pole. Bitové pole $B[1], \dots, B[N]$ se rozšíří na dvourozměrné $B[1][1], \dots, B[N][C]$, kde čítače C obsahují rozsah o $(1, \dots, m)$, kde m označuje počet čítačů. (Čítače sledují počet prvků v Bloomově filtru.) Mazání je nyní možné provést bezpečně dekrementací

příslušného čítače. Čítač musí být zvolen dostatečně velký, aby se zabránilo přetečení, nebo se musí vyhradit speciální bit na detekci přetečení. Ve většině aplikací postačí čtyři bity na čítač [2].

Správnou velikost čítačů zjistíme u (PBF) s m čítači pro soubor s n prvky hashovacími funkcemi k následující rovnicí. Počet spojení s i-tým čítačem značí $c(i)$. Pravděpodobnost $\mathbb{P}$, že i-tý čítač je inkrementován j-té krát, je doba binomické náhodné proměnné

$$\mathbb{P}\big(c(i) = j\big) = \binom{nk}{j}\left(\frac{1}{m}\right)^j\left(1 - \frac{1}{m}\right)^{nk-j}. \tag{2.20}$$

Pravděpodobnost, že každý čítač je nejméně j, se váže ohraničením shora k $m\mathbb{P}\big(c(i) \geq j\big)$, kterou lze vypočítat podle výše uvedeného vzorce. Ztráta může být také vázána výpočtem

$$\mathbb{P}\big(c(i) \geq j\big) \leq \binom{nk}{j}\frac{1}{m^j} \leq \left(\frac{enk}{jm}\right)^j. \tag{2.21}$$

Předpokládáme, že omezení je na $k \leq (\ln 2)m/n$. Pak můžeme optimalizovat pravděpodobnost chyb s $k = (\ln 2)m/n$. Platí

$$\mathbb{P}\big(\max_i c(i) \geq j\big) \leq m\left(\frac{e \ln 2}{j}\right)^j. \tag{2.22}$$

Pokud bychom použily 4 bity na čítač, pak přeteče jen tehdy, pokud dosáhne hodnoty 16. Z výše uvedeného pak vyplývá

$$\mathbb{P}\big(\max_i c(i) \geq 16\big) \leq 1.37 \times 10^{-15} \times m. \tag{2.23}$$

Během přetečení s největší pravděpodobností $1,37 \times 10 - 15 \times mt$ platí pro všechny soubory nejvýše n položek, že výčtový typ bude vázány na (PBF), který reprezentuje t různých souborů nejvýše n položek. To stačí pro většinu aplikací.

Kapitola 3

Praktické využití Bloomova filtru

Bloomův filtr se využívá v databázích, síťových aplikacích a mnoha dalších aplikacích. Dále se uplatňuje všude tam, kde je potřeba co nejmenší zátěž na paměť a není nutné mít vstupní data přímo uložena. Druhy využití Bloomova filtru v praxi jsou uvedeny v jednotlivých blocích.

Google Bigtable používá Bloomovy filtry k snížení vyhledávání neexistujících řádků nebo sloupců na disku. Zabraňuje nákladnému vyhledávání na disku a výrazně zvyšuje výkon dotazů při provozu databáze. Bigtable je distribuovaný úložný systém pro správu strukturovaných dat. Obsahuje přes tisíc komoditních serverů k uložení velkého množství dat. Využívá mnoho aplikací pro ukládání dat, např.: web indexování, Google Earth a Google Finance. Bigtable je flexibilní z hlediska velikosti dat a na latenci požadavků v real-time zpracování. Aplikace znázorňuje vysoce výkonné řešení pro produkty Google. Zdroj [4].

Venti archivační systémy pro ukládání dat využívají Bloomův filtr na detekci dříve uložených dat. Venti je blokový úložný server určený pro archivaci dat. Obsah tohoto systému s unikátním SHA1 hash blokem se chová jako blokový identifikátor pro operace čtení a zápis. Kromě předcházení zničení dat, systém kontroluje duplicitní kopie bloku, čímž se snižuje potřeba skladování dat a zjednodušují operace prováděné klienty. Venti je stavebním kamenem pro výstavbu různých skladovacích aplikací, jako logické zálohování, fyzické zálohování a snapshot souborové systémy. Zdroj [9].

Síťové aplikace, kde Bloomův Filtr je jednoduchý prostorově účinný v náhodné datové struktuře, představuje soubor vytvořený za účelem podpory mezi dotazy. Bloomovy filtry dovolují pozitivní chyby s určitou pravděpodobností, ale nad touto nevýhodou často vítězí snížení paměťového prostoru. V posledních letech se Bloomovy filtry stávají populární v síťové literatuře. Síťové aplikace pomáhají v celé řadě problémů. Cílem je zajistit jednotnou matematickou a praktickou konstrukci za účelem porozumění jim a jejich použití v praxi. Zdroj [3].

Kapitola 4

Návrh řešení

V části návrhu se zaměřím na praktické řešení aplikace „Datová struktura Bloomův filtr a její vlastnosti“. Aplikace umožňuje zpracovávání velkého množství dat při malé paměťové náročnosti. Je potřeba dosáhnout dvou povinných částí, první zkonstruování Bloomova filtru podle podkapitoly 2.3 a za druhé jeho rozšíření. Toto rozšíření budu směřovat k Počítanému Bloomově filtru, jak je uvedeno v podkapitole 2.6.

Jeden z prvních požadavků na aplikaci je jednoduché zpracování parametrů z příkazového řádku. Obsah nalezneme v podkapitole 4.1. Již od počátku programování předpokládám práci s konzolovou verzí. K tomuto mínění se přikláním z důvodu celé konstrukce, nýbrž hlavním cílem je vytvořit knihovnu pro Bloomův filtr a pro Počítaný Bloomův filtr. Knihovna bude ztvárněna v podobě konzolové aplikace, i když by se dalo vytvořit řešení rozsáhlejší.

K dosažení jednoduché konstrukce a průběžné kontroly v průběhu chodu programu je zapotřebí vytvořit modul kategorie ladění více v 4.2. Díky této části bude nejen přehledný chod aplikace, ale i budoucí uživatelé nebo programátoři rychleji pochopí její ovládání a konstrukci.

Základní myšlenkou je použití vláken pro průběžné zpracovávání vstupních dat při chodu aplikace. Určitě se naskýtá několik možností jak přesně využít vlákna, a proto podrobnější informace o těchto možnostech nalezneme v podkapitole 4.3.

Vlastní ztvárnění „Datová struktura Bloomův filtr a její vlastnosti“ je součástí poslední části 4.4 Kapitole 4. Zde se budu zabývat celkovou konstrukcí knihovny pro Bloomův filtr a Počítaný Bloomův filtr. Konečné řešení budu dělit na menší logicky související částí, které jsem nazval *bloky*. Do budoucna bude snadnější a přijatelnější knihovnu rozšířit.

Popis diagramu

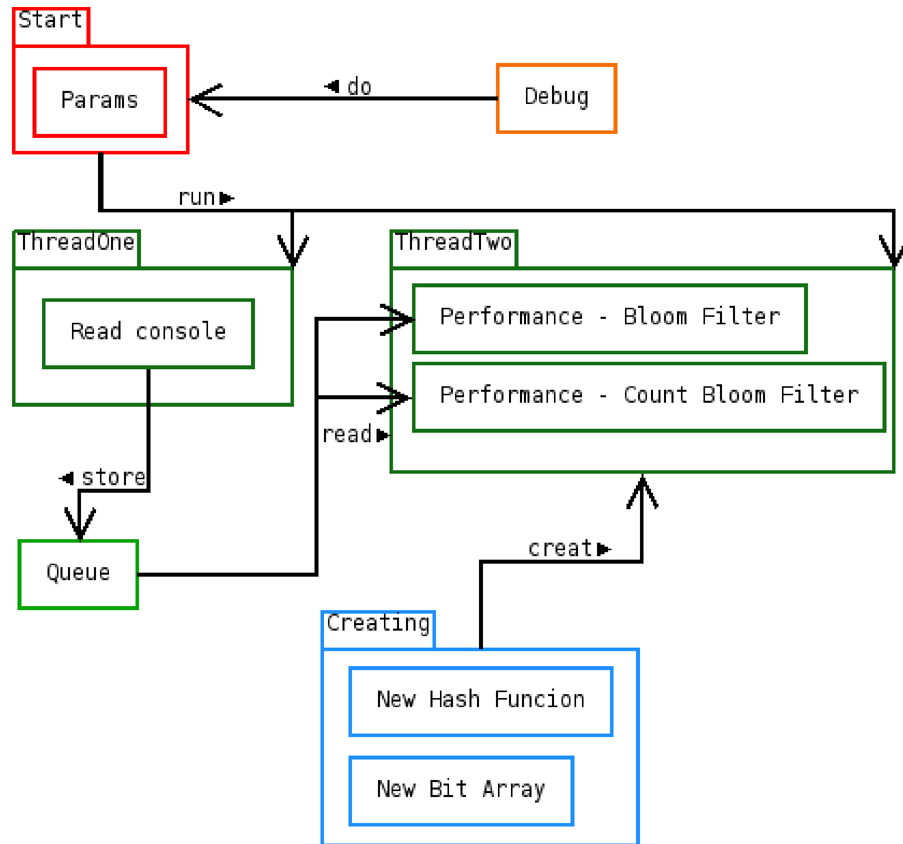
Celý diagram 4.1 návrhu řešení se skládá ze 4 základních bloků (neboli logických částí) zjednodušujících aplikaci.

Blok `Start` zajišťuje spuštění aplikace podle zadaných vlastností v parametrech. Obsluhu parametrů řídí podblok `Params`, který obstarává načítání parametrů přímo z konzole.

Blok `Debug` navazuje na blok `Start`. Po zadání parametru uživatelem blok `Debug` provádí potřebný výpis infomací podle hodnoty jeho argumentu. Dále je blok využit i v níže položených větvích navrženého diagramu.

Jádrem celé aplikace jsou bloky `ThreadOne` a `ThreadTwo`, které jsou znázorněny v podobě vzájemně komunikujících vláken, jak je vidět v diagramu 4.1. Oba bloky využívají blok `Queue` neboli frontu. Bloku `ThreadOne` slouží fronta k ukládání vstupních dat. Blok `ThreadTwo` čte z fronty data, která rovnou zpracovává dle zadaných parametrů. Sou-

částí bloku `ThreadTwo` jsou dva podbloky `Performance - Bloom filter` a `Performance - Count Bloom filter`. Podbloky tvoří samotnou knihovnu Bloomův filtr a Počítaný Bloomův filtr. Ke knihovně patří oddělený blok `Creating`, který vytváří novou hashovací funkci podle typu parametru od uživatele a nové bitové pole.



Obrázek 4.1: *Návrh celé aplikace, kde diagram znázorňuje jednotlivé logicky související části, s kterými se bude pracovat.*

4.1 Zpracování parametrů

Úkolem parametrů je informovat aplikaci, jak se má chovat a co má provádět. Uživatel musí intuitivně vědět, co daný parametr dělá a jaká hodnota se přiřazuje k jeho argumentu. Nápověda pro informování o chování celé aplikace a její nastavení musí být součástí navžených parametrů.

Mezi základní parametry rozhodně budou patřit parametry pro typ Bloomova filtru, velikost bitového pole a počet hashovacích funkcí. V rámci tohoto bloku už předpokládám přídavný parametr pro ladění (neboli kontrolní výpis během provádění). Uživatel se také musí dozvědět o chybném zadání parametru, aby mohl provést jeho nápravu.

4.2 Kontrola činností programu

Kontrola činností musí zvládat více typů výpisu, který z nich to bude zadává uživatel. Dále musí být kontrolní výpis rozšířen do co nejvíc částí celé aplikace. Rozhodně nemůže být zařazen mezi povinné parametry aplikce, ale právně naopak aplikace musí provadět svou činnost i bez tohoto rozšíření.

4.3 Zpracování vláken

Místo obyčejného načítání vstupní dat v cyklu byl předpoklad využití vláken, což je jeden z charakteristických rysů tohoto návrhu. Vlákna umožní průběžné zpracovávání dat pomocí jejich komunikace. Jiný předpoklad vedl k hashovacím funkcím, kdy každé vlákno vlastní jednu hashovací funkci. Záporem tohoto předpokladu je zbytečné vytvoření pole vláken, které vede ke zhoršení obsluhy komunikace mezi vlákny. Optimální řešení vzniklo z předpokladu souvislé a jednoduché komunikace dvou vláken. Jedno vlákno bude obsahovat obsluhu konzolové části a druhé vlákno obstará zpracování dat do aplikace. Poslední částí bude podle diagramu 4.1 fronta `Queue`, do které jedno vlákno data uloží a druhé je přečte.

Vlákna se nesmějí dopustit kolize v paměti. Podmínkou vláken musí být kontrola stavu fronty `Queue`, stav kdy fronta bude plná nebo prázdná. Jakmile nastane vyprázdnění fronty, pak si vlákna mohou ustanovit změnu čtení nebo zápisu do filtru. K porušení podmínky kontroly by mohlo vést ke ztrátě dat ve frontě `Queue`.

4.3.1 Vlákno pro data z konzole

Vlákno obstará jednoduchou komunikaci s konzolí a uloží vstupní data do fronty `Queue`. Práce vlákna bude spočívat na úrovni nekonečného cyklu s řízenou komunikací pomocí semaforu a událostí. Když uživatel zadá `CTRL+D`, nebo nastane konec souboru `EOF` nekonečný cyklus musí skončit. Tento jednoduchý koncept by měl zaručit celkové načítání dat.

4.3.2 Vlákno pro zpracování dat

Vlákno musí při zjištění nového obsahu ve frontě `Queue` začít číst a následně zpracovávat data. Diagram 4.1 demonstruje činnost vlákna. Jakmile vlákno přečte data, tak musejí přejít ke zpracování přes Bloomův filtr nebo Počítaný Bloomův filtr.

4.4 Zpracování Bloomova filtru a rozšíření

Samotná knihovna musí být jednoduchá a celkový návrh by neměl obsahovat složité konstrukce. Z tohoto důvodu je potřeba na počátku uvažovat o oddělení hashovacích funkcí a bitového pole. Úkolem knihovny je zvládnutí Bloomova filtru a rozšíření Počítaného Bloomova filtru. Zde bude opět užitečné rozdělit knihovnu na dva Bloomovy filtry, které umějí zvládat *čtení*, *zápis* a rozšířená verze *odstranění*.

4.4.1 Vytvoření hashovací funkce

Pro zkvalitnění škálovatelnosti možností pro uživatelů je umožněn výběr zvoleného typu hashovací funkce. Budoucí programátoři, kteří využívají knihovnu, uvítají její jednoduché rozšíření.

4.4.2 Vytvoření bitového pole

Jeden ze složitejšík bloků je vytvoření bitového pole, neboť se u něj očekává co nejmenší paměťová náročnost. Rozhodně není nutné využívat klasické pole typu `int`, ale spíše pole v podobě bitů. Obyčejné bitové pole stačí pro Bloomův filtr a pro Počítaný Bloomův filtr se vytvoří počítané bitové pole.

Kapitola 5

Implementace

Celková implementace je vytvořena v jazyce `Python s verzí 2.5.4` nebo `s verzí 2.6.2`. Aplikaci tvoří několik spustitelných modulů a jejich obsah se nachází v následujících podkapitolách Kapitoly 5. Každý modul se skládá z jednotlivých tříd. Třídy tvoří větší komplexní vzájemně navazující bloky, které umožňují běh aplikace. Hlavní startující modul `start` je konečnou fázi, a proto sdružuje jednotlivé třídy a spouští celou implementaci. Informace ke všem modulům s třídami včetně stromové struktury celé aplikace naleznete v souboru `README`. K řešení byl využit přídavný modul `bitarray`, více na [7].

Popis diagramu

Diagram 5.1 implementace znázorňuje konstrukci celé aplikace. Skládá se z jednotlivých tříd, které na sebe navazují. Diagram tvoří stromovou strukturu, kde menší bloky patří do větších, až do hlavního modulu `start` celé implementace.

Třída `ClassParam` je přímo závislá na třídě `ClassStart`, která ji spouští. Informace o metodách třídy `ClassParam` a popis jí samé obsahuje modul `params` v podkapitole 5.1.

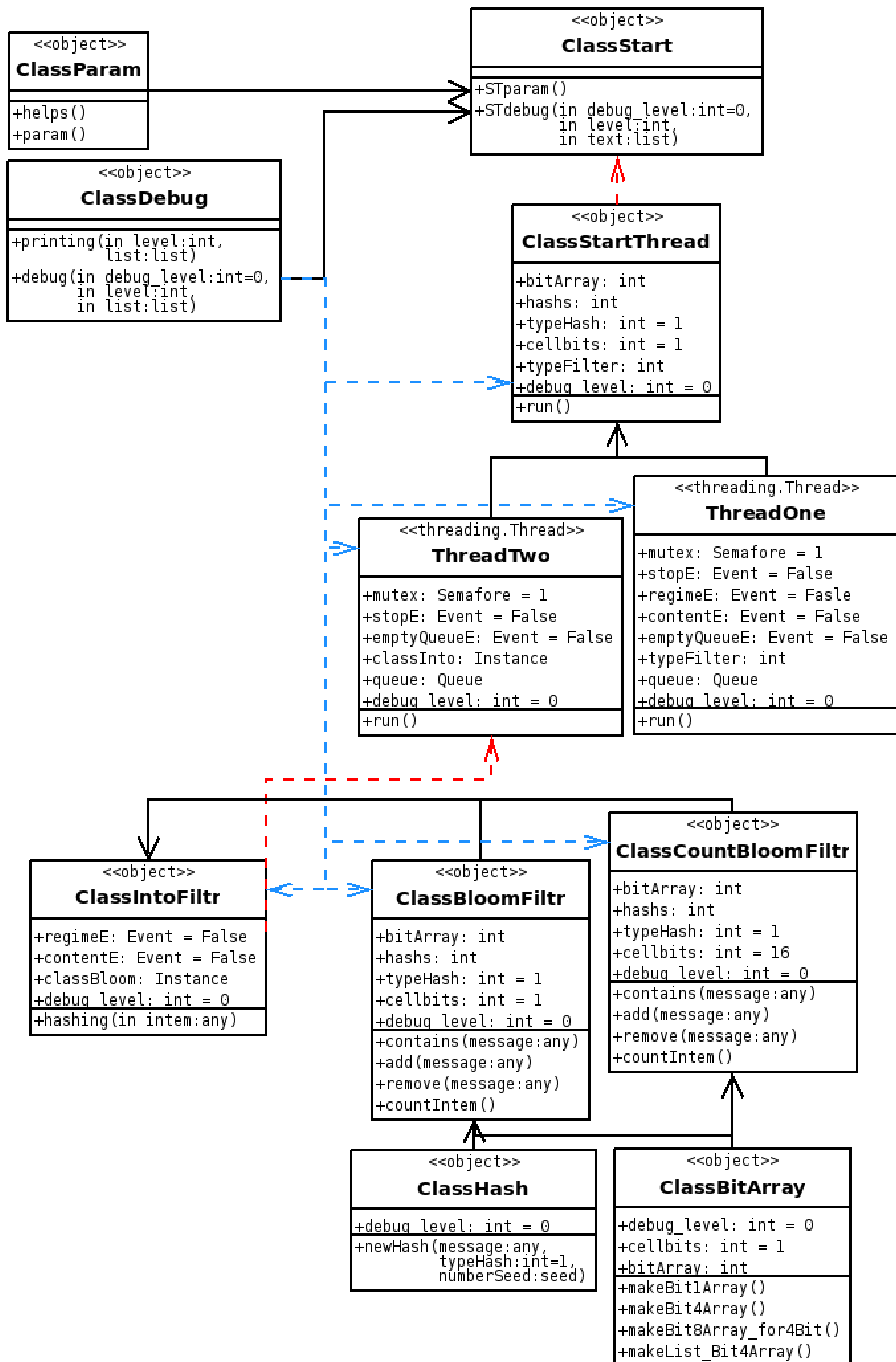
Za nejvíc rozsáhlý blok považujeme třídu `ClassDebug` v modulu `debug`. Tato třída se používá ve všech nejdůležitějších třídách, jak je vidět na diagramu 5.1. Podrobná formulace modulu je zmíněne v podkapitole 5.2.

Modul `start`, blíže vysvětlený v podkapitole 5.3, čerpá z předcházejících modulů a spouští dvě vlákna v modulu `twoThread`. Následující podkapitola 5.4 rozděluje vlákna do dvou podsekcí včetně přidružených informací.

Teď se dostáváme k nejdůležitější části celé implementace a to ke knihovně pro „Datovou strukturu Bloomův filtr a její vlastnosti“. Knihovna obsahuje modul `bloom` v podkapitole 5.5. Modul využívá podle diagramu 5.1 následující třídy `ClassHash` a `ClassBitArray`. Tyto třídy nalezneme v podkapitole 5.6 týkající se modulu `hashBitArray`.

5.1 Modul `params`

Modul obsahuje třídu `ClassParam`, která je jádrem pro tuto část aplikace. Budoucí uživatel zadává parametry do příkazového řádku a třída samotná se postará o jejich zpracování. Samozřejmostí třídy je převedení argumentu u parametru z řetězce na číslo a samotné vyhodnocení argumentu pro další provádění aplikace. Pokud by budoucí programátor chtěl třídu rozšířit, neměl by mít problém, protože k usnadnění práce je využita jednoduchá metoda `getopt()`.



Obrázek 5.1: *Diagram tříd programu Datová struktura Bloomův filtr a její vlastnosti.*

Třída `ClassParam` je rozdělena na tři základní metody

- `__init__()`,
- `helps()`,
- `param()`.

Metoda `__init__()` je inicializační pro celou třídu. Zajímavější jsou metody `helps()` pro výpis nápovědy a `param()` pro zpracování všech parametrů i jejich argumentů. Metoda `param()` v případě zadání chybného parametru informuje uživatele.

5.2 Modul debug

Modul tvoří jedna třída `ClassDebug`, která provádí průběžný výpis během programu. Tento výpis informuje uživatele nebo programátora o průběhu částí aplikace.

Třída `ClassDebug` obsahuje dvě základní metody

- `printing(level, list)`,
- `debug(debug_level, level, list)`.

Nejprve se zaměříme na metodu `debug()`, kde `debug_level` je parametr od uživatele. Zbylé dva parametry jsou přednastaveny v celé aplikaci. Podle kontroly `debug_level` a `level` metoda vyhodnotí, co se bude tisknout. Následně metoda `printing(level, list)` vytiskne úroveň výpisu a seznam hodnot.

5.3 Modul start

Modul vlastní jednoduchou třídu `ClassStrat`, která se skládá z předcházejících podkapitol 5.1 a 5.2. Tento modul spouští celou aplikaci a dále třídy modulu `twoThread` v podkapitole 5.4.

Třídu `ClassStrat` tvoří tři základní metody, se shodnými vlastnostmi předcházejících podkapitol,

- `__init__()`,
- `STparam()`,
- `STdebug(debug_level, level, list)`.

5.4 Modul twoThread

Modul obsahuje dvě třídy s vlákny `ThreadOne` a `ThreadTwo`. Vlákna jsou spouštěna třídou `ClassStartThread`. Poslední třída `ClassIntoFiltr` se váže nejen ke třídě `ThreadTwo`, ale přímo vykonává metody z modulu v podkapitole 5.5.

Třída `ClassStartThread` spouští obě vlákna pomocí následujících metod

- `__init__(bitArray, hashs, typeHash, cellbits, typeFilter, d)`,
- `run()`.

Zde metoda `__init__()` inicializuje velikost bitového pole, počet hashovacích funkcí, typ hashovacích funkcí, velikost buňky bitového pole, typ Bloomova filtru a v neposlední řadě typ výpisu (`debug_level`). Po celkové inicializaci třída spouští pomocí metody `run()` obě vlákna. Vlákna rozdělíme do dvou následujících kategorií. Vlákna `ThreadOne` a `ThreadTwo` mají společnou frontu `queue` na zápis a čtení vstupních dat. K ovládání vláken jsou využity události `stopE, regimeE, contentE, emptyQueueE` a semafor `mutex`.

5.4.1 Vlákno pro data z konzole

Třída `ThreadOne` obsluhuje vlákno, které čte vstupní data z konzole a ukládá je do fronty `queue`. Pomocí semaforu `mutex` jsou jednotlivá vlákna přepínána mezi sebou, aby nedocházelo ke stárnutí vláken. Vlákno `ThreadOne` se skládá z následujících řídících režimů *přidávání* `__ADD__`, *odstraňování* `__REM__` a *obsahování* `__CON__`. První vlákno je přímo závislé na změně režimu druhého vlákna `ThreadTwo`.

Metody třídy `ThreadOne`

- `__init__(mutex,stopE,regimeE,contentE,emptyQueueE,typeFilter, queue, d)`,
- `run()`.

Metoda `run()` spouští vlákno a metoda `__init__()` inicializuje třídu. Pokud se vlákno dozví pomocí události `emptyQueueE`, že fronta `queue` již nic neobsahuje, pak muže dojít ke změně režimu řízení. Tímto způsobem nedochází při všech režimech ke konfliktům s pamětí. Události `regimeE` a `contentE` slouží pro změnu řídícího režimu. Při režimu *odstraňování* `__REM__` nelze využít Bloomův filtru, ale pouze Počítaný Bloomův filtr. O typu filtru informuje proměnná `typeFilter`. Poslední událost `stopE` složí k ukončení vláken jen v případě, když se při čtení dat na vstupu objeví `EOF` nebo `CTRL+D`.

5.4.2 Vlákno pro zpracování dat

Třída `ThreadTwo` obsluhuje vlákno, které čte vstupní data z fronty `queue`. Druhé vlákno v cyklu odebírá data z fronty `queue` dokud nebude prázdná. Ke zpracování dat vlákno využívá metodu `hashing()` ze třídy `ClassIntoFiltr`. Semefor `mutex` a události `stopE, emptyQueueE` jsou využity shodně jako u prvního vlákna.

Metody třídy `ThreadTwo`

- `__init__(mutex, stopE, emptyQueueE, classInto, queue, d)`,
- `run()`.

Metoda `__init__()` je inicializační a obsahuje parametr v podobě instance `classInto` ze třídy `ClassIntoFiltr`. Druhá metoda slouží ke spuštění vlákna a provádění třídy `ClassIntoFiltr`.

Metody třídy `ClassIntoFiltr`

- `__init__(regimeE, contentE, classBloom, d)`,
- `hashing(intem)`.

Třída `ClassIntoFiltr` slouží ke zpracování dat do Bloomova filtru nebo do Počítaného Bloomova filtru. Metoda `__init__()` obsahuje události `regimeE`, `contentE`, jejichž vlastnosti jsou uvedeny výše. Instance `classBloom` třídy `ClassBloomFiltr` nebo třídy `ClassCountBloomFiltr` záleží na vstupním parametru od uživatele. Samotná metoda `hashing()`, která je prováděna podle režimu *přidávání*, *odstraňování* a *obsahování*, je vyvolána druhým vláknem.

5.5 Modul `bloom`

Modul `bloom` se skládá ze dvou základních tříd `ClassBloomFiltr` a `ClassCountBloomFiltr`. S těmito třídami pracuje modul `hashBitArray` v podkapitole 5.6. Oba moduly spolu tvoří knihovnu pro „Datovou strukturu Bloomův filtr a její vlastnosti“. Knihovnu zpracovává druhé vlákno v části 5.4.2.

Metody třídy `ClassBloomFiltr`

- `__init__(bitArray, hashs, typeHash, cellbits, d)`,
- `contains(message)`,
- `add(message)`,
- `remove(message)`,
- `countItem()`,
- `__del__()`.

Uvedené metody `__init__()`, `contains()` a `add()` závisejí na modulu `hashBitArray` a jeho třídách `ClassHash`, `ClassBitArray`. Metoda `__init__()` je inicializační pro celou třídu a vytváří nové bitové pole. Následující metoda `contains()` s parametrem `message` kontroluje obsah Bloomova filtru. Metoda `add()` přidává položku do Bloomova filtru. Opačná metoda `remove()` odstraňující položky nemá pro běžný Bloomův filtr využití. Informativní metoda `countItem()` obsahuje počet vložených vstupních dat. Poslední metoda `__del__()`, která má opačný charakter než metoda první, obsahuje pouze soubor pro uzavření. Soubor slouží pro metodu `contains()`, která kontroluje zadaná data. Výsledný soubor se porovnává s testovanými daty z testů. Více v Kapitole 6.

Metody třídy `ClassCountBloomFiltr`

- `__init__(bitArray, hashs, typeHash, cellbits, d)`,
- `contains(message)`,
- `add(message)`,
- `remove(message)`,
- `countItem()`,
- `__del__()`.

U třídy `ClassCountBloomFiltr` metody `__init__()`, `contains()`, `add()` a `remove()` závisejí na třídách `ClassHash`, `ClassBitArray` z modulu `hashBitArray`. Metoda `__init__()` je opět inicializační pro celou třídu a vytváří nové počítané bitové pole. Následující metoda `contains()` podle parametru `message` kontroluje obsah Počítaného Bloomova filtru. Metoda `add()` přidává položku do filtru, pokud by se zahashovala na již obsazenou pozici, tak se navýší čítač počítaného bitového pole. Opačná metoda `remove()` slouží k odstranění položky z Počítaného Bloomova filtru. Jestliže bude nastaven bit přetečení, pak poslední položka na pozici v počítaném bitovém poli nesmí být odstraněna kvůli závislosti na ostatní položky. Informativní metoda `countItem()` obsahuje počet vložených vstupních dat. Poslední metoda `__del__()` obsahuje soubor pro uzavření. Soubor slouží pro metodu `contains()`, která kontroluje zadaná data ve filtru. Výsledný soubor se porovnává s testovanými daty z testů. Více v kapitole 6.

5.6 Modul `hashBitArray`

Modul `hashBitArray` je složen ze dvou tříd. První třída `ClassHash` vytváří novou hashovací funkci. Druhá třída `ClassBitArray` vytváří bitové pole nebo počítané bitové pole. Tyto třídy jsou využity v předcházející podkapitole 5.5 a společně tvoří knihovnu.

Metody třídy `ClassHash`

- `__init__(d)`,
- `newHash(message, typeHash, numberSeed)`.

Metoda `__init__()` je inicializační pro třídu. Úžívanější metoda je `newHash()`, která vytváří podle vstupních dat výsledný hash kód. Při tvorbě hashovacích funkcí je nutné znát od uživatele typ hashovací funkce. V případě, že typ není zadán, aplikace má přednastavený typ na `md5`. Dalším aspektem je parametr `numberSeed`, který určuje s jakým seedem bude nová hashovací funkce vytvořena.

Metody třídy ClassBitArray

- __init__(bitArray, cellbits, d),
- makeBit1Array(),
- makeBit4Array(),
- makeBit8Array_for4Bit(),
- makeList_Bit4Array().

Metoda __init__() má podobné vlastnosti jako v předchozích částech. U následujících metod je použit přídavný modul bitarray instalace uvedená ve zdroji [7]. Pokud není modul nainstalovaný, bude vyvolána vyjímka ImportError, která způsobí změnu ve vytváření bitového pole. Nepoužije se bitové pole z modulu bitarray, ale modul standardní knihovny array. Jestliže bude zapotřebí standardní knihovna, tak nové bitové pole nebude vytvořeno o minimální pamětové velikosti. Jedná o změnu velikosti buňky z 1 bitu na 1 byte (bajt), který je roven 8 bitům. Bohužel takovéto řešení navýší spotřebu paměti. Řízení vyjímkou nastává u metod makeBit1Array() a makeList_Bit4Array(). Metoda makeBit1Array() tvoří bitové pole pro Bloomův filtr. Následující metoda makeList_Bit4Array() vytváří počítané bitové pole pro Počítaný Bloomův filtr, ale při vyjímce bude vyvolána metoda makeBit8Array_for4Bit(). Metoda makeBit8Array_for4Bit() je náhradou za metodu makeBit4Array().

Kapitola 6

Experimenty a výsledky

Mezi nejdůležitější část každé aplikace patří validace. Jde o ověření platnosti, že skutečně pracujeme s adekvátní aplikací. Otázkou je, jak ověřit validaci? Validace musí být jasná, přesvědčivá a její výsledky správné. Z těchto důvodů se validace zabývá všemi částmi aplikace, která musí projít sadou testů. Po proběhnutí testů následuje jejich vyhodnocení, které není vždy jednoduché. Grafy zpřehlední získané výsledky a usnadní jejich pochopení. V následujících podkapitolách nalezneme testy pro Bloomův filtr, testy pro Počítaný Bloomův filtr a dosažené výsledky. Experimenty jsou prováděny s `Pythonem 2.6.2 [GCC 4.3.3]` na `Linuxu verze 2` s jádrem `2.6.28-15-generic` pro `laptopy`.

6.1 Testy pro Bloomův filtr

Testy zaměřené na část aplikace Bloomův filtr se zabývají typem hashovacích funkcí, jejich počtem a odlišnou velikostí bitového pole. První test slouží k zjištění optimálního typu a počtu hashovacích funkcí při pozitivní chybě s pravděpodobností fp. Výsledek optimálního typu hashovací funkce bude využit ve všech následujících testech i pro Počítaný Bloomův filtr. Druhý test určuje pravděpodobnost chyby pro různou velikost bitového pole. Výsledkem bude nejvhodnější řešení pro pevně stanovené n vstupních dat. Výsledky, které korespondují s výpočty z teoretické Kapitoly 2, jsou vyobrazeny v podobě grafu.

6.1.1 Test různého typu a počtu hashovacích funkcí

Test se zabývá dvěma parametry hashovacích funkcí. Prvním parametrem je typ hashovací funkce, který lze v aplikaci nastavit na 6 možných variant. Byly vybrány tři nejdůležitější typy hashovacích funkcí `md5`, `sha1` a `sha512`.

	Bloomův filtr	
Test č.1.1 – č.1.3	Vstupní Data	Jiná Data
Vstupní Data	1 088	0
Kontrola obsahu	1 088	255

Typ hashovací funkce	md5	sha1	sha512
Počet hashovacích funkcí	0 – 15		
Velikost bitového pole	8 000		

Tabulka 6.1: Nastavení aplikace pro Testy č.1.1 – č.1.3.

Tabulka 6.1 demonstruje nastavení aplikace pro jednotlivé Testy č.1.1 – č.1.3. Vstupní data zadává uživatel aplikace, může je zadávat jednotlivě nebo jednorázově při startu aplikace. V těchto testech pak probíhá kontrola obsahu dat. U vstupních dat Bloomův filtr odpovídá vždy kladně a u jiných dat by měl odpovídat negativně. Pokud tomu tak není a nastane kladná odpověď u jiných dat, tak hovoříme o pozitivní chybě Bloomova filtru.

Druhá část tabulky nastavuje pro aplikaci typ i počet hashovacích funkcí a velikost bitového pole. Hodnoty v tabulce 6.1 jsou libovolné.

Výsledky

Po proběhnutí Testů č.1.1 – č.1.3 výsledky podle tabulek 6.2, 6.3 a 6.4 demonstrují odlišnou hashovací funkci. V testech byl kladen důraz na počet hashovacích funkcí a počet pozitivních chyb s určitou pravděpodobností. Z objevených chyb byla zjištěna pravděpodobnost fp znázorňující třetí sloupec v tabulkách. Výsledky korespondují s pravděpodobnosti fp z vypočtené Teoretické části Kapitoly 2.

Test č.1.1	Typ hashovací funkce *md5*		
Počet (HF)	Počet chyb	fp z testu	fp z teorie
0	∞	1,000000	1,000000
1	23	0,017126	0,154549
2	6	0,004468	0,081347
3	9	0,006701	0,061950
4	8	0,005957	0,057216
5	5	0,003723	0,059143
6	0	0,000000	0,065437
7	2	0,001489	0,075407
8	4	0,002978	0,088914
10	2	0,001489	0,126797
12	1	0,000745	0,179475
15	0	0,000000	0,283519
30	21	0,015637	0,822410

Tabulka 6.2: Provedení Testu č.1.1 pro hashovací funkci `md5`.

Test č.1.2	Typ hashovací funkce *sha1*		
Počet (HF)	Počet chyb	fp z testu	fp z teorie
0	∞	1,000000	1,000000
1	16	0,011914	0,154549
2	5	0,003723	0,081347
3	7	0,005212	0,061950
4	5	0,003723	0,057216
5	2	0,001489	0,059143
6	2	0,001489	0,065437
7	2	0,001489	0,075407
8	2	0,001489	0,088914
10	2	0,001489	0,126797
12	1	0,000745	0,179475
15	1	0,000745	0,283519
30	18	0,013403	0,822410

Tabulka 6.3: Provedení Testu č.1.2 pro hashovací funkci `sha1`.

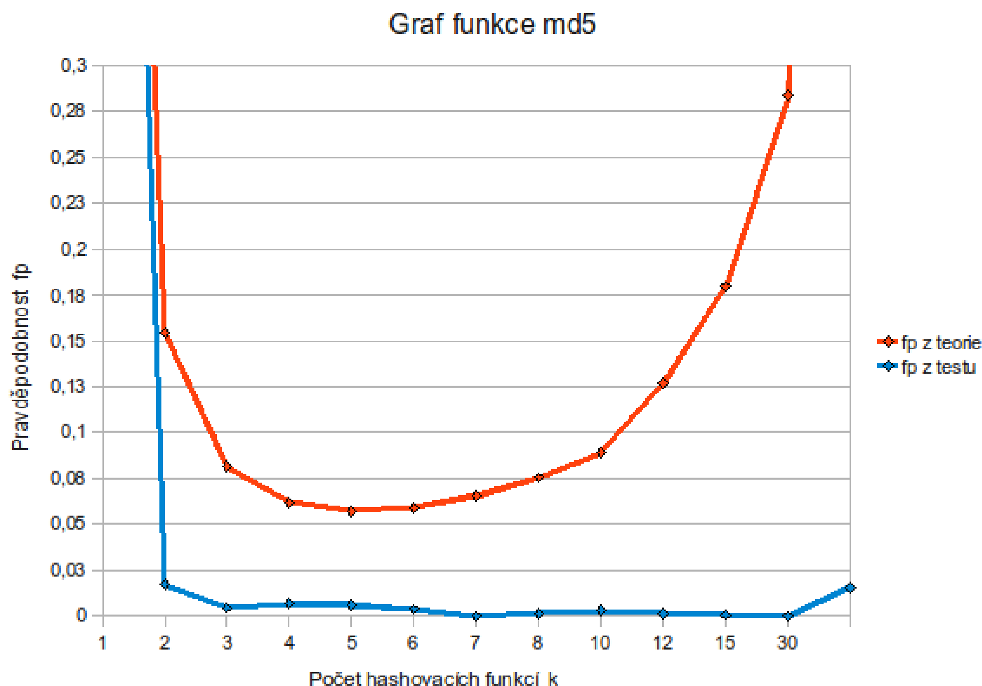
Test č.1.3	Typ hashovací funkce *sha512*		
Počet (HF)	Počet chyb	fp z testu	fp z teorie
0	∞	1,000000	1,000000
1	16	0,011914	0,154549
2	2	0,001489	0,081347
3	1	0,000745	0,061950
4	1	0,000745	0,057216
5	1	0,000745	0,059143
6	0	0,000000	0,065437
7	1	0,000745	0,075407
8	1	0,000745	0,088914
10	1	0,000745	0,126797
12	3	0,002234	0,179475
15	7	0,005212	0,283519
30	24	0,017870	0,822410

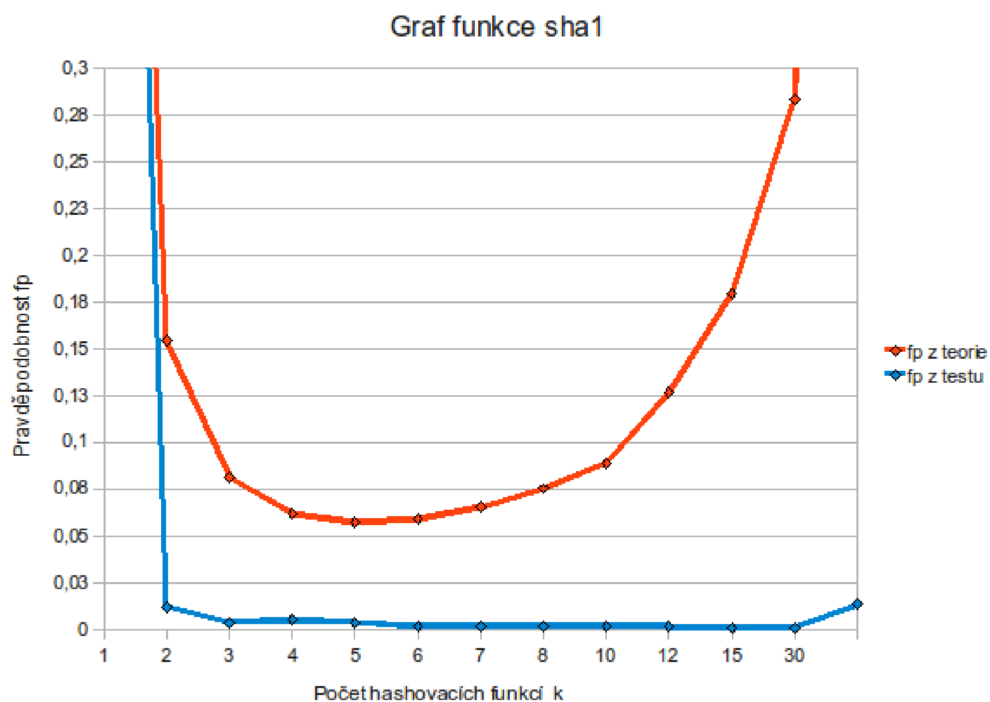
Tabulka 6.4: Provedení Testu č.1.3 pro hashovací funkci `sha512`.

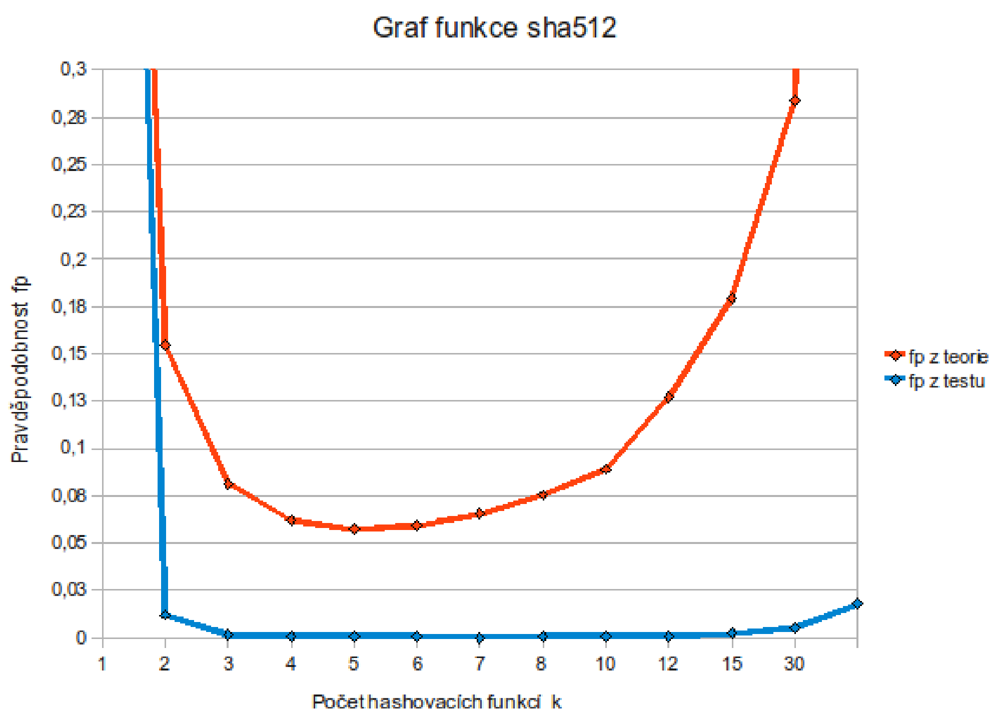
Podle grafů a tabulek zjistíme, že optimální řešení pro Bloomův filtr je při typu `sha512` funkce a počtu 6 hashovacích funkcí. Doba zpracování, zátěže CPU a zátěže RAM je pro Testy č.1.1 – č.1.3 vyobrazena v tabulce 6.5.

Zpracování	
Doba cca	2,26 – 11,85 s
Zátěž CPU	100 %
Zátěž RAM	< 4 kb

Tabulka 6.5: Vyobrazení zatížení pro Bloomův filtr při *přidávání* dat a *zjištění obsahu*.







Obrázek 6.1: *Grafy znázorňují Testy č.1.1 – č.1.3 pro hashovací funkce* `md5`*,* `sha1` *a* `sha512` *při zvyšujícím počtu funkcí 0 – 15 se závislostí na pravděpodobnosti fp v podobě modré barvy. Červená barva vypovídá o vypočítaných hodnotách pravděpodobnosti fp z teorie. Kde hodnoty počtu hashovacích funkcí 0 – 1 pro pravděpodobnost fp jsou 1. Ostatní hodnoty znázorňuje graf.*

6.1.2 Test různě velkého bitového pole

Jednoduchý test ovlivňuje velikost bitového pole od předpokládaných špatných hodnot až po hodnoty extrémně optimální neboli nadbytečné.

Test č.2	**Bloomův filtr**	
	Vstupní Data	Jiná Data
Vstupní Data	1 088	0
Kontrola obsahu	1 088	255

Typ hashovací funkce	sha512
Počet hashovacích funkcí	6
Velikost bitového pole	2 000 – 16 000

Tabulka 6.6: Nastavení aplikace pro Testu č.2.

Tabulka 6.6 znázorňuje nastavení aplikace pro Test č.2. Vstupní data jsou zadávána uživatel aplikace, může je zadávat jednotlivě nebo jednorázově při startu aplikace. V tomto testu pak probíhá kontrola obsahu dat. U vstupních dat Bloomův filtr odpovídá vždy kladně a u jiných dat by měl odpovídat negativně. Pokud tomu tak není a nastane kladná odpověď u jiných dat, tak hovoříme o pozitivní chybě Bloomova filtru.

Díky předchozímu testu jsem zjistil, že nejvhodnější typ hashovací funkce je `sha512` o počtu hashovacích funkcí 6. V tomto testu testuji velikost bitového pole 2 000 až 16 000.

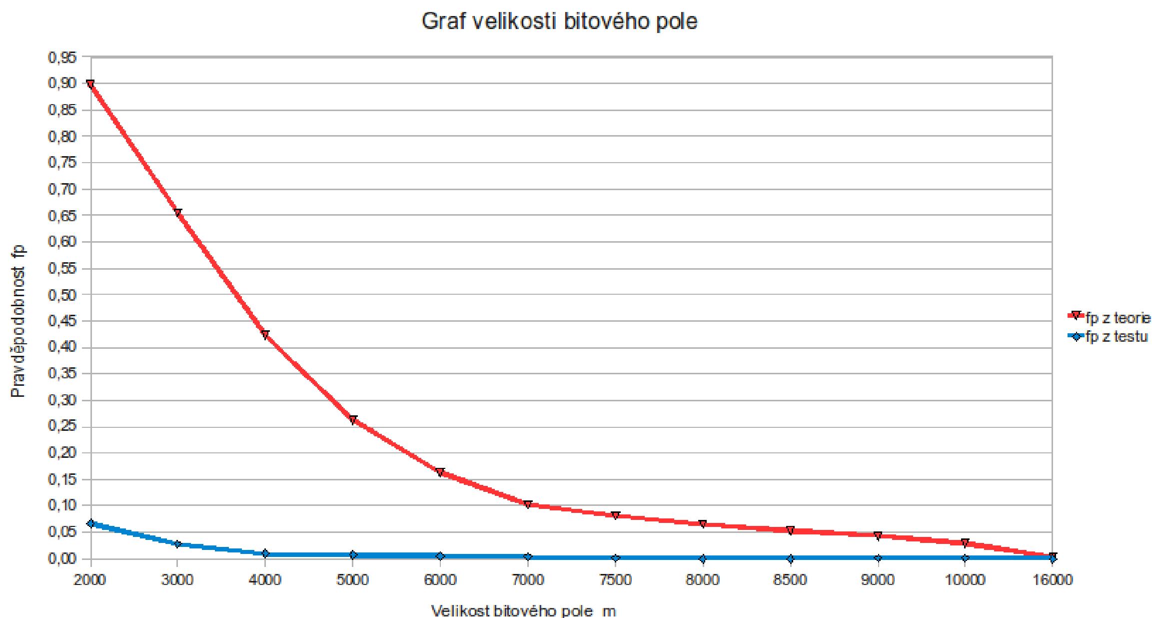
Výsledky

Cílem testu bylo zjistit optimální velikost bitového pole. Následující tabulka 6.7 zaznamenává různé nastavení bitového pole. Pro ověření optimálního výsledku z objevených chyb byla zjištěna pravděpodobnost fp pomocí Testu č.2 korespondující s pravděpodobností fp z vypočtené Teoretické části Kapitoly 2. Graf 6.2 je vytvořen porovnáním výsledků obou pravděpodobností fp.

Test č.2	Typ hashovací funkce *sha512*		
Velikost (BP)	Počet chyb	fp z testu	fp z teorie
2000	89	0,066270	0,897983
3000	37	0,027550	0,654867
4000	13	0,009680	0,423694
5000	9	0,006701	0,263058
6000	6	0,004468	0,162837
7000	5	0,003723	0,102185
7500	1	0,000745	0,081549
8000	0	0,000000	0,065437
8500	0	0,000000	0,052804
9000	1	0,000745	0,042849
10000	1	0,000745	0,028688
16000	0	0,000000	0,003837

Tabulka 6.7: Provedení Testu č.2 pro hashovací funkci `sha512`.

Z grafu 6.2 zjistíme, že optimální velikost bitového pole pro Bloomův filtr je v rozmezí 8 000 – 8 500. Doba zpracování, zátěže CPU a zátěže RAM je pro Test č.2 zobrazena v tabulce 6.8.



Obrázek 6.2: *Graf znázorňuje Test č.2 s typem hashovací funkce* `sha512` *pro velikost bitového pole 2 000 – 16 000 v závislosti na pravděpodobnosti fp v podobě modré barvy. Červená barva označuje vypočtené hodnoty pravděpodobnosti fp z teorie, které jsou hned od počatku téměř nulové.*

Zpracování	
Doba	cca 3,81 – 8,91 s
Zátěž CPU	100 %
Zátěž RAM	< 4 kb

Tabulka 6.8: Vyobrazuje zatížení pro Bloomův filtr při *přidávání* dat a *zjištění obsahu.*

6.2 Testy pro Počítaný Bloomův filtr

Zaměření Testů č.1.1 a č.1.2 se bude soustřeďovat na část aplikace Počítaný Bloomův filtr. První test slouží ke zjištění optimálního počtu hashovacích funkcí bez obdebírání dat při pozitivní chybě s pravděpodobností fp a s odebíráním. Výsledkem testů je nejvhodnější počet hashovacích funkcí. Test č.2 zjišťuje pozitivní chyby pro různou velikost počítaného bitového pole, které využívá vlastností obyčejného bitového pole s čítači. Výsledkem Testu č.2 bude optimální řešení pro pevně stanovené n vstupních dat. Veškeré výsledky zobrazují grafy, které korespondují s výpočty z Kapitoly 2.

6.2.1 Test různého počtu hashovacích funkcí bez odebíráním dat a s ním

Test zkoumá počet hashovacích funkcí při odebírání položek a bez odebírání. Při testu bude využita optimální hashovací funkce z obyčejného Bloomova filtru.

Tabulky 6.9 zobrazují nastavení aplikace pro Testy č.1.1 a č.1.2. Testy probíhají stejným způsobem jako předchozí testy, ale Test č.1.2 je doplněn o odebírání dat z Počítaného Bloomůva filtru. Pak probíha opětovná kontrola obsahu dat. Po proběhnutí dílčích částí testu se uskuteční celková kontrola výsledku podle vzorce uvedeného v tabulce 6.9.

Třetí část tabulky 6.9 obsahuje optimální hodnoty typu hashovací funkce a velikosti bitového pole podle klasického Bloomova filtru. Hodnoty počtu funkcí a velikosti čítače jsou libovolně zvolené pro daný test.

Testy č.1.1	**Počítaný Bloomův filtr**	
	Vstupní Data	Jiná Data
Vstupní Data	1 088	0
Kontrola obsahu	1 088	255

Testy č.1.2	**Počítaný Bloomův filtr**	
	Vstupní Data	Jiná Data
Vstupní Data	429	0
Kontrola obsahu	429	255
Odebrání dat	167	0
Kontrola obsahu	429	255
Celková kontrola	$429 \cdot 2 + 255 \cdot 2 = 1368$	

Typ hashovací funkce	sha512
Počet hashovacích funkcí	0 – 15
Velikost počítaného bitového pole	8 250
Velikost čitače buňky pole	16

Tabulky 6.9: Pro nastavení aplikace v Testech č.1.1 – č.1.2.

Výsledky

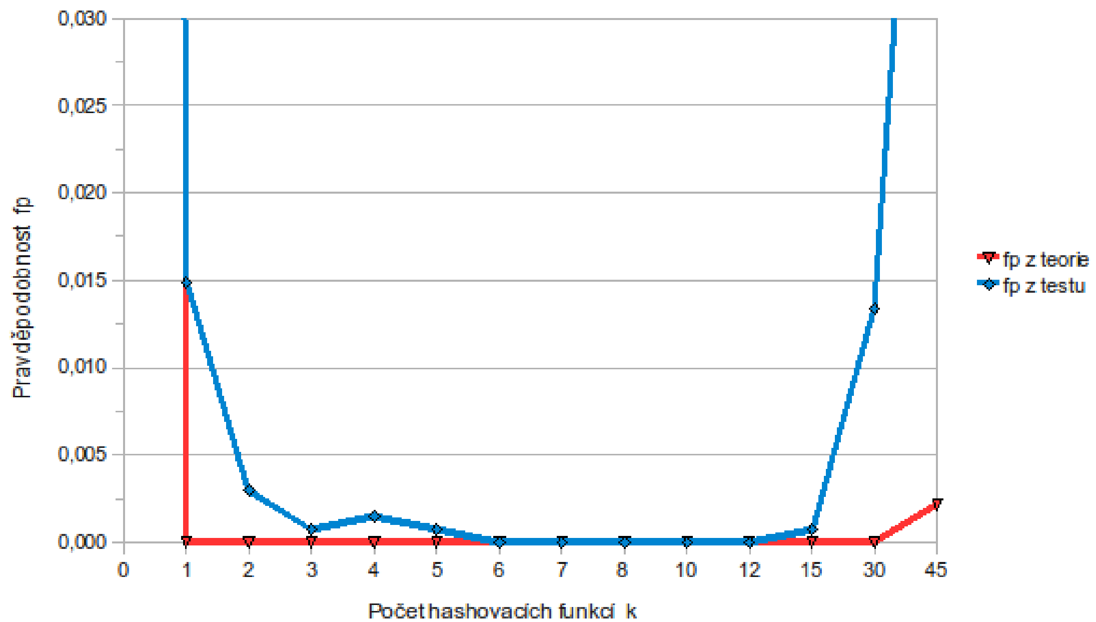
Testy č.1.1 – č.1.2 v tabulkách 6.10 a 6.11 při použití optimálního typu funkce `sha512` zobrazují výsledky nejvhodnějšího počtu hashovacích funkcí. Testy byly zaměřeny na počet hashovacích funkcí se závislostí na počet pozitivních chyb. Po proběhnutí testů byla zjištěna pravděpodobnost fp z nalezených chyb zaznaménána ve třetím sloupci v tabulkách. Výsledky korespondují s pravděpodobností fp vypočítané podle Teoretické části Kapitoly 2.

Test č.1.1	Typ hashovací funkce *sha512*		
Počet (HF)	Počet chyb	fp z testu	fp z teorie
0	∞	1,00E+000	1,00E+000
1	20	1,49E-002	9,05E-027
2	4	2,98E-003	5,27E-022
3	1	7,45E-004	2,99E-019
4	2	1,49E-003	2,55E-017
5	1	7,45E-004	7,73E-016
6	0	0,00E+000	1,22E-014
7	0	0,00E+000	1,22E-013
8	0	0,00E+000	8,81E-013
10	0	0,00E+000	2,27E-011
12	0	0,00E+000	3,03E-010
15	1	7,45E-004	6,61E-009
30	18	1,34E-002	3,78E-005
45	87	6,48E-002	2,16E-003

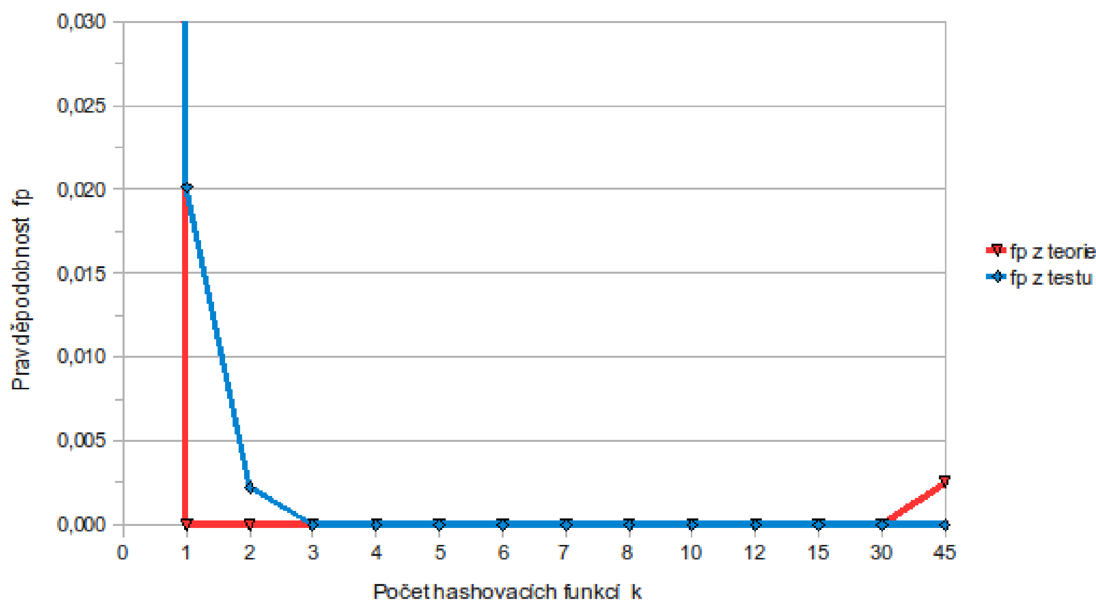
Tabulka 6.10: Provedení Testu č.1.1 pro hashovací funkci `sha512` bez odebírání položek.

Test č.1.2	Typ hashovací funkce *sha512*		
Počet (HF)	Počet chyb	fp z testu	fp z teorie
0	∞	1,00E+000	1,00E+000
1	27	2,01E-002	1,21E-026
2	3	2,23E-003	7,04E-022
3	0	0,00E+000	3,98E-019
4	0	0,00E+000	3,39E-017
5	0	0,00E+000	1,02E-015
6	0	0,00E+000	1,61E-014
7	0	0,00E+000	1,61E-013
8	0	0,00E+000	1,16E-012
10	0	0,00E+000	2,95E-011
12	0	0,00E+000	3,93E-010
15	0	0,00E+000	8,49E-009
30	0	0,00E+000	4,64E-005
45	0	0,00E+000	2,54E-003

Tabulka 6.11: Provedení Testu č.1.2 pro hashovací funkci `sha512` s odebíráním položek.



Obrázek 6.3: *Graf Testu č.1.1 koresponduje s optimální hashovací funkcí* `sha512` *bez odebírání položek a její počet od 0 – 15 v závislosti na pravděpodobnosti fp v podobě modré barvy. Červená barva charakterizuje vypočtené hodnoty pravděpodobnosti fp z teorie. Kde hodnoty počtu hashovacích funkcí 0 – 1 pro pravděpodobnost fp jsou 1. Ostatní hodnoty znázorňuje graf.*



Obrázek 6.4: *Graf Testu č.1.2 zachycuje optimální hashovací funkci* `sha512` *s odebíráním položek a její počet od 0 – 15 v závislosti na pravděpodobnosti fp v podobě modré barvy. Vypočtené hodnoty pravděpodobnosti fp z teorie značí červená barva. Hodnoty počtu hashovacích funkcí 0 – 1 pro pravděpodobnost fp jsou 1. Ostatní hodnoty znázorňuje graf.*

Z grafů 6.3 a 6.4 vyčteme, že optimální počet hashovacích funkcí `sha512` je pro Počítaný Bloomův filtr 5. Doba zpracování, zatížení CPU a zatížení RAM jsou pro Testy č.1.1 – č.1.2 zachyceny v následujících tabulkách 6.12.

Zpracování bez odebírani	
Doba	cca 2,65 – 17,32 s
Zátěž CPU	100 %
Zátěž RAM	< 4 kb

Zpracování s odebíráním	
Doba	cca 4,27 – 14,27 s
Zátěž CPU	100 %
Zátěž RAM	< 4 kb

Tabulky 6.12: Zobrazují zatížení při *přídávání* položek, při *zjištění obsahu* a bez *odebírání* i s *odebíráním* položek z Počítaného Bloomova filtru.

6.2.2 Test různě velkého počítaného bitového pole

Jednoduchý test ovlivňuje velikost počítaného bitového pole od předpokládaných špatných hodnot až po hodnoty nadbytečné.

Tabulka 6.13 charakterizuje nastavení aplikace pro Test č.2. Tento test probíhá stejným způsobem jako předchozí testy. Druhá část tabulky obsahuje nastavení optimálního typu hashovací funkce a vhodného počtu funkcí. Velikost počítaného bitového pole a velikost čítače v poli je zvolena libovolně.

Test č.2	Počítaný Bloomův filtr	
	Vstupní Data	Jiná Data
Vstupní Data	1 088	0
Kontrola obsahu	1 088	255

Typ hashovací funkce	sha512
Počet hashovacích funkcí	6
Velikost počítaného bitového pole	2 000 – 16 000
Velikost čitače v poli	2 – 16

Tabulka 6.13: Nastavení aplikace pro Test č.2.

Výsledky

Optimální velikost počítaného bitového pole bylo cílem určit v Testu č.2. Následující tabulka 6.14 znázorňuje různá nastavení počítaného bitového pole. Z Testu č.2 podle odhalených chyb zjistíme pravděpodobnost fp ve třetím sloupci tabulky 6.14. Pravděpodobnost z testu koresponduje s pravděpodobností fp vypočítané podle Teoretické části Kapitoly 2. Porovnání výsledků obou pravděpodobností fp zobrazuje graf 6.5.

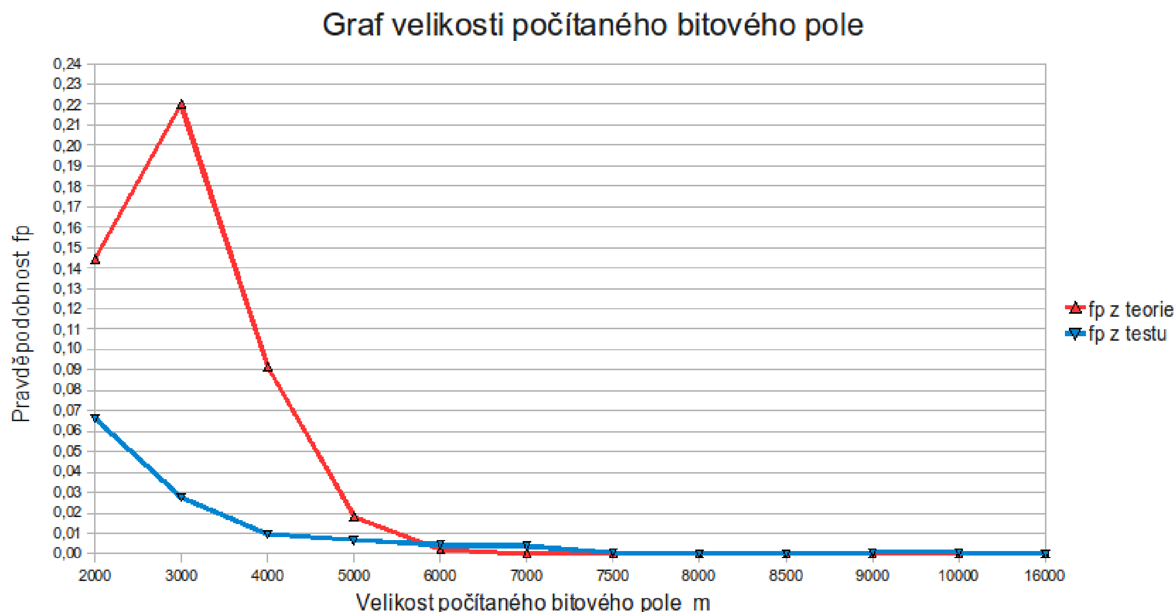
Test č.2	Typ hashovací funkce *sha512*			
Velikost (BP)	Čítačů v (BP)	Počet chyb	fp z testu	fp z teorie
2000	2	89	6,63E-002	1,44E-001
3000	3	37	2,76E-002	2,20E-001
4000	4	13	9,68E-003	9,15E-002
5000	5	9	6,70E-003	1,81E-002
6000	6	6	4,47E-003	2,13E-003
7000	7	5	3,72E-003	1,68E-004
7500	7	1	7,45E-004	1,12E-004
8000	8	0	0,00E+000	9,57E-006
8500	8	0	0,00E+000	6,25E-006
9000	9	1	7,45E-004	4,15E-007
10000	10	1	7,45E-004	1,41E-008
16000	16	0	0,00E+000	4,88E-019

Tabulka 6.14: Provedení Testu č.2 pro hashovací funkci `sha512`.

Z grafu 6.5 vyčteme, že optimální velikost počítaného bitového pole pro Počítaný Bloomův filtr, je v rozmezí 8 000 – 8 500. Doba zpracování, zátěže CPU a zátěže RAM je pro Test č.2 zobrazena v tabulce 6.15.

Zpracování	
Doba	cca 4,38 – 8,21 s
Zátěž CPU	100 %
Zátěž RAM	< 4 kb

Tabulka 6.15: Zobrazje zatížení pro Počítaný Bloomův filtr při *přidávání* položek a *zjištění obsahu*.



Obrázek 6.5: *Test č.2 hashovacích funkcí* `sha512` *pro velikost bitového pole 2 000 – 16 000 v závislosti na pravděpodobnosti* fp *v podobě modré barvy značí graf. Červená barva označuje vypočtené hodnoty pravděpodobnosti* fp *z teorie, kde vypočtené hodnoty silně závisí na počtu čítačů v poli.*

6.3 Souhrný závěr z experimentů

V testové části Bloomova filtru bylo zapotřebí zjistit optimální typ hashovací funkce. Po proběhnutí testů a vyhodnocení výsledku se jako vhodný typ hashovací funkce ukázal `sha512`. Dalším testovaným aspektem byl počet hashovacích funkcí, který se jeví v počtu 6 podle tabulek 6.2 a 6.4. V následujícím testování Bloomova filtru bylo zapotřebí se zaměřit na velikost bitového pole. K získání optimální velikosti byl použit zjištěný typ hashovací funkce, který umožnil snazší nalezení velikosti bitového pole přibližně v počtu 8 250 prvků.

V druhé části testů pro Počítaný Bloomův filtr bylo potřeba určit podobné vztahy pro počet hashovacích funkcí a velikost počítaného bitového pole. Pro nejvhodnější výsledek k počtu funkcí byl použit typ hashovací funkce `sha512` z klasického Bloomova filtru. Získaný počet funkcí bez odebírání položek je 6, ale pro počet funkcí s odebíráním položek je nižší hodnota a to 3. Poslední částí testů bylo zjistit velikost počítaného bitového pole. Porovnáním tabulek 6.7 a 6.14 jsem ověřil, že velikost bitového pole je shodná pro oba Bloomovy filtry. Pak velikost bitového pole vychází okolo 8 250 prvků. Počítaný Bloomův filtr umožňuje odebírání položek, a proto je lépe použitelný v reálném prostředí než klasický Bloomův filtr.

Kapitola 7

Závěr

Cílem bakalářské práce bylo nastudovat „Datovou strukturu Bloomův filtr a její vlastnosti", následně ji navrhnou a implementovat do podoby knihovny. Implementace byla rozšířena na funkční aplikaci s použitím vytvořené knihovny. Aplikace se dělí na několik menších částí, které lze jednoduše pochopit i v budoucnu rozšířit.

V Teoretické části nalezneme hashovací funkce, jejich vlastnosti včetně jejich potenciálního použití v kryptografii. V následujících podkapitolách pokračuji se zaměřením na základní úskalí návrhu od Konvenčních hashovacích metod až po samotnou teorii o Bloomově filtru. Všechny Bloomovy filtry jsou zatíženy pozitivní chybou (anglicky „false positive") s pravděpodobností fp, která byla objasněna. V neposlední řadě se Kapitola zabývá rozšířením na Počítaný Bloomův filtr. Toto rozšíření má větší možnosti využití v reálném prostředí informatiky zásluhou odebíratelnosti položek z filtru.

Kapitola Praktické využití Bloomova filtru nám vysvětlila reálné uplatnění v Informačních technologiích.

Na počátku tvorby knihovny bylo zapotřebí se zabývat hashovacími funkcemi, neboť Bloomův filtr využívá právě jejich vlastností. Další součástí knihovny byla implementace bitového pole v podobě vhodné velikosti paměti bez nadbytku alokovaných stránek v RAM paměti. Z tohoto důvodu byla využita přídavná knihovna `bitarray`, která je uvedená v Kapitole Návrh řešení. Tato kapitola se zaměřuje na návrh celé aplikace včetně knihovny samotné. Jednoduchý diagram znázorňuje základní předpoklady pro chod aplikace. Zde byly zveřejněny základní myšlenky a podmínky pro samotný návrh řešení.

V Kapitole Implementace byly popsány jednotlivé moduly aplikace s třídami. Diagram v této kapitole zpřehledňuje celou implementační část a ukazuje závislosti mezi vlákny a ostatními třídami.

V poslední části bakalářské práce jsem navrhl a implementovat několik experimentů, kterými byla testována celá dokončená aplikace. Při experimentech jsem použil libovolná vstupní data, která nebyla v průběhu testů nijak měněna. Experimenty byly zaměřeny jak na Bloomův filtr, tak i na jeho rozšíření, Počítaný Bloomův filtr. Pomocí testů jsem zkoumal pozitivní chyby s pravděpodobností fp. V závěrečné části experimentů jsem zhodnotil dosažené výsledky pro Bloomovy filtry.

Pro budoucí hlubší rozšíření knihovny (neboli celé aplikace) navrhuji Škálovatelné Bloomovy filtry a Komprimované Bloomovy filtry. Takové rozšíření by mělo prohloubit celou problematiku Bloomova filtru a pomoci v různých síťových aplikacích nebo jiných problémech.

Literatura

[1] Bloom, B. H.: Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM*, ročník 13, č. 7, 1970: s. 422–426.

[2] Bonomi, F.; Mitzenmacher, M.; Panigrahy, R.; aj.: An Improved Construction for Counting Bloom Filters. In *ESA*, *Lecture Notes in Computer Science*, ročník 4168, editace Y. Azar; T. Erlebach, Springer, 2006, ISBN 3-540-38875-3, s. 684–695.

[3] Broder, A. Z.; Mitzenmacher, M.: Survey: Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, ročník 1, č. 4, 2003.

[4] Chang, F.; Dean, J.; Ghemawat, S.; aj.: Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, ročník 26, č. 2, 2008.

[5] Coron, J.-S.; Dodis, Y.; Malinaud, C.; aj.: Merkle-Damgård Revisited: How to Construct a Hash Function. In *CRYPTO*, *Lecture Notes in Computer Science*, ročník 3621, editace V. Shoup, Springer, 2005, ISBN 3-540-28114-2, s. 430–448.

[6] Donnet, B.; Baynat, B.; Friedman, T.: Retouched bloom filters: allowing networked applications to trade off selected false positives against false negatives. In *CoNEXT*, editace C. Diot; M. H. Ammar, ACM, 2006, ISBN 1-59593-456-1, str. 13.

[7] Ilan Schnell: Bitarray, Version 0.3.5 [online]. April 2009, [cit. 2009-04-07].
URL http://pypi.python.org/pypi/bitarray/

[8] N. Ferguson, S. Lucks, B. Schneier, D. Whiting, M. Bellare, T. Kohno, J. Callas and J.Walker: The Skein Hash Function Family, Version 1.2 [online]. September 2009, [cit. 2009-11-15].
URL http://www.schneier.com/skein1.2.pdf

[9] Quinlan, S.; Dorward, S.: Venti: A New Approach to Archival Storage. In *FAST*, editace D. D. E. Long, USENIX, 2002, ISBN 1-880446-03-0, s. 89–101.

[10] Rafael P. Laufer, Pedro B. Velloso, Otto Carlos M. B. Duart: *Generalized Bloom filters*. Electrical Engineering Program, COPPE/UFRJ, September 2005.

[11] Weisstein, Eric W.: Hash funcions [online]. May 2010, [cit. 2010-05-04].
URL http://mathworld.wolfram.com/HashFunction.html

[12] Wikipedia: Hash function [online]. April 2010, [cit. 2010-04-29].
URL http://en.wikipedia.org/wiki/Hash_function

[13] Wikipedia: Bloom filter [online]. April 2010, [cit. 2010-04-30].
URL http://en.wikipedia.org/wiki/Bloom_filter

[14] Wikipedia: Kryptografická hašovací funkce [online]. Únor 2010, [cit. 2010-02-01]. URL http://cs.wikipedia.org/wiki/Kryptografická_hašovací_funkce

Dodatek A

Obsah CD

Obsahem CD jsou zdrojové kódy aplikace v jazyce Python. Program tvoří startující skript, balíčky hlavního programu, balíček pro přepis slov do jednoho sloupce, zálohující skript a v neposlední řadě testující skript pro Bloomův filtr a Počítaný Bloomův filtr. K CD je také přidám soubor README, který obsahuje základní informace o aplikaci a spuštění. Součástí je Bakalářská práce ve formátu *.pdf, která obsahuje teorii, návrh, implementaci, informace o typech testů, výsledky testů, závěr a přílohy (CD, manuál).

Dodatek B

Manuál

Popis parametrů pro spuštění

1. [--help/-h] Nápověda programu.
2. [--info] Informace o třídě.
3. [--dir] Informace o importovaných modulech.
4. [--BF] klasický Bloomův filtr.
5. [--CBF] Počítaný Bloomův filtr.
6. [-b velikost_bitovéhoPole] Velikost bitového pole pro hashovací funkce (povinný parametr).
7. [-k počet_hashovacíchFunkcí] Počet různých hashovacích funkcí pro ukládání do bitového pole (povinný parametr).
8. [-t typ_hashovacíFunkce] Typ hashovací funkce (nepovinný parametr):
   - md5 = přednastaveno(1),
   - sha1 = (2),
   - sha224 = (3),
   - sha256 = (4),
   - sha384 = (5),
   - sha512 = (6).
9. [-n počet_bitovýchBuněk] Velikost bitové buňky (nepovinný parametr):
   - Bloomův filtr = 1 bit pevně,
   - Počítaný Bloomův filtr = 16 bitů plus 1 bit přetečení.
10. [-d číslo] Ladící parametr a jeho argument určuje typ výpisu. (nepovinný parametr)

Příklady použití

- `python start.py` [`--help/-h`]
- `python start.py` [`--BF/--CBF`] [`-b` velikost_bitovéhoPole] [`-k` počet_hashovacíchFunkcí] (povinné parametry) a dále [`-t` typ_hashovacíFunkce] [`-n` počet_bitovýchBuněk] [`-d` číslo] (nepovinné parametry)
- `python start.py --help`
- `python start.py -h`

  Aplikace vypíše nápovědu.
- `python start.py --BF -b 10000 -k 3 < VstupniData.txt`
- `python start.py --BF -b 10000 -k 3 < VstupniData`

  Pro obsah souboru VstupniData musí být jednotlivé položky dat organizovány do sloupce.

  Příklad:

  - polozka1\n
  - polozka2\n
  - polozka3\n
  - a_dalsi\n
- `python start.py --BF -b 10000 -k 3 -d 1`

  V případě tohoto spuštění nebo ostatních možností spuštění se očekávají vstupní data jako zápis do konzole s `ENTREM` na konci řádku.

  Příklad:

  - polozka1\n
- `python start.py --BF -b 10000 -k 3 -t 5`
- `python start.py --CBF -b 10000 -k 3`
- `python start.py --CBF -b 10000 -k 3 -n 5`
- `python start.py --CBF -b 10000 -k 3 -n 6 -t 6`