PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

BAKALÁŘSKÁ PRÁCE

Metody hašování

2015 Jiří Šubík

Anotace

Cílem této práce je vytvořit aplikaci demonstrující metody hašování a empirickými testy provést srovnání jednotlivých metod. Práce zahrnuje statické i dynamické hašování.

Děkuji vedoucímu práce, RNDr. Arnoštu Večerkovi, za odborné vedení a rady při tvorbě této bakalářské práce.

Obsah

1. Úvod **8**

2. Hašování **8**
- 2.1. Hašovací tabulka 9
- 2.2. Hašovací funkce 10
  - 2.2.1. Kvalita hašovací funkce 10
  - 2.2.2. Hašovací funkce pro řetězce 10
- 2.3. Kolize 12

3. Metody hašování **13**
- 3.1. Otevřené adresování 13
  - 3.1.1. Lineární vyhledávání 14
  - 3.1.2. Kvadratické vyhledávání 17
  - 3.1.3. Dvojí hašování 19
  - 3.1.4. Perfektní hašování 20
- 3.2. Zřetězení 21

4. Složitost hašování **24**

5. Rozšířitelné hašování **25**

6. Empirické testy **29**
- 6.1. Velikost tabulky m = 1001 29
  - 6.1.1. Funkce c_1 29
  - 6.1.2. Funkce c_2 31
  - 6.1.3. Funkce c_3 33
- 6.2. Velikost tabulky m = 10001 34
  - 6.2.1. Funkce c_1 34
  - 6.2.2. Funkce c_2 36
  - 6.2.3. Funkce c_3 38
- 6.3. Velikost tabulky m = 100001 39
  - 6.3.1. Funkce c_1 39
  - 6.3.2. Funkce c_2 41
  - 6.3.3. Funkce c_3 43
- 6.4. Porovnání funkcí a hašovacích metod 45
  - 6.4.1. Funkce pro řetězce 45
  - 6.4.2. Metody hašování 45

7. Implementace **47**
- 7.1. Programátorská část 47
  - 7.1.1. Aplikační logika 47
  - 7.1.2. C++/CLI wrapper 52

7.1.3. Generátor řetězců 55
7.2. Uživatelská část 57
7.2.1. Aplikační logika 57
7.2.2. C++/CLI wrapper 58
7.2.3. Empirické testy 58

Závěr **60**

Reference **61**

A. První příloha **62**

B. Obsah přiloženého CD **63**

Seznam obrázků

1. Princip hašování. 8
2. Hašovací tabulka. 9
3. Lineární vyhledávání - hledání volné pozice 1 14
4. Lineární vyhledávání - hledání volné pozice 2 15
5. Lineární vyhledávání - odstranění prvku z tabulky 16
6. Kvadratické vyhledávání - hledání volné pozice 17
7. Kvadratické vyhledávání - hledání volné pozice na konci tabulky . 18
8. Dvojí hašování - hledání volné pozice 1 19
9. Zřetězení - tabulka ukazatelů na seznam 21
10. Spojový seznam - vkládání na začátek seznamu 21
11. Spojový seznam - vkládání na konec seznamu 22
12. Zřetězení - vkládání na začátek seznamu 23
13. Rozšířitelné hašování 25
14. Rozšířitelné hašování - příklad 26
15. Rozšířitelné hašování - příklad 27
16. Rozšířitelné hašování - příklad 27
17. Výpis programu HashingConsoleApp.exe 57
18. Výpis programu WrapperTesting.exe 58
19. Výpis programu EmpiricalTesting.exe 59

Seznam tabulek

1. Hašovací tabulka - vkládání prvků. 12
2. Vyhledávací metody - složitost. 24
3. Empirické testy - Lineární vyhledávání - funkce c1, m = 1001 . . . 29
4. Empirické testy - Kvadratické vyhledávání - funkce c1, m = 1001 . 30
5. Empirické testy - Dvojí hašování - funkce c1, m = 1001 30
6. Empirické testy - Zřetězení - funkce c1, m = 1001 30
7. Empirické testy - Lineární vyhledávání - funkce c2, m = 1001 . . . 31
8. Empirické testy - Kvadratické vyhledávání - funkce c2, m = 1001 . 31
9. Empirické testy - Dvojí hašování - funkce c2, m = 1001 32
10. Empirické testy - Zřetězení - funkce c2, m = 1001 32
11. Empirické testy - Lineární vyhledávání - funkce c3, m = 1001 . . . 33
12. Empirické testy - Kvadratické vyhledávání - funkce c3, m = 1001 . 33
13. Empirické testy - Dvojí hašování - funkce c3, m = 1001 33
14. Empirické testy - Zřetězení - funkce c3, m = 1001 34
15. Empirické testy - Lineární vyhledávání - funkce c1, m = 10001 . . 34
16. Empirické testy - Kvadratické vyhledávání - funkce c1, m = 10001 35
17. Empirické testy - Dvojí hašování - funkce c1, m = 10001 35
18. Empirické testy - Zřetězení - funkce c1, m = 10001 35
19. Empirické testy - Lineární vyhledávání - funkce c2, m = 10001 . . 36
20. Empirické testy - Kvadratické vyhledávání - funkce c2, m = 10001 36
21. Empirické testy - Dvojí hašování - funkce c2, m = 10001 37
22. Empirické testy - Zřetězení - funkce c2, m = 10001 37
23. Empirické testy - Lineární vyhledávání - funkce c3, m = 10001 . . 38
24. Empirické testy - Kvadratické vyhledávání - funkce c3, m = 10001 38
25. Empirické testy - Dvojí hašování - funkce c3, m = 10001 38
26. Empirické testy - Zřetězení - funkce c3, m = 10001 39
27. Empirické testy - Lineární vyhledávání - funkce c1, m = 100001 . 39
28. Empirické testy - Kvadratické vyhledávání - funkce c1, m = 100001 40
29. Empirické testy - Dvojí hašování - funkce c1, m = 100001 40
30. Empirické testy - Zřetězení - funkce c1, m = 100001 40
31. Empirické testy - Lineární vyhledávání - funkce c2, m = 100001 . 41
32. Empirické testy - Kvadratické vyhledávání - funkce c2, m = 100001 41
33. Empirické testy - Dvojí hašování - funkce c2, m = 100001 42
34. Empirické testy - Zřetězení - funkce c2, m = 100001 42
35. Empirické testy - Lineární vyhledávání - funkce c3, m = 100001 . 43
36. Empirické testy - Kvadratické vyhledávání - funkce c3, m = 100001 43
37. Empirické testy - Dvojí hašování - funkce c3, m = 100001 43
38. Empirické testy - Zřetězení - funkce c3, m = 100001 44

1. Úvod

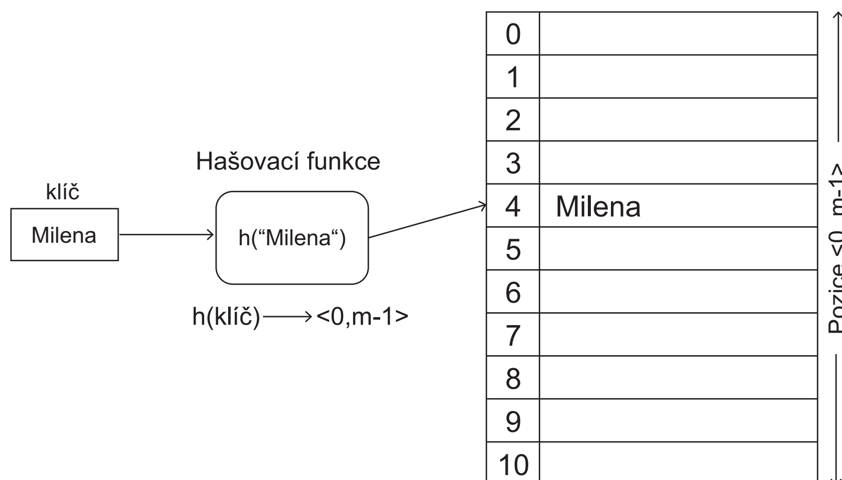
Vyhledávání je, nejen v informatice, velmi častá disciplína, kdy naším cílem je nalézt hledanou hodnotu v nějaké množině hodnot. Vyhledávat můžeme v různých datových strukturách, přičemž vyhledávací metody se liší především v časové složitosti nalezení hledané hodnoty. V této práce si představíme vyhledávací metodu zvanou hašování.

Cílem této práce je vytvořit aplikaci demonstrující metody hašování. Nejprve si popíšeme princip hašování, použitou datovou strukturu pro ukládání prvků a vysvětlíme si fungování hašovací funkce. Následně si popíšeme jednotlivé metody hašování a na příkladech si ukážeme jejich vlastnosti. V závěru jednotlivé hašovací metody empiricky srovnáme z hlediska časové složitosti.

Hašování můžeme rozdělit na statické a dynamické, kde u statického hašování máme definovanou hašovací tabulku pevné, předem dané, velikosti. Naopak při dynamickém hašování se velikost hašovací tabulky nenastavuje, ale její velikost se v průběhu hašování přizpůsobuje aktuálním potřebám.

2. Hašování

Hašování je vysoce účinná metoda vyhledávání založená na transformaci klíče. Princip hašování (obr. 1.) spočívá v tom, že hašovací funkce transformuje vyhledávací klíč do hašovací tabulky, čili na určitou pozici v tabulce.



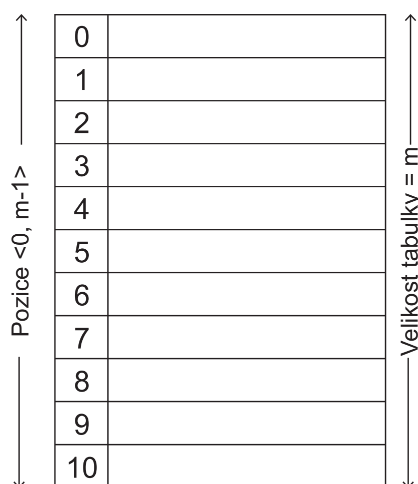
Obrázek 1. Princip hašování.

Účinnost hašování závisí na různých faktorech, které si v dalším textu detailně představíme. Hašování srovnáme s dalšími vyhledávacími metodami, což nám může pomoci s výběrem vyhledávací metody.

2.1. Hašovací tabulka

Hašovací tabulka se skládá z m řádků a v každém řádku tabulky může být uložena nejvýše jedna hodnota. Velikost hašovací tabulky značíme m a volíme ji jako prvočíslo, jelikož to nemá netriviálního dělitele, čímž dosáhneme nejlepšího předpokladu rovnoměrného rozmístění záznamů v tabulce. V případě statického hašování velikost tabulky volíme na začátku hašování. Hašovací tabulku implementujeme pomocí pole[1].

Datový typ hodnoty, ukládané do hašovací tabulky, zvolíme dle potřeby. Můžeme použít jednoduché datové typy, jako jsou čísla nebo řetězce, nebo můžeme použít strukturované datové typy, které se skládají z jednoduchých datových typů, z nichž jeden zvolíme vyhledávacím klíčem.



Obrázek 2. Hašovací tabulka.

Rozsah hašovací tabulky je v intervalu $< 0, m - 1 >$, jednotlivé pozice v tabulce tedy můžeme adresovat čísly $0, 1, 2...m - 1$, které odpovídají jednotlivým indexům pole v případě číslování indexů pole od nuly, což je nejběžnější[2] způsob indexování pole.

Faktor zaplnění λ[3] je definován vztahem:

$$\lambda = \frac{počet\ záznamů}{m}$$

λ je poměr počtu záznamů (obsazených pozic) hašovací tabulky k její velikosti. Např. při velikosti hašovací tabulky $m = 10$ a *počtu záznamů* $= 7$ je $\lambda = 0,7$, což odpovídá tabulce zaplněné ze 70%.

[1] Pole je lineární datová struktura s přímým přístupem pomocí indexu

[2] Existují programovací jazyky indexující pole od 1 nebo s možností počáteční index zvolit

[3] Řecké písmeno lambda

2.2. Hašovací funkce

Hašovací funkce je zobrazení $h : x \rightarrow < 0, m - 1 >$, které hodnotě prvku nebo vyhledávacímu klíči[4] přiřazuje nezáporné celé číslo z intervalu $< 0, m-1 >$, které odpovídá pozici v hašovací tabulce o velikosti m.

Hašovací funkce se skládá ze dvou funkcí:

$$h(x) = c(x) mod\ m$$

1. Funkce *c(x)* - je zobrazení $c : x \rightarrow \mathbb{N}_0$, funkce tedy zobrazuje hodnotu prvku nebo vyhledávací klíč na nezáporné celé číslo

2. Funkce *mod* - zbytek po celočíselném dělení

V dalším textu budeme pracovat s velikostí hašovací tabulky $m = 11$ a jednotlivé pozice v hašovací tabulce budeme adresovat čísly $0, 1, 2 ... m - 1$.

2.2.1. Kvalita hašovací funkce

Požadavky na kvalitní hašovací funkci:

- Zobrazovat prvky na co největší počet čísel
- Rovnoměrné zobrazení
- Výpočetně nenáročný výpočet hašovací funkce

Na kvalitě hašovací funkce závisí účinnost hašování.

2.2.2. Hašovací funkce pro řetězce

Řetězce patří mezi nejčastější vyhledávací klíče.

Definice 1. Značení řetězců:

$$z = z_1 z_2 ... z_{k-1} z_k$$

kde z_i je znak v řetězci, pro $i = 1, 2, ..., k - 1, k$, kde $|z| = k$ je délka řetězce.

Definice 2. Řetězce jsou uloženy v počítači jako pole znaků, jedná se tedy o posloupnost znaků. Způsob reprezentace znaků v počítači je dán číselným kódem daného znaku. Hovoříme pak o znakové sadě, která každému znaku přiřazuje číselný kód. Historicky nejznámější je ASCII[5] znaková sada anglické abecedy. V současnosti nejpoužívanější znaková sada Unicode obsahuje znaky všech existujících abeced.

[4] V případě strukturovaného datového typu
[5] American Standard Code for Information Interchange

V dalším textu se omezíme na znakovou sadu ASCII a navrhneme si funkce pro řetězce. Předpokládáme, že budeme do hašovací tabulky ukládat řetězce délky 3 až 20 znaků.

Funkce 1. Funkce $c_1(z)$:

$$c_1(z) = \sum_{i=1}^{k} asc(z_i) = asc(z_1) + asc(z_2) + ... + asc(z_{k-1}) + asc(z_k)$$

kde z_i je znak v řetězci a $asc(z_i)$ je funkce vracející ASCII kód znaku z_i. Funkce je součet ASCII kódů všech znaků v řetězci.

Funkce 2. Funkce $c_2(z)$:

$$c_2(z) = p \times asc(z_1) + q \times asc(z_2) + asc(z_h) + q \times asc(z_k) + k$$

kde z_1 je první znak v řetězci, z_2 je druhý znak v řetězci, z_h je prostřední znak, kde $h = \frac{k}{2}$, z_k je poslední znak v řetězci, k je délka řetězce a $asc(z_i)$ je funkce vracející ASCII kód znaku z_i. Funkce je součet ASCII kódů prvního × p, druhého × q, prostředního, posledního znaku × q v řetězci a délky řetězce. Proměnné p a q jsou předem zvolené konstanty, nejlépe prvočísla.

Funkce 3. Funkce $c_3(z)$:

$$c_3(z) = p \times asc(z_1) + q \times asc(z_2) + asc(z_k) + k$$

kde z_1 je první znak v řetězci, z_2 je druhý znak v řetězci, z_k je poslední znak v řetězci, k je délka řetězce a $asc(z_i)$ je funkce vracející ASCII kód znaku z_i. Funkce je součet ASCII kódů prvního × p, druhého × q, posledního znaku v řetězci a délky řetězce. Proměnné p a q jsou předem zvolené konstanty, nejlépe prvočísla.

Proměnné p a q ve funkcích $c_2(z)$ a $c_3(z)$ závisí na velikosti hašovací tabulky. Např. pro velikost tabulky $m = 11$ volíme $p = 7$ a $q = 3$, pro velikost tabulky $m = 101$ volíme $p = 127$ a $q = 31$ nebo pro velikost tabulky $m = 1001$ volíme $p = 512$ a $q = 127$ atd.

Uvedené funkce v kapitole 6. (Empirické testy) vzájemně srovnáme ve všech metodách hašování a rozhodneme o jejich správném či nesprávném návrhu v závislosti na účinnosti hašování, vzhledem k volbě funkce pro řetězce a použité metodě hašování.

Příklad 1. Do hašovací tabulky(viz. 1.) velikosti $m = 11$ budeme ukládat řetězce. Použijeme funkci c_1. Hašovací funkce je definována následovně:

$$h(z) = c_1(z) \mathit{mod\ 11}$$

Do tabulky uložíme následující řetězce:
h(Milena) = (77 + 105 + 108 + 101 + 110 + 97) mod 11 = **4**
h(Felix) = (70 + 101 + 108 + 105 + 120) mod 11 = **9**
h(Ivan) = (73 + 118 + 97 + 110) mod 11 = **2**
h(Petr) = (80 + 101 + 116 + 114) mod 11 = **4**

Výsledek hašovací funkce odpovídá pozici v hašovací tabulce, na kterou prvek vložíme. Např. h(Milena) = 4 odpovídá pozici číslo 4 v hašovací tabulce.

Při pokusu o vložení řetězce *Petr* dojde k situaci, kdy pozice v hašovací tabulce je již obsazena. Taková situace se nazývá kolize.

0	
1	
2	
3	
4	Milena
5	
6	
7	
8	
9	
10	

0	
1	
2	
3	
4	Milena
5	
6	
7	
8	
9	Felix
10	

0	
1	
2	Ivan
3	
4	Milena
5	
6	
7	
8	
9	Felix
10	

Tabulka 1. Hašovací tabulka - vkládání prvků.

2.3. Kolize

Kolize je situace, pro kterou platí:

$$h(klíč_1) = h(klíč_2) \text{ a zároveň } klíč_1 \neq klíč_2$$

Dva různé klíče mají stejnou hodnotu hašovací funkce, ukazují tedy na stejnou pozici v hašovací tabulce. Kolize jsou v hašování běžné a jejich řešení závisí na zvolené metodě hašování.

3. Metody hašování

Metody hašování dělíme dle způsobu řešení kolizí. Otevřené adresování řeší vzniklé kolize vyhledáním volné pozice v hašovací tabulce a tam pak hodnotu či prvek vloží. Zřetězení řeší kolize vkládáním hodnot, se stejnou hodnotou hašovací funkce, do spojového seznamu.

3.1. Otevřené adresování

Otevřené adresování řeší kolize vyhledáním volné pozice v hašovací tabulce a vložením hodnoty na tuto volnou pozici. Způsob vyhledání volné pozice se liší dle použité metody, kdy každá z nich používá jiný postup vedoucí k nalezení volné pozice v hašovací tabulce.

Definované operace v hašovací tabulce s otevřeným adresováním:

Vyhledávání prvku: Spočítáme hodnotu hašovací funkce *h(x)*. Výsledkem je primární pozice v hašovací tabulce, na kterou se podíváme. Mohou nastat následující situace:

1. Pozice je prázdná → hledaný prvek není v tabulce uložen a vyhledávání prvku neúspěšně končí
2. Na pozici je hledaný prvek → vyhledávání prvku úspěšně končí
3. Na pozici je jiný prvek → vyhledávání pokračuje na dalších pozicích dle použité metody hašování a pokračujeme body 1, 2 nebo 3
4. Při zcela zaplněné tabulce po prozkoumání celé tabulky není hledaný prvek v tabulce nalezen a vyhledávání prvku neúspěšně končí

Vkládání prvku: Jestliže je hašovací tabulka zcela zaplněná, vkládání končí neúspěšně, jelikož nemáme volnou pozici pro vložení prvku. Při nezaplněné hašovací tabulce spočítáme hodnotu hašovací funkce *h(x)* a prvek vyhledáme. Pokud je prvek nalezen, vkládání končí neúspěšně, jelikož prvek může být v hašovací tabulce nejvýše jednou. Jinak pokračujeme následovně:

1. Pozice je prázdná → prvek vložíme na tuto pozici a vložení prvku úspěšně končí
2. Na pozici je jiný prvek → vyhledávání pokračuje na dalších pozicích dle použité metody hašování a pokračujeme body 1 a 2

Smazání prvku: Spočítáme hodnotu hašovací funkce *h(x)* a prvek vyhledáme. Mohou nastat následující situace:

1. Prvek nebyl nalezen → prvek není v hašovací tabulce, nelze jej tedy odebrat, smazání prvku neúspěšně končí
2. Prvek byl nalezen → prvek z pozice odebereme a tím smazání prvku úspěšně končí

3.1.1. Lineární vyhledávání

Lineární vyhledávání, při obsazené primární pozici, vyhledává další možné volné pozice dle následujícího vztahu:

$$H(x,i) = (h(x) + i) mod\ m$$

kde $h(x)$ je výchozí hašovací funkce, $i \in < 0, m-1 >$ je celočíselný parametr a m je velikost hašovací tabulky.

Pokud je primární pozice $H(x,0) = (h(x) + 0) mod\ m$ obsazená, posouváme se na pozici $H(x,1) = (h(x) + 1) mod\ m$. Posouváme se na další pozice, dokud nenalezneme prázdnou pozici nebo se nedostaneme do situace, že jsme již prošli celou tabulku. Této situaci odpovídá pozice $H(x, m-1) = (h(x) + m - 1) mod\ m$. V příkladech používáme hašovací funkci pro řetězce c_1.

Příklad 1. Hašovací tabulka o velikosti $m = 11$ (Viz. obrázek 3.) zobrazuje situaci po vložení řetězců:

h(Milena) = (77 + 105 + 108 + 101 + 110 + 97) mod 11 = **4**
h(Alan) = (65 + 108 + 97 + 110) mod 11 = **6**
h(Filip) = (70 + 105 + 108 + 105 + 112) mod 11 = **5**
h(Jana) = (74 + 97 + 110 + 97) mod 11 = **4**

0	
1	
2	
3	
4	Milena
5	Filip
6	Alan
7	Jana
8	
9	
10	

Obrázek 3. Lineární vyhledávání - hledání volné pozice.

Řetězec *Milena* jsme vložili na jeho primární pozici h(Milena) = 4, řetězec *Alan* na jeho primární pozici h(Alan) = 6, řetězec *Filip* na jeho primární pozici h(Filip) = 5. Při pokusu o vložení řetězce *Jana* na pozici h(Jana) = 4 jsme zjistili, že tato primární pozice je již obsazená, tudíž pokračujeme na pozici 5, ta je taky obsazená. Pokračujeme na pozici 6, která je obsazená. Pokračujeme na pozici 7, která je volná a na kterou řetězec vložíme.

Příklad 2. Hašovací tabulka o velikosti $m = 11$ z Příkladu 1. (Viz. obrázek 4.) zobrazuje situaci po vložení dalších řetězců:

h(Alex) = (65 + 108 + 101 + 120) mod 11 = **9**
h(Marie) = (77 + 97 + 114 + 105 + 101) mod 11 = **10**
h(Pavel) = (80 + 97 + 118 + 101 + 108) mod 11 = **9**

0	Pavel
1	
2	
3	
4	Milena
5	Filip
6	Alan
7	Jana
8	
9	Alex
10	Marie

Obrázek 4. Lineární vyhledávání - hledání volné pozice na konci tabulky.

Řetězec *Alex* jsme vložili na jeho primární pozici h(Alex) = 9, řetězec *Marie* na jeho primární pozici h(Marie) = 10. Při pokusu o vložení řetězce *Pavel* na pozici h(Pavel) = 9 jsme zjistili, že tato primární pozice je již obsazená, tudíž pokračujeme na pozici 10. Tato pozice je obsazená a vzhledem k tomu, že tato pozice je poslední v tabulce, pokračujeme v prohledávání hašovací tabulky od začátku, tedy od pozice 0. Pokračujeme na pozici 0, která je volná a na kterou řetězec vložíme.

Pro lineární vyhledávání je typická situace, při které dochází ke vzniku skupin obsazených řádků. Takové skupiny nazýváme shluky. Shluky prodlužují operace vkládání, vyhledávání i smazání hodnoty z hašovací tabulky.

Na obrázku 4. vidíme shluk obsazených řádků na pozicích 4-7. Při vyhledání řetězce *Jana* s h(Jana) = 4, zjistíme, že primární pozice 4 obsahuje jinou hodnotu, tudíž pokračujeme ve vyhledávání na další pozici. Stejná situace se opakuje na pozicích 5 a 6. A až na pozici 7 je řetězec *Jana* nalezen.

Z toho vyplývá, že nejméně příznivá situace je taková, při které je primární pozice obsazená a zároveň je uvnitř skupiny obsazených řádků.

Následující příklad nám odhalí problém s odstraňováním prvků z hašovací tabulky.

Příklad 3. Hašovací tabulka o velikosti $m = 11$ (Viz. obrázek 5.) zobrazuje situaci před a po odstranění řetězce:

h(Filip) = (70 + 105 + 108 + 105 + 112) mod 11 = **5**

Obrázek 5. Lineární vyhledávání - tabulka před a po odstranění prvku z tabulky.

Řetězec *Filip* jsme nalezli na jeho primární pozici h(Filip) = 5, tudíž jsme ho mohli přímo odebrat, bez jeho hledání na dalších pozicích.

Vyhledáme nyní řetězec *Jana*, kdy postupujeme následovně:
Řetězec *Jana* s h(Jana) = 4 se pokusíme vyhledat na pozici 4, na tuto pozici se podíváme. Zjistíme, že pozice je obsazená jiným prvkem a pokračujeme tudíž ve vyhledáváním na další pozici. Po přesunu na další pozici zjistíme, že tato pozice je prázdná a vyhledávání tímto končí, prvek nebyl nalezen.

Což je ale v rozporu se skutečností, jelikož hašovací tabulka řetězec *Jana* obsahuje a nachází se na pozici 7.

Při odebírání prvků z hašovací tabulky si musíme jejich původní pozice nějak označit, abychom tyto pozice mohli při vyhledávání přeskakovat a při vkládání na takto označené pozice vkládat prvky nové.

Pozice v hašovací tabulce, po odebrání prvku, si můžeme například označit, respektive odstraňovaný řetězec nahradit, řetězcem *smazano* nebo použít nějaký specifický znak(např. #). Následně bychom takto označené pozice při vyhledávání přeskakovali a při vkládání do nich prvky ukládali.

Takové řešení by bylo možné pouze pro ukládání řetězců. Pokud bychom chtěli ukládat jiné datové typy, musíme si navrhnout jiné řešení.

Řešením může být ukládat do hašovací tabulky strukturovaný datový typ, jehož jeden člen by udržoval informaci o typu pozice(PRAZDNA, OBSAZENA, VOLNA) a druhý člen by obsahoval hodnotu(libovolný datový typ).

3.1.2. Kvadratické vyhledávání

Kvadratické vyhledávání, při obsazené primární pozici, vyhledává další možné volné pozice dle následujícího vztahu:

$$H(x,i) = (h(x) + i^2) mod\ m$$

kde $h(x)$ je výchozí hašovací funkce, i je celočíselný parametr a m je velikost hašovací tabulky.

Pokud je primární pozice $H(x,0) = (h(x) + 0^2) mod\ m$ obsazená, posouváme se na pozici $H(x,1) = (h(x)+1^2) mod\ m$. Pokud i tato pozice je obsazená, posouváme se na další pozici $H(x,2) = (h(x) + 2^2) mod\ m$, dokud nenalezneme prázdnou pozici nebo se nedostaneme do situace, že jsme danou pozici již navštívili. V příkladech používáme hašovací funkci pro řetězce c_1.

Příklad 1. Hašovací tabulka o velikosti $m = 11$ (Viz. obrázek 6.) zobrazuje situaci po vložení řetězců:

h(Milena) = (77 + 105 + 108 + 101 + 110 + 97) mod 11 = **4**
h(Irena) = (73 + 114 + 101 + 110 + 97) mod 11 = **0**
h(Marek) = (77 + 97 + 114 + 101 + 107) mod 11 = **1**
h(Ivana) = (73 + 118 + 97 + 110 + 97) mod 11 = **0**

0	Irena
1	Marek
2	
3	
4	Milena
5	
6	
7	
8	
9	Ivana
10	

Obrázek 6. Kvadratické vyhledávání - hledání volné pozice.

Řetězec *Milena* jsme vložili na jeho primární pozici h(Milena) = 4, řetězec *Irena* na jeho primární pozici h(Irena) = 0, řetězec *Marek* na jeho primární pozici h(Marek) = 1. Při pokusu o vložení řetězce *Ivana* na pozici h(Ivana) = 0 jsme zjistili, že tato primární pozice je již obsazená, tudíž pokračujeme na pozici 1, ta je taky obsazená. Pokračujeme na pozici 4, která je obsazená. Pokračujeme na pozici 9, která je volná a na kterou řetězec vložíme.

Příklad 2. Hašovací tabulka o velikosti $m = 11$ (Viz. obrázek 7.) zobrazuje situaci po vložení řetězců:

$\mathrm{h(Milena)} = (77 + 105 + 108 + 101 + 110 + 97) \bmod 11 = \mathbf{4}$
$\mathrm{h(Filip)} = (70 + 105 + 108 + 105 + 112) \bmod 11 = \mathbf{5}$
$\mathrm{h(Adam)} = (65 + 100 + 97 + 109) \bmod 11 = \mathbf{8}$
$\mathrm{h(Jana)} = (74 + 97 + 110 + 97) \bmod 11 = \mathbf{4}$

0	
1	
2	Jana
3	
4	Milena
5	Filip
6	
7	
8	Adam
9	
10	

Obrázek 7. Kvadratické vyhledávání - hledání volné pozice na konci tabulky.

Řetězec *Milena* jsme vložili na jeho primární pozici h(Milena) = 4, řetězec *Filip* na jeho primární pozici h(Filip) = 5, řetězec *Adam* na jeho primární pozici h(Adam) = 8. Při pokusu o vložení řetězce *Jana* na pozici h(Jana) = 4 jsme zjistili, že tato primární pozice je již obsazená, tudíž pokračujeme na pozici 5, ta je taky obsazená. Pokračujeme na pozici 8, která je obsazená. Pokračujeme na pozici 2, která je volná a na kterou řetězec vložíme.

Při kvadratickém vyhledávání již nedochází k tolik častému vzniku skupin obsazených řádků, jako u lineárního vyhledávání, které prodlužují dobu vyhledání a vkládání. Pro kvadratické vyhledávání je typická situace, kde při vyhledání volné pozice pro vložení prvku, při obsazené primární pozici, se dostaneme na pozici, kterou jsme již jednou navštívili. Tato situace je zjevně nežádoucí, jelikož se nám nepodaří vložit prvek do hašovací tabulky i přes volné pozice v tabulce a dojde k zacyklení. Zacyklení můžeme řešit ukládáním navštívených pozic, při vkládání prvků, do pomocného datové struktury a srovnáním s aktuální pozicí.

Při potřebě odstraňovat hodnoty či klíče z hašovací tabulky, je potřeba pozice po odstranění označovat, aby nedocházelo k situacím kdy, přes přítomnost prvku v hašovací tabulce, nelze tento prvek nalézt. Viz. Příklad 3. v kapitole 3.1.1.

3.1.3. Dvojí hašování

Dvojí hašování, při obsazené primární pozici, vyhledává další možné volné pozice dle následujícího vztahu:

$$H(x,i) = (h(x) + i \times h_2(x)) mod\ m$$

kde $h(x)$ je výchozí hašovací funkce, $i \in < 0, m-1 >$ je celočíselný parametr, $h_2(x)$ je sekundární hašovací funkce a m je velikost hašovací tabulky.

Sekundární hašovací funkce $h_2(x)$ je dána následujícím vztahem:

$$h_2(x) = 1 + (c(x) mod\ (m\ \text{-}\ 1))$$

kde $c(x)$ je funkce pro řetězce nabývající hodnot $< 1, m-1 >$ a m je velikost hašovací tabulky.

Pokud je primární pozice $H(x,0) = (h(x) + 0 \times h_2(x)) mod\ m$ obsazená, posouváme se na pozici $H(x,1) = (h(x) + 1 \times h_2(x)) mod\ m$. Posouváme se na další pozice, dokud nenalezneme prázdnou pozici nebo se nedostaneme do situace, že jsme již prošli celou tabulku. Této situaci odpovídá pozice $H(x, m-1) = (h(x) + (m-1) \times h_2(x)) mod\ m$.

Obrázek 8. Dvojí hašování - hledání volné pozice.

Příklad 1. Hašovací tabulka o velikosti $m = 11$ (Viz. obrázek 8.) zobrazuje situaci po vložení řetězců:

h(Milena) = (77 + 105 + 108 + 101 + 110 + 97) mod 11 = **4**
h(Alan) = (65 + 108 + 97 + 110) mod 11 = **6**
h(Filip) = (70 + 105 + 108 + 105 + 112) mod 11 = **5**
h(Jana) = (74 + 97 + 110 + 97) mod 11 = **4**

Řetězec *Milena* jsme vložili na jeho primární pozici h(Milena) = 4, řetězec *Alan* na jeho primární pozici h(Alan) = 6, řetězec *Filip* na jeho primární pozici h(Filip) = 5. Při pokusu o vložení řetězce *Jana* na pozici h(Jana) = 4 jsme zjistili, že tato primární pozice je již obsazená. Pokračujeme na pozici 2, která je volná a na kterou řetězec vložíme.

Při odstraňování prvků z hašovací tabulky je nutné tyto pozice po odstranění označovat. Pro všechny metody Otevřeného hašování tudíž platí stejná pravidla při odstraňování prvků z hašovací tabulky. Viz. Příklad 3. v kapitole 3.1.1.

3.1.4. Perfektní hašování

Perfektní hašování je hašování bez kolizí, pro které platí:

$$h(klíč_1) \neq h(klíč_2) \text{ , pro } \forall klíč_1, klíč_2$$

Dva různé klíče mají různou hodnotu hašovací funkce, ukazují tedy na jinou pozici v hašovací tabulce.

Minimální perfektní hašování je perfektní hašování, pro které platí:

$$h(klíč_1) \neq h(klíč_2), m = n, \text{ pro } \forall klíč_1, klíč_2$$

kde m je velikost hašovací tabulky a n je počet klíčů uložených v hašovací tabulce. Hašovací tabulka je tedy úplně zaplněná, nemá žádné volné pozice na vložení dalšího prvku.

Perfektní hašování se používá při známém počtu prvků, které plánujeme do hašovací tabulky uložit. Pro tyto prvky následně navrhneme bezkolizní hašovací funkci. Návrh bezkolizní funkce je zjevně složitý.

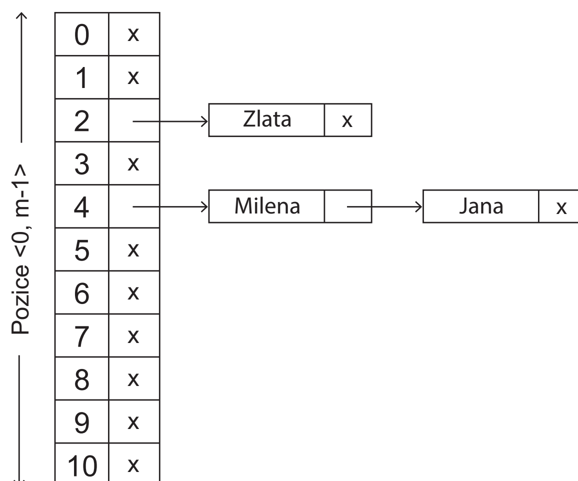
U perfektního hašování tedy neuvažujeme operaci odebírání prvků z hašovací tabulky, to znamená, že po uložení všech prvků do hašovací tabulky již nebudeme žádné další prvky vkládat ani odebírat.

Perfektní hašování se používá např. u překladačů[6] programovacích jazyků k uložení klíčových slov daného jazyka.

[6] Překladač je program převádějící zdrojový kód programu do jiného cílového jazyku

3.2. Zřetězení

Zřetězení řeší kolize ukládáním prvků, se stejnou hodnotou hašovací funkce, do spojového seznamu. Hašovací tabulka obsahuje ukazatele na seznam.

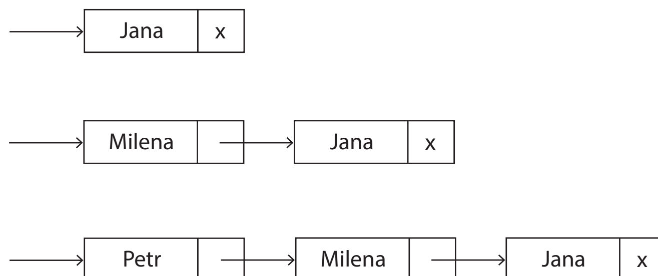


Obrázek 9. Zřetězení - tabulka ukazatelů na seznam.

Na začátku hašování, kdy není vložen žádný prvek, hašovací tabulka obsahuje nulové ukazatele[7], které znamenají, že daná pozice neobsahuje žádný seznam. Při vkládání prvků do hašovací tabulky se na daných pozicích tyto seznamy postupně vytvářejí.

Spojový seznam se skládá z uzlů, kde každý uzel se skládá z ukládané hodnoty či jiné datové struktury a ukazatele na následníka daného uzlu. Uzel obsahuje nulový ukazatel, pokud se jedná o poslední uzel spojového seznamu.

Do spojového seznamu můžeme nové uzly vkládat několika způsoby.

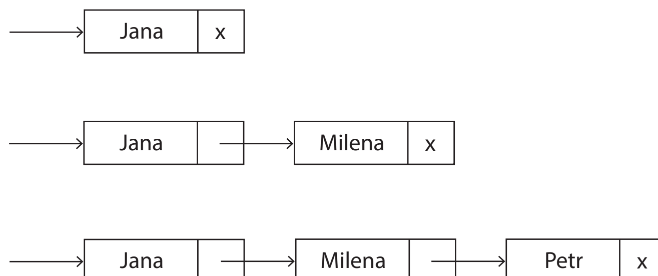


Obrázek 10. Spojový seznam - vkládání na začátek seznamu.

Uzly můžeme vkládat na začátek seznamu (Obrázek 10.), na konec seznamu (Obrázek 11.), na specifickou pozici seznamu nebo můžeme mít seznam setříděný

[7]Konstanta nulového ukazatele - nullptr

podle klíče. V této práci se zaměříme na vkládání nového uzlu na začátek a na konec seznamu.



Obrázek 11. Spojový seznam - vkládání na konec seznamu.

Definované operace v hašovací tabulce se zřetězením:

Vyhledávání prvku: Spočítáme hodnotu hašovací funkce *h(x)*. Výsledkem je pozice v hašovací tabulce, na kterou se podíváme. Mohou nastat následující situace:

1. Pozice obsahuje nulový ukazatel → hledaný prvek tak není v tabulce uložen a vyhledávání prvku neúspěšně končí
2. Procházíme jednotlivé uzly spojového seznamu → pokud prvek není v žádném uzlu uložen, vyhledávání prvku neúspěšně končí jinak je vyhledávání úspěšné

Vkládání prvku: Spočítáme hodnotu hašovací funkce *h(x)* a prvek vyhledáme. Pokud je prvek nalezen, vkládání končí neúspěšně, jelikož prvek může být v hašovací tabulce nejvýše jednou. Jinak pokračujeme následovně:

1. Pozice obsahuje nulový ukazatel → pozice neobsahuje žádný seznam, vytvoříme nový uzel a nastavíme na něj ukazatel
2. Pozice neobsahuje nulový ukazatel → na pozici je seznam, vytvoříme nový uzel a vložíme jej na začátek či konec seznamu, dle zvoleného způsobu vkládání

Smazání prvku: Spočítáme hodnotu hašovací funkce *h(x)* a prvek vyhledáme. Mohou nastat následující situace:

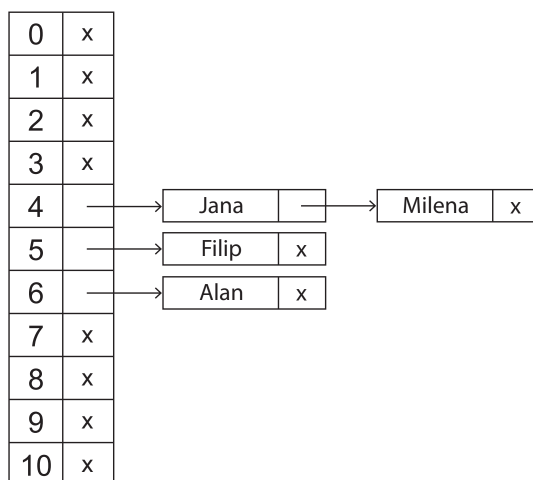
1. Prvek nebyl nalezen → prvek není v hašovací tabulce, nelze jej tedy odebrat, smazání prvku neúspěšně končí
2. Prvek byl nalezen → uzel s prvkem odstraníme, smazání prvku úspěšně končí

Při vkládání uzlu na začátek seznamu přesměrujeme ukazatel na seznam dané pozice tabulky na nový uzel a jako následníka nového uzlu nastavíme předchozí první uzel seznamu, pokud vkládaný uzel není první uzel seznamu. Při vkládání na konec seznamu, pokud vkládaný uzel není první uzel seznamu, nastavíme ukazatel následníka posledního uzlu na nový uzel.

Při odebírání uzlu ze začátku seznamu přesměrujeme ukazatel na seznam dané pozice tabulky na následníka odebíraného uzlu, pokud následníka má, a uzel zrušíme. Při odebírání uzlu z konce seznamu uzel zrušíme a předchůdci nastavíme ukazatel na následníka jako nulový ukazatel, pokud předchůdce má. Pokud odebíraný uzel nemá předchůdce a následníka, čili je to jednoprvkový seznam, nastavíme ukazatel na seznam dané pozice tabulky na nulový ukazatel. Pokud odebíraný uzel má předchůdce a následníka, nastavíme ukazatel na další uzel předchůdce na následníka odebíraného uzlu a uzel zrušíme.

Příklad 1. Hašovací tabulka o velikosti $m = 11$ (Viz. obrázek 12.) zobrazuje situaci po vložení řetězců:

h(Milena) = (77 + 105 + 108 + 101 + 110 + 97) mod 11 = **4**
h(Alan) = (65 + 108 + 97 + 110) mod 11 = **6**
h(Filip) = (70 + 105 + 108 + 105 + 112) mod 11 = **5**
h(Jana) = (74 + 97 + 110 + 97) mod 11 = **4**



Obrázek 12. Zřetězení - vkládání na začátek seznamu.

Řetězec *Milena* jsme vložili do nového uzlu na pozici h(Milena) = 4, řetězec *Alan* do nového uzlu na pozici h(Alan) = 6, řetězec *Filip* do nového uzlu na pozici h(Filip) = 5. Řetězec *Jana* jsme vložili do nového uzlu na pozici h(Jana) = 4, jelikož daná pozice již obsahovala seznam, byl nový uzel vložen na začátek seznamu a následník nového uzlu byl nastaven na uzel s řetězcem *Milena* a ukazatel na seznam dané pozice tabulky byl nastaven na nový uzel.

4. Složitost hašování

Složitost hašování závisí na náročnosti hašovací funkce, kterou ale můžeme považovat za konstantní, a na počtu kolizí při vkládání prvků do hašovací tabulky. Složitost hašování tak s přibývajícími kolizemi roste. Při perfektním hašování, kde se kolize nevyskytují, je složitost evidentně $\Theta(1)$.

Otevřené adresování složitost vyjadřuje poměrem počtu vzniklých kolizí k počtu vložených prvků. Všechny operace v hašovací tabulce tak mají stejnou složitost, která závisí na počtu kolizí.

Zřetězení složitost vyjadřuje průměrnou délku spojových seznamů, do kterých jsou prvky se stejnou hodnotou hašovací funkce vkládány. Při vkládání prvků do spojového seznamu na začátek, je složitost vkládání zjevně $\Theta(1)$.

Při vyhledávání nebo odstraňování prvků ve spojovém seznamu je složitost rovna délce tohoto seznamu, čili při vyhledávání v hašovací tabulce je rovna průměrné délce spojových seznamů.

Vyhledávací metoda	**Složitost**
Hašování	$\Theta(1)$
Lineární vyhledávání v nesetříděném poli	$\Theta(n)$
Binární vyhledávání v setříděném poli	$\Theta(log(n))$
AVL stromy - samovyvažující se binární stromy	$\Theta(log(n))$

Tabulka 2. Vyhledávací metody - složitost.

Tabulka 2. srovnává vybrané metody z hlediska složitosti. Jak můžeme vidět, tak hašování je tedy velmi účinná metoda vyhledávání. V kapitole 6. si ukážeme, že hašování mívá složitost vyšší než $\Theta(1)$ a je závislá na zvolené metodě hašování a faktoru naplnění hašovací tabulky, čili s rostoucím zaplněním hašovací tabulky narůstá složitost hašování.

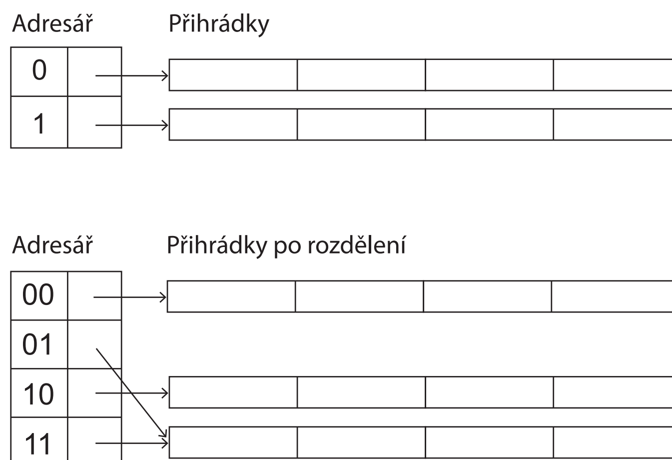
Např. pro 1024 prvků uložených v tabulce, kdy chceme nalézt hledaný prvek, máme při lineárním vyhledávání v nejhorším případě 1024 pokusů vedoucích k nalezení prvku. Při binárním vyhledávání máme těchto pokusů již jen 10, ale je potřeba mít prvky v tabulce setříděné, což přináší další režii. Při hašování prvek většinou nalezneme na několik málo pokusů.

5. Rozšířitelné hašování

Rozšířitelné hašování je dynamické hašování, které nevyžaduje na začátku nastavení velikosti hašovací tabulky. Datová struktura používaná v rozšířitelném hašování se skládá z následujících částí:

- **Adresář**
  Adresář je pole ukazatelů na přihrádky. Jeho velikost se dynamicky mění a přizpůsobuje počtu prvků v hašovací struktuře. Jeho velikost je 2^d, kde proměnná d je přirozené číslo, které se zvyšuje, pokud nastanou určité situace v daných přihrádkách.

- **Přihrádky**
  Přihrádka se používá pro uložení pevného počtu prvků, všechny přihrádky tak mají stejnou neměnnou velikost, kterou definujeme na začátku hašování. Velikost bývá zpravidla mocnina čísla 2.



Obrázek 13. Rozšířitelné hašování.

Hašovací funkce pro rozšířitelné hašování:

$$h(x) = c(x)$$

1. Funkce *c(x)* - je zobrazení $c : x \to \mathbb{N}_0$, funkce tedy zobrazuje hodnotu prvku nebo vyhledávací klíč na nezáporné celé číslo

Definice 1. Značení bitů:

$$...b_{d+1}b_db_{d-1}...b_2b_1$$

kde b_i je bit hodnoty hašovací funkce a b_1 je nejméně významový bit.

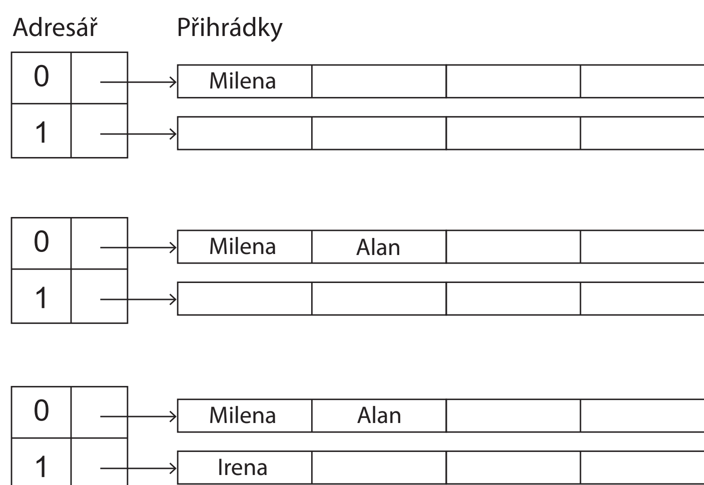
Základem metody je hašovací funkce zobrazující prvek na nezáporné celé číslo, kde použijeme posledních d bitů, které nám určují pozici, na které je daný prvek v přihrádce uložen.

Definované operace v hašovací struktuře:

Vyhledávání prvku: Spočítáme hodnotu hašovací funkce *h(x)*. Výsledkem je nezáporné celé číslo, z něhož vezmeme posledních d bitů určujících pozici v adresáři. Přihrádku na této pozici prohledáme, pokud je prvek nalezen, vyhledávání úspěšně končí.

Vkládání prvku: Spočítáme hodnotu hašovací funkce *h(x)*. Výsledkem je nezáporné celé číslo, z něhož vezmeme posledních d bitů určujících pozici v adresáři. Pokud je prvek v přihrádce obsažen, vkládání neúspěšně končí, prvek může být v hašovací struktuře pouze jednou. Pokud je přihrádka nezaplněná, prvek do ní vložíme, jinak přihrádku rozdělíme.

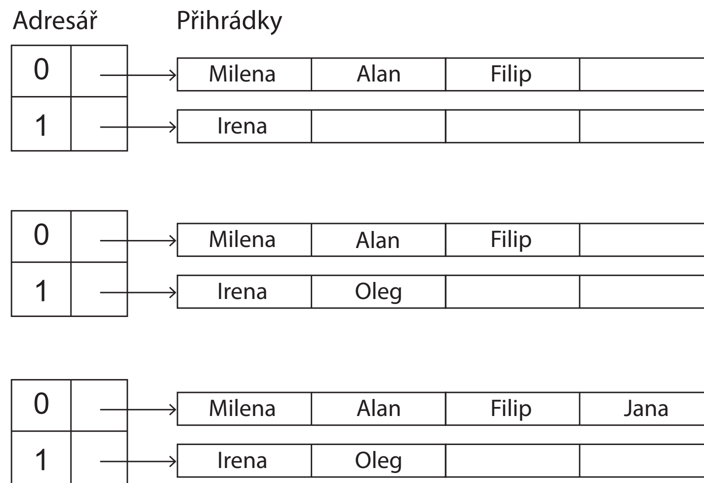
Smazání prvku: Spočítáme hodnotu hašovací funkce *h(x)* a prvek vyhledáme. Pokud byl nalezen, tak jej z dané přihrádky odstraníme. Při odstranění většího počtu prvků z přihrádek mohou vzniknou prázdné přihrádky, které se sloučí.



Obrázek 14. Rozšířitelné hašování - příklad 1a.

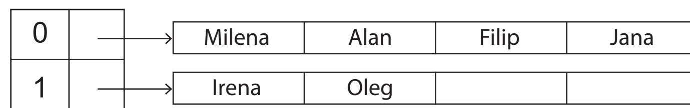
Příklad 1. Hašovací struktura s proměnnou $d = 1$, hašovací funkcí pro řetězce c_1, (Viz. obrázek 14.) zobrazuje situaci po vložení řetězců:

h(Milena) = (77 + 105 + 108 + 101 + 110 + 97) = 598 = 100101011**0**
h(Alan) = (65 + 108 + 97 + 110) = 380 = 10111110**0**
h(Irena) = (73 + 114 + 101 + 110 + 97) = 495 = 11110111**1**



Obrázek 15. Rozšířitelné hašování - příklad 1b.

dále vkládáme:
h(Filip) = (70 + 105 + 108 + 105 + 112) = 500 = 11111010**0**
h(Oleg) = (79 + 108 + 101 + 103) = 391 = 11000011**1**
h(Jana) = (74 + 97 + 110 + 97) = 378 = 10111101**0**
h(Pavel) = (80 + 97 + 118 + 101 + 108) = 504 = 11111100**0**



Obrázek 16. Rozšířitelné hašování - příklad 1c.

Při pokusu o vložení řetězce *Pavel*, kde h(Pavel) = 11111100**0**, došlo k situaci, kdy přihrádka pro uložení byla již zcela zaplněna. Tudíž došlo k rozdělení přihrádky a prvky z této přihrádky byly rozděleny do nově vzniklých přihrádek dle posledních 2 bitů(zvýšili jsme hodnotu proměnné $d = 2$, toto navýšení nám zvětšilo velikost adresáře na hodnotu 4).

Řetězce *Milena* s h(Milena) = 10010101**10** a řetězec Jana, kde h(Jana) = 1011110**10**, byly vloženy na pozici 10.

Zbývající řetězce, dle hodnoty posledních dvou bitů byly uloženy na pozici 00.
h(Alan) = (65 + 108 + 97 + 110) = 380 = 1011111**00**
h(Filip) = (70 + 105 + 108 + 105 + 112) = 500 = 1111101**00**
h(Pavel) = (80 + 97 + 118 + 101 + 108) = 504 = 1111110**00**

Složitost rozšířitelného hašování
Složitost závisí především na velikosti přihrádek pro ukládání prvků. Při vyhledávání vypočteme hodnotu hašovací funkce a provedeme takový počet srovnání, jaký je počet prvků v přihrádce.

Složitost vkládání závisí na obsazenosti dané přihrádky, tedy pokud je zcela zaplněná a musíme přihrádku a následně i adresář zvětšit, složitost přirozeně narůstá, jelikož pak musíme prvky z původní přihrádky umístit do nově vzniklých přihrádek.

Složitost odebírání prvků z hašovací struktury závisí taktéž na počtu prvků v přihrádce, tedy pokud je prázdná a provádíme slučování přihrádek.

Použití rozšířitelného hašování
Hlavní výhoda rozšířitelného hašování spočívá v dynamickém přizpůsobování se velikosti hašovací struktury. Nemusíme velikost hašovací tabulky nastavovat na začátku hašování, tak jako u statického hašování. Nehrozí taktéž vyčerpání kapacity hašovací tabulky. Rozšířitelného hašování se používá při takovém množství prvků, na které nestačí kapacita operační paměti, pak bývá v paměti uložen jen adresář a přihrádky jsou umístěny na pevném disku.

V této práci jsme si popsali Rozšířitelné hašování na teoretické úrovni a nebudeme ho dále implementovat v sestaveném programu, který je vyhrazen pro metody statického hašování.

6. Empirické testy

Empirické testy budeme provádět pro velikosti hašovací tabulky $m = 1001$, $m = 10001$ a $m = 100001$ a pro každou z těchto velikostí hašovací tabulky srovnáme jednotlivé funkce pro řetězce v každé z metod otevřeného hašování a v metodě zřetězení. Do hašovací tabulky budeme postupně ukládat náhodně vygenerované řetězce délky 3 až 20 znaků.

Vygenerovaná množina řetězců může obsahovat duplicitní řetězce. Ale naopak hašovací tabulka může prvek obsahovat pouze jedenkrát, tudíž shodné řetězce jsou vloženy do hašovací tabulky pouze jednou.

Další řetězce, které nemusejí být vloženy do hašovací tabulky, jsou takové, pro které se nenašla volná pozice pro vložení. Typicky u metody Kvadratického hašování, kde může docházet k procházení již jednou navštívených pozic a prvek tak není vložen.

Tabulka s empirickými testy se skládá ze sloupců :

- `Vloženo položek`
  Udává počet úspěšně vložených řetězců do hašovací tabulky
- `Faktor naplnění` λ
  Vyjadřuje poměr počtu obsazených pozicí k velikosti hašovací tabulky
- `Složitost hašování`
  Vyjadřuje poměr počtu kolizí k počtu obsazených pozicí hašovací tabulky

6.1. Velikost tabulky m = 1001

Do hašovací tabulky o velikosti $m = 1001$ vkládáme řetězce a sledujeme nárůst složitosti hašování. Pro všechny funkce je vygenerovaná množina řetězců shodná.

6.1.1. Funkce c_1

Lineární vyhledávání - funkce c1, velikost tabulky 1001

Vloženo položek	Faktor naplnění λ	Složitost hašování
250	0.25	1.11
500	0.50	1.61
750	0.75	3.79
900	0.90	21.03
950	0.95	31.12
1001	1.00	48.93

Tabulka 3. Empirické testy - Lineární vyhledávání - funkce c1, m = 1001

Kvadratické vyhledávání - funkce c1, velikost tabulky 1001

Vloženo položek	**Faktor naplnění λ**	**Složitost hašování**
250	0.25	1.11
500	0.50	1.46
750	0.75	2.25
900	0.90	3.42
950	0.95	4.05
981	0.98	4.60

Tabulka 4. Empirické testy - Kvadratické vyhledávání - funkce c1, m = 1001

Dvojí hašování - funkce c1, velikost tabulky 1001

Vloženo položek	**Faktor naplnění λ**	**Složitost hašování**
250	0.25	1.11
500	0.50	1.37
750	0.75	1.92
900	0.90	2.58
950	0.95	3.23
991	0.99	5.15

Tabulka 5. Empirické testy - Dvojí hašování - funkce c1, m = 1001

Zřetězení - funkce c1, velikost tabulky 1001

Vloženo položek	**Faktor naplnění λ**	**Složitost hašování**
250	0.23	1.10
500	0.40	1.25
750	0.52	1.44
900	0.59	1.53
950	0.61	1.55
1001	0.63	1.60

Tabulka 6. Empirické testy - Zřetězení - funkce c1, m = 1001

Vyhodnocení dosažených výsledků pro funkci c_1 a $m = 1001$:

Lineární vyhledávání

Při zaplnění tabulky z 90% je složitost rovna 21.

Kvadratické vyhledávání

Při zaplnění tabulky z 90% je složitost rovna 3,42.

Dvojí hašování

Při zaplnění tabulky z 90% je složitost rovna 2,58.

Zřetězení

Po vložení všech řetězců do hašovací tabulky jsme ji zaplnili z 63 % a průměrná délka spojového seznamu na těchto pozicích je 1,6. Složitost je tedy 1,6.

6.1.2. Funkce c_2

Lineární vyhledávání - funkce c2, velikost tabulky 1001

Vloženo položek	Faktor naplnění λ	Složitost hašování
250	0.25	1.18
500	0.50	1.46
750	0.75	2.58
900	0.90	4.41
950	0.95	7.47
1001	1.00	19.49

Tabulka 7. Empirické testy - Lineární vyhledávání - funkce c2, m = 1001

Kvadratické vyhledávání - funkce c2, velikost tabulky 1001

Vloženo položek	Faktor naplnění λ	Složitost hašování
250	0.25	1.18
500	0.50	1.45
750	0.75	2.01
900	0.90	2.79
950	0.95	3.23
989	0.99	3.94

Tabulka 8. Empirické testy - Kvadratické vyhledávání - funkce c2, m = 1001

Dvojí hašování - funkce c2, velikost tabulky 1001

Vloženo položek	**Faktor naplnění λ**	**Složitost hašování**
250	0.25	1.15
500	0.50	1.46
750	0.75	1.89
900	0.90	2.42
950	0.95	3.19
994	0.99	5.02

Tabulka 9. Empirické testy - Dvojí hašování - funkce c2, m = 1001

Zřetězení - funkce c2, velikost tabulky 1001

Vloženo položek	**Faktor naplnění λ**	**Složitost hašování**
250	0.22	1.13
500	0.39	1.29
750	0.51	1.46
900	0.59	1.51
950	0.62	1.54
1001	0.63	1.58

Tabulka 10. Empirické testy - Zřetězení - funkce c2, m = 1001

Vyhodnocení dosažených výsledků pro funkci c_2 a $m = 1001$:
Lineární vyhledávání
Při zaplnění tabulky z 90% je složitost rovna 4,41.
Kvadratické vyhledávání
Při zaplnění tabulky z 90% je složitost rovna 2,79.
Dvojí hašování
Při zaplnění tabulky z 90% je složitost rovna 2,42.
Zřetězení
Po vložení všech řetězců do hašovací tabulky jsme ji zaplnili z 63 % a průměrná délka spojového seznamu na těchto pozicích je 1,58. Složitost je tedy 1,58.

6.1.3. Funkce c_3

Lineární vyhledávání - funkce c3, velikost tabulky 1001

Vloženo položek	Faktor naplnění λ	Složitost hašování
250	0.25	1.15
500	0.50	1.39
750	0.75	2.10
900	0.90	3.55
950	0.95	5.16
1001	1.00	11.38

Tabulka 11. Empirické testy - Lineární vyhledávání - funkce c3, m = 1001

Kvadratické vyhledávání - funkce c3, velikost tabulky 1001

Vloženo položek	Faktor naplnění λ	Složitost hašování
250	0.25	1.16
500	0.50	1.35
750	0.75	1.85
900	0.90	2.45
950	0.95	3.00
983	0.98	3.58

Tabulka 12. Empirické testy - Kvadratické vyhledávání - funkce c3, m = 1001

Dvojí hašování - funkce c3, velikost tabulky 1001

Vloženo položek	Faktor naplnění λ	Složitost hašování
250	0.25	1.11
500	0.50	1.30
750	0.75	1.73
900	0.90	2.38
950	0.95	2.99
996	1.00	4.71

Tabulka 13. Empirické testy - Dvojí hašování - funkce c3, m = 1001

Zřetězení - funkce c3, velikost tabulky 1001

Vloženo položek	Faktor naplnění λ	Složitost hašování
250	0.22	1.11
500	0.41	1.23
750	0.54	1.39
900	0.61	1.47
950	0.63	1.51
1001	0.65	1.54

Tabulka 14. Empirické testy - Zřetězení - funkce c3, m = 1001

Vyhodnocení dosažených výsledků pro funkci c_3 a $m = 1001$:
Lineární vyhledávání
Při zaplnění tabulky z 90% je složitost rovna 3,55.
Kvadratické vyhledávání
Při zaplnění tabulky z 90% je složitost rovna 2,45.
Dvojí hašování
Při zaplnění tabulky z 90% je složitost rovna 2,38.
Zřetězení
Po vložení všech řetězců do hašovací tabulky jsme ji zaplnili z 65 % a průměrná délka spojového seznamu na těchto pozicích je 1,54. Složitost je tedy 1,54.

6.2. Velikost tabulky m = 10001

Do hašovací tabulky o velikosti $m = 10001$ vkládáme řetězce a sledujeme nárůst složitosti hašování. Pro všechny funkce je vygenerovaná množina řetězců shodná.

6.2.1. Funkce c_1

Lineární vyhledávání - funkce c1, velikost tabulky 10001

Vloženo položek	Faktor naplnění λ	Složitost hašování
2500	0.25	408.36
5000	0.50	1650.13
7500	0.75	2891.59
9000	0.90	3642.77
9500	0.95	3892.68
10001	1.00	4143.45

Tabulka 15. Empirické testy - Lineární vyhledávání - funkce c1, m = 10001

Kvadratické vyhledávání - funkce c1, velikost tabulky 10001

Vloženo položek	**Faktor naplnění λ**	**Složitost hašování**
2500	0.25	13.56
5000	0.50	33.76
7500	0.75	47.18
9000	0.90	53.81
9500	0.95	55.85
9896	0.99	57.40

Tabulka 16. Empirické testy - Kvadratické vyhledávání - funkce c1, m = 10001

Dvojí hašování - funkce c1, velikost tabulky 10001

Vloženo položek	**Faktor naplnění λ**	**Složitost hašování**
2500	0.25	2.21
5000	0.50	4.27
7500	0.75	7.41
9000	0.90	10.55
9500	0.95	12.53
9975	1.00	17.46

Tabulka 17. Empirické testy - Dvojí hašování - funkce c1, m = 10001

Zřetězení - funkce c1, velikost tabulky 10001

Vloženo položek	**Faktor naplnění λ**	**Složitost hašování**
2500	0.13	1.86
5000	0.17	3.01
7500	0.17	4.31
9000	0.18	5.08
9500	0.18	5.35
10001	0.18	5.61

Tabulka 18. Empirické testy - Zřetězení - funkce c1, m = 10001

Vyhodnocení dosažených výsledků pro funkci c_1 a $m = 10001$:

Lineární vyhledávání

Při zaplnění tabulky z 90% je složitost rovna 3642,77.

Kvadratické vyhledávání

Při zaplnění tabulky z 90% je složitost rovna 53,8.

Dvojí hašování

Při zaplnění tabulky z 90% je složitost rovna 10,55.

Zřetězení

Po vložení všech řetězců do hašovací tabulky jsme ji zaplnili z 18 % a průměrná délka spojového seznamu na těchto pozicích je 5,35. Složitost je tedy 5,35.

Vysoké a neočekávané hodnoty složitosti signalizují špatný návrh funkce pro řetězce c_1. Zjevně dochází k vysokému nárůstu počtu kolizí. U metody Zřetězení vidíme, že došlo k obsazení pouze 18% pozic.

6.2.2. Funkce c_2

Lineární vyhledávání - funkce c2, velikost tabulky 10001

Vloženo položek	Faktor naplnění λ	Složitost hašování
2500	0.25	1.24
5000	0.50	2.11
7500	0.75	10.49
9000	0.90	25.29
9500	0.95	35.41
9997	1.00	122.67

Tabulka 19. Empirické testy - Lineární vyhledávání - funkce c2, m = 10001

Kvadratické vyhledávání - funkce c2, velikost tabulky 10001

Vloženo položek	Faktor naplnění λ	Složitost hašování
2500	0.25	1.22
5000	0.50	1.72
7500	0.75	2.99
9000	0.90	4.32
9500	0.95	5.13
9939	0.99	6.40

Tabulka 20. Empirické testy - Kvadratické vyhledávání - funkce c2, m = 10001

Dvojí hašování - funkce c2, velikost tabulky 10001

Vloženo položek	**Faktor naplnění λ**	**Složitost hašování**
2500	0.25	1.19
5000	0.50	1.47
7500	0.75	2.03
9000	0.90	2.85
9500	0.95	3.51
9994	1.00	7.52

Tabulka 21. Empirické testy - Dvojí hašování - funkce c2, m = 10001

Zřetězení - funkce c2, velikost tabulky 10001

Vloženo položek	**Faktor naplnění λ**	**Složitost hašování**
2500	0.21	1.17
5000	0.37	1.35
7500	0.48	1.56
9000	0.54	1.68
9500	0.55	1.72
9997	0.57	1.76

Tabulka 22. Empirické testy - Zřetězení - funkce c2, m = 10001

Vyhodnocení dosažených výsledků pro funkci c_2 a $m = 10001$:

Lineární vyhledávání

Při zaplnění tabulky z 90% je složitost rovna 25,29.

Kvadratické vyhledávání

Při zaplnění tabulky z 90% je složitost rovna 4,32.

Dvojí hašování

Při zaplnění tabulky z 90% je složitost rovna 2,85.

Zřetězení

Po vložení všech řetězců do hašovací tabulky jsme ji zaplnili z 57 % a průměrná délka spojového seznamu na těchto pozicích je 1,76. Složitost je tedy 1,76.

6.2.3. Funkce c_3

Lineární vyhledávání - funkce c3, velikost tabulky 10001

Vloženo položek	Faktor naplnění λ	Složitost hašování
2500	0.25	1.27
5000	0.50	2.24
7500	0.75	11.54
9000	0.90	26.20
9500	0.95	37.77
10001	1.00	123.34

Tabulka 23. Empirické testy - Lineární vyhledávání - funkce c3, m = 10001

Kvadratické vyhledávání - funkce c3, velikost tabulky 10001

Vloženo položek	Faktor naplnění λ	Složitost hašování
2500	0.25	1.25
5000	0.50	1.78
7500	0.75	3.04
9000	0.90	4.40
9500	0.95	5.21
9926	0.99	6.51

Tabulka 24. Empirické testy - Kvadratické vyhledávání - funkce c3, m = 10001

Dvojí hašování - funkce c3, velikost tabulky 10001

Vloženo položek	Faktor naplnění λ	Složitost hašování
2500	0.25	1.20
5000	0.50	1.49
7500	0.75	2.07
9000	0.90	2.91
9500	0.95	3.54
9999	1.00	8.59

Tabulka 25. Empirické testy - Dvojí hašování - funkce c3, m = 10001

Zřetězení - funkce c3, velikost tabulky 10001

Vloženo položek	Faktor naplnění λ	Složitost hašování
2500	0.21	1.18
5000	0.37	1.37
7500	0.48	1.55
9000	0.53	1.69
9500	0.55	1.73
10001	0.56	1.77

Tabulka 26. Empirické testy - Zřetězení - funkce c3, m = 10001

Vyhodnocení dosažených výsledků pro funkci c_3 a $m = 10001$:
Lineární vyhledávání
Při zaplnění tabulky z 90% je složitost rovna 26,2.
Kvadratické vyhledávání
Při zaplnění tabulky z 90% je složitost rovna 4,4.
Dvojí hašování
Při zaplnění tabulky z 90% je složitost rovna 2,91.
Zřetězení
Po vložení všech řetězců do hašovací tabulky jsme ji zaplnili z 56 % a průměrná délka spojového seznamu na těchto pozicích je 1,77. Složitost je tedy 1,77.

6.3. Velikost tabulky m = 100001

Do hašovací tabulky o velikosti $m = 100001$ vkládáme řetězce a sledujeme nárůst složitosti hašování. Pro všechny funkce je vygenerovaná množina řetězců shodná.

6.3.1. Funkce c_1

Lineární vyhledávání - funkce c1, velikost tabulky 100001

Vloženo položek	Faktor naplnění λ	Složitost hašování
25000	0.25	11629.77
50000	0.50	24125.05
75000	0.75	36625.38
90000	0.90	44123.02
95000	0.95	1412.83
99895	1.00	6075.81

Tabulka 27. Empirické testy - Lineární vyhledávání - funkce c1, m = 100001

Kvadratické vyhledávání - funkce c1, velikost tabulky 100001

Vloženo položek	Faktor naplnění λ	Složitost hašování
25000	0.25	100.37
50000	0.50	145.89
75000	0.75	180.24
90000	0.90	197.95
95000	0.95	203.52
99894	1.00	211.30

Tabulka 28. Empirické testy - Kvadratické vyhledávání - funkce c1, m = 100001

Dvojí hašování - funkce c1, velikost tabulky 100001

Vloženo položek	Faktor naplnění λ	Složitost hašování
25000	0.25	25.04
50000	0.50	55.93
75000	0.75	99.50
90000	0.90	134.47
95000	0.95	152.13
97240	0.97	162.80

Tabulka 29. Empirické testy - Dvojí hašování - funkce c1, m = 100001

Zřetězení - funkce c1, velikost tabulky 100001

Vloženo položek	Faktor naplnění λ	Složitost hašování
25000	0.02	13.57
50000	0.02	26.81
75000	0.02	39.94
90000	0.02	47.77
95000	0.02	50.40
99895	0.02	52.99

Tabulka 30. Empirické testy - Zřetězení - funkce c1, m = 100001

Vyhodnocení dosažených výsledků pro funkci c_1 a $m = 100001$:

Lineární vyhledávání

Při zaplnění tabulky z 90% je složitost rovna 44123,02.

Kvadratické vyhledávání

Při zaplnění tabulky z 90% je složitost rovna 197,95.

Dvojí hašování

Při zaplnění tabulky z 90% je složitost rovna 134,43.

Zřetězení

Po vložení všech řetězců do hašovací tabulky jsme ji zaplnili z 2 % a průměrná délka spojového seznamu na těchto pozicích je 52,99. Složitost je tedy 52.99.

Vysoké a neočekávané hodnoty složitosti signalizují špatný návrh funkce pro řetězce c_1. Zjevně dochází k enormnímu nárůstu počtu kolizí. U metody Zřetězení vidíme, že došlo k obsazení pouze 2% pozic.

6.3.2. Funkce c_2

Lineární vyhledávání - funkce c2, velikost tabulky 100001

Vloženo položek	Faktor naplnění λ	Složitost hašování
25000	0.25	1.29
50000	0.50	3.03
75000	0.75	9.83
90000	0.90	20.05
95000	0.95	32.88
99874	1.00	191.96

Tabulka 31. Empirické testy - Lineární vyhledávání - funkce c2, m = 100001

Kvadratické vyhledávání - funkce c2, velikost tabulky 100001

Vloženo položek	Faktor naplnění λ	Složitost hašování
25000	0.25	1.25
50000	0.50	1.83
75000	0.75	2.87
90000	0.90	4.02
95000	0.95	4.82
99874	1.00	8.63

Tabulka 32. Empirické testy - Kvadratické vyhledávání - funkce c2, m = 100001

Dvojí hašování - funkce c2, velikost tabulky 100001

Vloženo položek	Faktor naplnění λ	Složitost hašování
25000	0.25	1.20
50000	0.50	1.48
75000	0.75	2.00
90000	0.90	2.77
95000	0.95	3.36
99874	1.00	6.91

Tabulka 33. Empirické testy - Dvojí hašování - funkce c2, m = 100001

Zřetězení - funkce c2, velikost tabulky 100001

Vloženo položek	Faktor naplnění λ	Složitost hašování
25000	0.21	1.17
50000	0.37	1.35
75000	0.49	1.54
90000	0.54	1.66
95000	0.56	1.70
99874	0.57	1.74

Tabulka 34. Empirické testy - Zřetězení - funkce c2, m = 100001

Vyhodnocení dosažených výsledků pro funkci c_2 a $m = 100001$:
Lineární vyhledávání
Při zaplnění tabulky z 90% je složitost rovna 20,05.
Kvadratické vyhledávání
Při zaplnění tabulky z 90% je složitost rovna 4,02.
Dvojí hašování
Při zaplnění tabulky z 90% je složitost rovna 2,77.
Zřetězení
Po vložení všech řetězců do hašovací tabulky jsme ji zaplnili z 57 % a průměrná délka spojového seznamu na těchto pozicích je 1,74. Složitost je tedy 1,74.

6.3.3. Funkce c_3

Lineární vyhledávání - funkce c3, velikost tabulky 100001

Vloženo položek	Faktor naplnění λ	Složitost hašování
25000	0.25	1.56
50000	0.50	6.86
75000	0.75	26.78
90000	0.90	54.42
95000	0.95	94.15
99877	1.00	579.33

Tabulka 35. Empirické testy - Lineární vyhledávání - funkce c3, m = 100001

Kvadratické vyhledávání - funkce c3, velikost tabulky 100001

Vloženo položek	Faktor naplnění λ	Složitost hašování
25000	0.25	1.42
50000	0.50	2.49
75000	0.75	4.35
90000	0.90	6.10
95000	0.95	7.26
99877	1.00	11.90

Tabulka 36. Empirické testy - Kvadratické vyhledávání - funkce c3, m = 100001

Dvojí hašování - funkce c3, velikost tabulky 100001

Vloženo položek	Faktor naplnění λ	Složitost hašování
25000	0.25	1.26
50000	0.50	1.63
75000	0.75	2.25
90000	0.90	3.11
95000	0.95	3.79
99873	1.00	7.47

Tabulka 37. Empirické testy - Dvojí hašování - funkce c3, m = 100001

Zřetězení - funkce c3, velikost tabulky 100001

Vloženo položek	**Faktor naplnění λ**	**Složitost hašování**
25000	0.20	1.24
50000	0.33	1.49
75000	0.43	1.76
90000	0.47	1.93
95000	0.48	1.98
99877	0.49	2.04

Tabulka 38. Empirické testy - Zřetězení - funkce c3, m = 100001

Vyhodnocení dosažených výsledků pro funkci c_3 a $m = 100001$:

Lineární vyhledávání

Při zaplnění tabulky z 90% je složitost rovna 54,42.

Kvadratické vyhledávání

Při zaplnění tabulky z 90% je složitost rovna 6,1.

Dvojí hašování

Při zaplnění tabulky z 90% je složitost rovna 3,11.

Zřetězení

Po vložení všech řetězců do hašovací tabulky jsme ji zaplnili z 49 % a průměrná délka spojového seznamu na těchto pozicích je 2,04. Složitost je tedy 2,04.

Výsledky empirických testů v následují kapitole zanalyzujeme a dostaneme tak návod na účinné hašování.

6.4. Porovnání funkcí a hašovacích metod

Porovnejme nyní výsledky empirických testů z pohledu funkcí pro řetězce a z pohledu volby metody hašování. Toto porovnání nám pomůže s výběrem funkce pro řetězce použité v hašovací funkci a výběrem metody hašování s co nejlepší účinností hašování.

6.4.1. Funkce pro řetězce

- **Funkce** c_1
  Empirické testy nám ukázaly na špatný návrh funkce pro řetězce c_1, kdy je tato funkce prakticky nepoužitelná pro velikost hašovací tabulky vyšší než $m = 10001$. Při těchto velikostech hašovací tabulky již dochází k enormnímu narůstání počtu kolizí. Tato funkce se zjevně zobrazuje na velice malý počet nezáporných celých čísel a nesplňuje tedy požadavky na kvalitní hašovací funkci.

- **Funkce** c_2
  Funkce dává příznivé a očekávané výsledky složitosti hašování, její návrh je správný a funkce c_2 je tedy kvalitní.

- **Funkce** c_3
  Funkce dává příznivé a očekávané výsledky složitosti hašování, její návrh je správný a funkce c_3 je tedy kvalitní.

Porovnáním funkcí pro řetězce můžeme učinit závěr, že pro praktické použití jsou použitelné funkce c_2 a c_3. Funkce c_2 je výpočetně náročnější než funkce c_3, ale pro vyšší velikosti hašovací tabulky nabízí lepší hodnoty složitosti.

6.4.2. Metody hašování

Metody máme seřazené dle složitosti z výsledků empirických testů od nejhorší složitosti k nejlepší.

- **Lineární vyhledávání**
  Vzhledem k řešení kolizí lineárním vyhledáváním dalších pozicí a vzniku shluků má tato metody nejhorší složitost.

- **Kvadratické vyhledávání**
  Vzhledem ke kvadratickému vyhledávání dalších pozicí nevznikají shluky v takové míře, ale může nastat problém s nevložení prvku z důvodu zacyklení, tedy že se jednou navštívená pozice prohledává znovu a prvek se nepodaří vložit i přes volné pozice v hašovací tabulce. Složitost je lepší než u Lineárního vyhledávání.

- **Dvojí hašování**
  Použití sekundární hašovací funkce nám tato metoda nabízí o polovinu lepší složitost oproti Kvadratickému vyhledávání.

- **Zřetězení**
  Použitím spojových seznamů k řešení kolizí nám tak Zřetězení poskytuje výborné výsledky složitosti ve srovnání s předchozími metodami hašování. Hlavním předpokladem pro vysokou účinnost této metody je co nejkratší průměrná délka spojových seznamů.

Empirické testy nám ukázaly závislost účinnosti hašování na volbě funkce pro řetězce a volbě metody hašování. Dalším důležitým znakem účinnosti je faktor naplnění λ, ze kterého vyplývá, že pokud je hašovací tabulka zaplněna nad 70%, dochází k nárůstu počtu kolizí u metod Otevřeného adresování. Pokud se tedy budeme snažit tabulky zaplňovat ze 70%, nastává situace typická pro hašování a to je plýtvání pamětí.

Zřetězení nám poskytlo nejlepší výsledky složitosti, ale při bližším pohledu na výsledky zjistíme, že hašovací tabulky byly při různých velikostech a různých funkcích pro řetězce v rozsahu zaplnění 50-60%. Zde tedy dochází k vyššímu plýtvání paměti, jelikož se prvky se stejnou hodnotou hašovací funkce ukládají do spojových seznamů. Další zjištěnou vlastností Zřetězení z provedených empirických testů je rychlost vkládání prvků do hašovací tabulky, kdy tato metoda prvky vkládá mnohonásobně rychleji než metody otevřeného adresování.

Zřetězení nám poskytuje možnost uložit do hašovací tabulky větší počet prvků než je velikost tabulky. Což je zjevně důsledek použití spojových seznamů k řešení kolizí. Při přeplnění hašovací tabulky pak vzrůstá složitost hašování s narůstající průměrnou délkou spojových seznamů.

Pro nejúčinnější hašování je tedy důležité mít k dipozici nebo si navrhnout kvalitní hašovací funkci a použít účinnou metodu hašování. Statické hašování se používá při práci s hašovací tabulkou uloženou v operační paměti.

7. Implementace

Pro naprogramování aplikace bylo použito vývojové prostředí Visual Studio 2013 a verze běhového prostředí .NET Framework 4.5 pro C++/CLI wrapper.

7.1. Programátorská část

Programátorská část popisuje aplikační logiku naprogramovanou v jazyce C++ a dynamickou knihovnu naprogramovanou v jazyce C++/CLI, sloužící pro použití jako wrapper pro jazyky platformy .NET.

7.1.1. Aplikační logika

Popisuje třídy aplikační logiky, jejich metody, výčtové typy a struktury.

Třída BaseHashingClass
Základní abstraktní třída, od které ostatní třídy dědí.

Datové členy

- `const unsigned int tableSize`
  Velikost hašovací tabulky
- `unsigned int items = 0`
  Počet položek hašovací tabulky
- `unsigned int collisions = 0`
  Počet kolizí hašovací tabulky
- `enum StrFunction strFunction`
  Výčtový typ - definuje funkci pro řetězce

Metody

- `BaseHashingClass(unsigned int, StrFunction)`
  Parametrický konstruktor třídy
- `~BaseHashingClass()`
  Destruktor třídy
- `virtual int Search(string) = 0`
  Čistě vituální metoda - definice náleží potomkům
- `virtual bool Insert(string) = 0`
  Čistě vituální metoda - definice náleží potomkům
- `virtual bool Delete(string) = 0`
  Čistě vituální metoda - definice náleží potomkům

- `unsigned int GetTableSize()`
  Vrací velikost hašovací tabulky
- `unsigned int ItemsCount()`
  Vrací počet položek hašovací tabulky
- `unsigned int CollisionsCount()`
  Vrací počet kolizí hašovací tabulky
- `void ItemsCountInc()`
  Inkrementuje počet položek
- `void ItemsCountDec()`
  Dekrementuje počet položek
- `double Difficulty()`
  Vrací složitost hašování
- `void CollisionsCountInc()`
  Inkrementuje počet kolizí
- `void CollisionsCountDec()`
  Dekrementuje počet kolizí
- `double Lambda()`
  Vrací faktor naplnění
- `enum StrFunction GetStrFunction()`
  Vrací použitou hašovací funkci pro řetězce

Výčtové typy

- `enum StrFunction`
  Definuje fukce pro řetězce

Třída OpenAddressing

Abstraktní třída pro otevřené adresování, dědí od třídy BaseHashingClass.

Datové členy

- `struct HashEntry * hashingTable`
  Ukazatel na hašovací tabulku

Metody

- `OpenAddressing(unsigned int, StrFunction)`
  Parametrický konstruktor třídy

- `~OpenAddressing()`
  Destruktor třídy
- `virtual int Search(string) = 0`
  Čistě vituální metoda - definice náleží potomkům
- `virtual bool Insert(string) = 0`
  Čistě vituální metoda - definice náleží potomkům
- `virtual bool Delete(string) = 0`
  Čistě vituální metoda - definice náleží potomkům
- `void PrintTable()`
  Tisk hašovací tabulky
- `bool FullTable()`
  Test zaplnění hašovací tabulky
- `struct HashEntry * HashTable()`
  Vrací ukazatel na hašovací tabulku

Výčtové typy

- `enum ENTRYTYPE`
  Definuje typ pozice v hašovací tabulce

Struktury

- `struct HashEntry`
  Definuje datový typ ukládaný do hašovací tabulky

Třída Chaining

Třída pro zřetězení, dědí od třídy BaseHashingClass.

Datové členy

- `struct LinkedHashEntry ** hashingTable`
  Ukazatel na hašovací tabulku (ukazatel na tabulku ukazatelů na LinkedHashEntry)
- `const bool addFirst`
  Definuje způsob vkládání uzlů do spojového seznamu

Metody

- `Chaining(const unsigned int, StrFunction, bool addFirst = true)`
  Parametrický konstruktor třídy s výchozím parametrem

- `~Chaining()`
  Destruktor třídy - odstraňuje jednotlivé uzly spojového seznamu a hašovací tabulku
- `int Search(string)`
  Vyhledává řetězec v hašovací tabulce
- `bool Insert(string)`
  Vkládá řetězec do hašovací tabulky
- `bool Delete(string)`
  Odstraňuje řetězec z hašovací tabulky
- `unsigned int ItemsCount()`
  Vrací počet položek hašovací tabulky, přepisuje zděděnou metodu
- `void PrintTable()`
  Tisk hašovací tabulky

Struktury

- `struct LinkedHashEntry`
  Definuje uzel spojového seznamu

Třída LinearProbing

Třída pro lineární vyhledávání, dědí od třídy OpenAddressing.

Metody

- `LinearProbing(const unsigned int, StrFunction)`
  Parametrický konstruktor třídy
- `~LinearProbing()`
  Destruktor třídy
- `int Search(string)`
  Vyhledává řetězec v hašovací tabulce
- `bool Insert(string)`
  Vkládá řetězec do hašovací tabulky
- `bool Delete(string)`
  Odstraňuje řetězec z hašovací tabulky

Třída QuadraticProbing

Třída pro kvadratické vyhledávání, dědí od třídy OpenAddressing.

Metody

- `QuadraticProbing(const unsigned int, StrFunction)`
  Parametrický konstruktor třídy
- `~QuadraticProbing()`
  Destruktor třídy
- `int Search(string)`
  Vyhledává řetězec v hašovací tabulce
- `bool Insert(string)`
  Vkládá řetězec do hašovací tabulky
- `bool Delete(string)`
  Odstraňuje řetězec z hašovací tabulky

Třída DoubleHashing

Třída pro dvojí hašování, dědí od třídy OpenAddressing.

Metody

- `DoubleHashing(const unsigned int, StrFunction)`
  Parametrický konstruktor třídy
- `~DoubleHashing()`
  Destruktor třídy
- `int Search(string)`
  Vyhledává řetězec v hašovací tabulce
- `bool Insert(string)`
  Vkládá řetězec do hašovací tabulky
- `bool Delete(string)`
  Odstraňuje řetězec z hašovací tabulky

Třída HashFunction

Třída pro zobrazení řetězce na celé číslo.

Metody

- `HashFunction()`
  Konstruktor třídy
- `~HashFunction()`
  Destruktor třídy
- `unsigned int FunctionC1(string)`
  Funkce pro řetězce

- `unsigned int FunctionC2(string, unsigned int)`
  Funkce pro řetězce

- `unsigned int FunctionC3(string, unsigned int)`
  Funkce pro řetězce

7.1.2. C++/CLI wrapper

Popisuje třídy C++/CLI wrapperu použitelného pro jazyky platformy .NET, jejich metody, výčtové typy a struktury.

Třída LinearProbing
Třída pro lineární vyhledávání.

Metody

- `LinearProbing(unsigned int, HashWrapper::MyStrFunction))`
  Parametrický konstruktor třídy, vytváří instanci třídy aplikační logiky

- `~LinearProbing()`
  Destruktor třídy

- `int Search(String^)`
  Vyhledává řetězec v hašovací tabulce

- `bool Insert(String^)`
  Vkládá řetězec do hašovací tabulky

- `bool Delete(String^)`
  Odstraňuje řetězec z hašovací tabulky

- `string GetString(String^)`
  Převádí řetězec na std::string

- `double Lambda()`
  Vrací faktor naplnění hašovací tabulky

- `double Difficulty()`
  Vrací složitost hašovací tabulky

- `int ItemsCount()`
  Vrací počet položek hašovací tabulky

Datové členy

- `HM::LinearProbing * linearProbing`
  Ukazatel na třídu aplikační logiky

Výčtové typy

- `enum MyStrFunction`
  Definuje fukce pro řetězce

Třída QuadraticProbing

Třída pro kvadratické vyhledávání.

Metody

- `QuadraticProbing(unsigned int, HashWrapper::MyStrFunction))`
  Parametrický konstruktor třídy, vytváří instanci třídy aplikační logiky

- `~QuadraticProbing()`
  Destruktor třídy

- `int Search(String^)`
  Vyhledává řetězec v hašovací tabulce

- `bool Insert(String^)`
  Vkládá řetězec do hašovací tabulky

- `bool Delete(String^)`
  Odstraňuje řetězec z hašovací tabulky

- `string GetString(String^)`
  Převádí řetězec na std::string

- `double Lambda()`
  Vrací faktor naplnění hašovací tabulky

- `double Difficulty()`
  Vrací složitost hašovací tabulky

- `int ItemsCount()`
  Vrací počet položek hašovací tabulky

Datové členy

- `HM::QuadraticProbing * quadraticProbing`
  Ukazatel na třídu aplikační logiky

Výčtové typy

- `enum MyStrFunction`
  Definuje fukce pro řetězce

Třída DoubleHashing

Třída pro dvojí hašování.

Metody

- `DoubleHashing(unsigned int, HashWrapper::MyStrFunction))`
  Parametrický konstruktor třídy, vytváří instanci třídy aplikační logiky
- ~`DoubleHashing()`
  Destruktor třídy
- `int Search(String^)`
  Vyhledává řetězec v hašovací tabulce
- `bool Insert(String^)`
  Vkládá řetězec do hašovací tabulky
- `bool Delete(String^)`
  Odstraňuje řetězec z hašovací tabulky
- `string GetString(String^)`
  Převádí řetězec na std::string
- `double Lambda()`
  Vrací faktor naplnění hašovací tabulky
- `double Difficulty()`
  Vrací složitost hašovací tabulky
- `int ItemsCount()`
  Vrací počet položek hašovací tabulky

Datové členy

- `HM::DoubleHashing * doubleHashing`
  Ukazatel na třídu aplikační logiky

Výčtové typy

- `enum MyStrFunction`
  Definuje fukce pro řetězce

Třída Chaining

Třída pro zřetězení.

Metody

- `Chaining(unsigned int, HashWrapper::MyStrFunction))`
  Parametrický konstruktor třídy, vytváří instanci třídy aplikační logiky

- `~Chaining()`
  Destruktor třídy

- `int Search(String^)`
  Vyhledává řetězec v hašovací tabulce

- `bool Insert(String^)`
  Vkládá řetězec do hašovací tabulky

- `bool Delete(String^)`
  Odstraňuje řetězec z hašovací tabulky

- `string GetString(String^)`
  Převádí řetězec na std::string

- `double Lambda()`
  Vrací faktor naplnění hašovací tabulky

- `double Difficulty()`
  Vrací složitost hašovací tabulky

- `int ItemsCount()`
  Vrací počet položek hašovací tabulky

Datové členy

- `HM::Chaining * chaining`
  Ukazatel na třídu aplikační logiky

Výčtové typy

- `enum MyStrFunction`
  Definuje fukce pro řetězce

7.1.3. Generátor řetězců

Popisuje třídu pro generování řetězců, její metody a datové členy.

Třída Generator
Třída pro generování řetězců.

Metody

- `Generator(unsigned int, unsigned int, unsigned int)`
  Parametrický konstruktor třídy

- `~Generator()`
  Destruktor třídy

- `string GenerateString(unsigned int)`
  Generuje řetězec dané délky

- `int GenerateNumber(unsigned int, unsigned int)`
  Generuje náhodné číslo daného rozsahu.

Datové členy

- `int from`
  Minimální délka řetězce

- `int to`
  Maximální délka řetězce

- `int length`
  Počet řetězců

- `vector<string> str`
  Vektor vygenerovaných řetězců

7.2. Uživatelská část

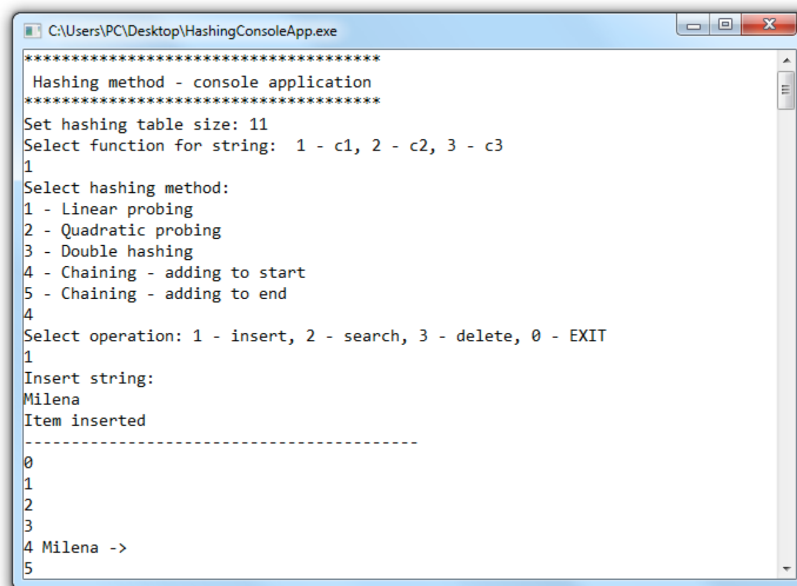
Uživatelská část popisuje použití a práci s naprogramovanou aplikací.

7.2.1. Aplikační logika

K demonstraci aplikační logiky slouží konzolová aplikace naprogramovaná v jazyce C++. Jedná se o program *HashingConsoleApp.exe*.

Ovládání aplikace

Po spuštění aplikace *HashingConsoleApp.exe* je uživatel vyzván k zadání velikosti hašovací tabulky, následuje výběr funkce pro řetězce a výběr metody hašování. Poté již můžeme, výběrem operace, do hašovací tabulky o zadané velikosti řetězce vkládat, vyhledávat nebo je odstraňovat.



Obrázek 17. Výpis programu HashingConsoleApp.exe

Konzolová aplikace *HashingConsoleApp.exe* nám po každé operaci vkládání, vyhledávání nebo odstraňování vytiskne na obrazovku informaci o úspěšném či neúspěšném provedení dané operace a vytiskne vizuální zobrazení hašovací tabulky.

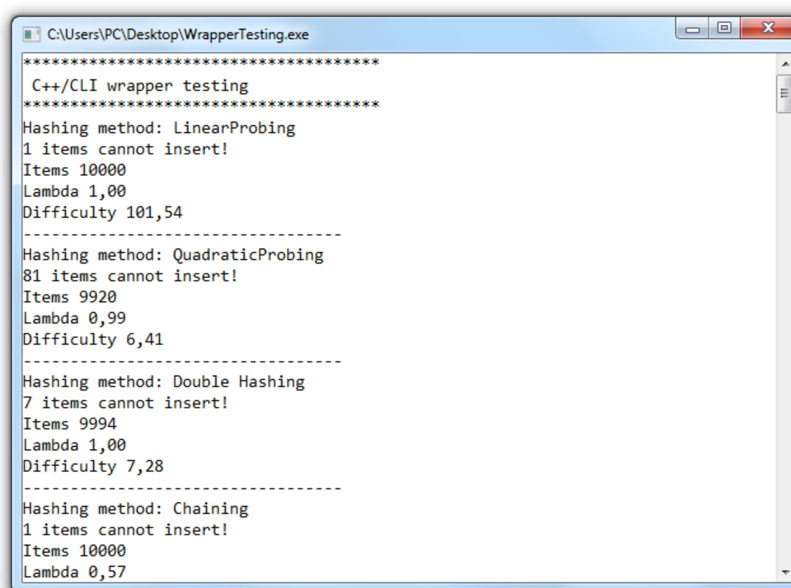
Můžeme sledovat přidávání řetězců do hašovací tabulky dle zvolené metody hašování. Vzhledem k omezenému počtu řádků konzoly tímto doporučujeme pro demonstraci metod hašování používat hašovací tabulky o velikostech nižších (např. velikost $m = 11$).

7.2.2. C++/CLI wrapper

Dynamická knihovna *HashWrapper.dll* slouží k využívání aplikační logiky v jazycích na platformě .Net.

Použití knihovny

Do projektu vytvořeného v jazyku .NET dynamickou knihovnu přidáme volbou z menu vývojového prostředí PROJECT → Add Reference, v okně Reference Manager dynamickou knihovnu vyhledáme a svou volbu potvrdíme. Nyní můžeme používat všechny třídy z této knihovny a jejich veřejné metody.



Obrázek 18. Výpis programu WrapperTesting.exe

Konzolová aplikace, naprogramovaná v jazyku C#, *WrapperTesting.exe* využívá knihovnu *HashWrapper.dll*. *WrapperTesting.exe* načítá soubor 10001.txt, který obsahuje 10001 vygenerovaných řetězců umístěný ve stejném adresáři.

Aplikace řetězce načte a pro každou metodu hašování jej vloží do hašovací tabulky o velikosti $m = 10001$. Výpise programu ukazuje obrázek 18., kde pro každou metodu hašování máme informace o vložených položkách, faktoru naplnění a složitosti dané hašovací metody.

7.2.3. Empirické testy

Empirické testy jsou konzolová aplikace naprogramovaná v jazyce C++. Jedná se o program *EmpiricalTesting.exe*, který slouží ke srovnání jednotlivých metod hašování v závislosti na zvolené funkci pro řetězce.

```
C:\Users\PC\Desktop\EmpiricalTesting.exe
***********************************************************
 Empirical testing of hashing method - console application
***********************************************************
Set hashing table size: 1001
File was created
Select function for string: 1 - c1, 2 - c2, 3 - c3 : 2
****************************************************
Hashing method: LinearProbing

Items = 250 Lambda = 0.25 Difficulty = 1.19
Items = 500 Lambda = 0.50 Difficulty = 1.49
Items = 750 Lambda = 0.75 Difficulty = 2.33
Items = 900 Lambda = 0.90 Difficulty = 4.72
Items = 950 Lambda = 0.95 Difficulty = 12.01
Items = 1001 Lambda = 1.00 Difficulty = 29.87
0 item cannot insert!
****************************************************

****************************************************
Hashing method: QuadraticProbing

Items = 250 Lambda = 0.25 Difficulty = 1.20
Items = 500 Lambda = 0.50 Difficulty = 1.44
Items = 750 Lambda = 0.75 Difficulty = 1.96
Items = 900 Lambda = 0.90 Difficulty = 2.69
```

Obrázek 19. Výpis programu EmpiricalTesting.exe

Ovládání aplikace

Po spuštění aplikace je uživatel vyzván k zadání velikosti hašovací tabulky. Poté dojde k vygenerování řetězců délky 3-20 znaků, které jsou uloženy v souboru 1001.txt (např. pro velikost hašovací tabulky 1001). Následuje výběr funkce pro řetězce. Po tomto výběru se spustí vlastní empirické testy pro všechny metody Otevřeného adresování a pro metodu Zřetězení.

Výstupem aplikace je výpis na obrazovku a uložení výsledku testu do souboru s názvem Empirical-testing-function-c2-table-size-1001.txt (např. pro funkci c2 a velikost hašovací tabulky 10001). Výstup do souboru i na obrazovku obsahuje tabulky jednotlivých metod, které vyjadřují počet vložených položek, dále faktor naplnění a složitost hašování uspořádaných dle postupného zaplňování.

Při každém spuštění konzolové aplikace *EmpiricalTesting.exe* se vygeneruje jiná množina řetězců, nicméně testy jsou prováděny pro všechny uvedené metody hašování nad jednou touto množinou řetězců. Výsledky empirických testů jsou pro každou množinu velmi podobné.

Závěr

V této práci jsme si představili hašování jako vysoce účinnou metodu vyhledávání. Popsali jsme si použité datové struktury pro hašování, navrhli jsme si funkce pro řetězce použité v hašovacích funkcích. U jednotlivých metod hašování jsme si vysvětlili jejich funkcionalitu a způsoby řešení kolizí. Empirické testy nám metody hašování vzájemně porovnaly, funkce pro řetězce a jejich návrh jsme si taktéž podrobně vysvětlili a porovnali. Na dynamické hašování jsme teoreticky nahlédli. Implementovali jsme aplikační logiku, C++/CLI wrapper pro použití na platformě .NET, vytvořili jsme aplikaci pro empirické testy.

Nyní již víme jak hašování funguje, jakou zvolit funkci pro řetězce a jakou metodu hašování. Jako vylepšení aplikace se nabízí možnost ukládání libovolného datového typu či struktury do hašovací tabulky za použití šablon v programovacím jazyku C++, jelikož jsme se v této práci věnovali pouze ukládání řetězců.

Reference

[1] Sedgewick, Robert. *Algorithms in C++: Parts 1-4: Fundamentals, Data Structure, Sorting, Searching.* Addison-Wesley, Massachusetts , 1998. ISBN 0-201-35088-2.

[2] Sedgewick, Robert. *Algoritmy v C, části 1-4.* SoftPress, Praha, 2003. ISBN 80-86497-56-9.

[3] Weiss, Mark Allen. *Data Structures and Algorithm Analysis in C++.* Addison-Wesley, Massachusetts , 2014. ISBN 0-13-284737-X.

[4] Prata, Stephen. *Mistrovství v C++.* Computer Press, Brno, 2013. ISBN 978-80-251-3828-1.

[5] Nagel, Christian. *C# 2008 Programujeme profesionálně.* Computer Press, Brno, 2009. ISBN 978-80-251-2401-7.

[6] Sharp, John. *Visual C# 2010 krok za krokem.* Computer Press, Brno, 2012. ISBN 978-80-251-3147-3.

[7] Večerka, Arnošt. *Základní algoritmy.* Elektronická publikace, 2007.

[8] Zhang, Donghui. *Extendible Hashing.* Elektronická publikace, 2008.

[9] Moura, Lucia. *Extendible Hashing.* Elektronická publikace, 2002.

A. První příloha

Níže uvedené přílohy se nacházejí na přiloženého CD ve složce `doc/`.

- **Diagramy tříd**
  Popisují jednotlivé třídy a případné vztahy mezi nimi
- **Diagramy případů užití**
  Popisují případy užití

B. Obsah přiloženého CD

Adresářová struktura obsahu přiloženého CD.

`bin/`
: Programy spustitelné přímo z CD. Adresář obsahuje i všechny potřebné knihovny a další soubory pro bezproblémové spuštění programu.

`doc/`
: Adresář s dokumentací práce ve formátu PDF, diagramy tříd, diagramy případů užití, zdrojový text dokumentace, vložené obrázky a ilustrace ve formátu EPS.

`src/`
: Adresář obsahuje zdrojové texty programů.

`readme.txt`
: Instrukce pro spuštění programů s požadavky pro účel obhajoby práce.