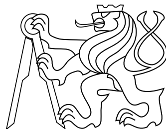
Rozptylovací tabulky
Hash tables

Jan Kybic

http://cmp.felk.cvut.cz/~kybic
kybic@fel.cvut.cz

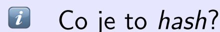
2016



Rozptylovací tabulka
Hash table

Rozptylovací tabulka = implementace množiny / asociativního pole

+ velmi rychlé vkládání i hledání, $O(1)$
- neudržuje uspořádání (hledání maxima/minima)
- méně efektivní využití paměti



Co je to *hash*?

- *hash* — rozemlít, rozsekat, sekané maso, haše, . . . hašiš
- *hash function* — rozptylovací/transformační/hašovací/hešovací/ funkce: objekt → celé číslo
- *hash / fingerprint* — haš/heš, otisk

Rozptylovací tabulka
(Hash table)

Základní myšlenky a vlastnosti

- pole m přihrádek *(slots)* pro ukládání položek.
- položka *(item)* = klíč *(key)* + hodnota *(value)*
- klíč je unikátní
- **rozptylovací funkce** *(hash function)* :
  φ: klíč $\rightarrow$ číslo přihrádky $0 \dots m-1$
- více položek v jedné přihrádce = **kolize** *(collision/clash)*
- operace jsou rychlé, protože
  - víme, v které přihrádce hledat
  - v každé přihrádce je jen omezený počet položek

Relativní naplnění tabulky
(load factor)

Průměrný počet položek na přihrádku

$$\text{load factor } \lambda = \frac{\text{počet položek } n}{\text{počet přihrádek } m}$$

- velké λ $\rightarrow$ hodně kolizí $\rightarrow$ zpomalení operací
- malé λ $\rightarrow$ hodně prázdných položek $\rightarrow$ nevyužitá paměť

Příklad

$m = 11$ přihrádek, rozptylovací funkce $\varphi(x) = x \bmod m =$ `x % m`

Vložíme čísla

x	54	26	93	17	77	31
$\varphi(x)$	10	4	5	6	0	9

Vznikne tabulka

0	1	2	3	4	5	6	7	8	9	10
77				26	93	17			31	54

Relativní naplnění $\lambda = 6/11 \approx 0.54$

Rozptylovací funkce
Hash function

Nutné vlastnosti

- 'Stejné' klíče musí mít stejný otisk — $x = y \Rightarrow \varphi(x) = \varphi(y)$
- Neměnnost / nenáhodnost / konstantnost / opakovatelnost

Požadované vlastnosti

- Rychlost výpočtu
- 'Různé' klíče mají mít pokud možno různý otisk — $x \neq y \Rightarrow$ velká $P\left[\varphi(x) \neq \varphi(y)\right]$
  - každý klíč jiný otisk = *perfect hashing*
  - rovnoměrné využití všech přihrádek
  - pravděpodobnost zvolení konkrétní přihrádky $1/m$ (i pro strukturované vstupy)
  - malé množství kolizí

Kvalitu lze ověřit experimentálně.
Souvislost s kryptografií a náhodnými čísly.

Rozptylovací funkce

- Pro celá čísla $\varphi(x) = x \bmod m =$ `x % m`
- Pro znaky `ord(c) % m`
- Pro k-tice

$$\varphi\big((x_1, x_2, \ldots, x_k)\big) = \sum_{i=1}^{k} x_i p^{i-1} \bmod m$$

kde p je vhodné prvočíslo — dostatečně velké a nesoudělné s m.

Rozptylovací funkce

- Pro celá čísla $\varphi(x) = x \bmod m =$ `x % m`
- Pro znaky `ord(c) % m`
- Pro k-tice

$$\varphi\big((x_1, x_2, \ldots, x_k)\big) = \sum_{i=1}^{k} x_i p^{i-1} \bmod m$$

kde p je vhodné prvočíslo — dostatečně velké a nesoudělné s m.

```
def hash_string(x,m):
  h=0
  for c in x:
    h=((h*67)+ord(c)) % m
  return h
```

Soubor `hashing.py`.

Rozptylovací funkce v Pythonu

Funkce `hash` — pro neměnné hodnoty *(immutable)*: čísla, řetězce, *n*-tice, logické hodnoty, funkce, neměnné množiny (`frozenset`), objekty...
nikoliv pro pole, množiny (`set`)

Vrací (velké) celé číslo.

```
print(hash(34))
```

```
34
```

```
print(hash("les"))
```

```
7824003431697358632
```

```
print(hash((7,"pes")))
```

```
-4517796161293337072
```

Rozptylovací funkce v Pythonu

Funkce `hash` — pro neměnné hodnoty *(immutable)*: čísla, řetězce, n-tice, logické hodnoty, funkce, neměnné množiny (`frozenset`), objekty…
nikoliv pro pole, množiny (`set`)

Vrací (velké) celé číslo.

```
print(hash(34))
```

```
34
```

```
print(hash("les"))
```

```
7824003431697358632
```

```
print(hash((7,"pes")))
```

```
-4517796161293337072
```

Používáme `hash(x) % m`.
V Pythonu `y % m` ≥ 0 pokud $m > 0$.

Další použití rozptylovacích funkcí

Rychlé ověřené rovnosti velkých objektů (DNA řetězce, otisky prstů, obrázky, ...):

- Předpočítej otisk každého objektu v databázi
- Pokud `hash(x)=hash(y)`, pokračuj úplným porovnáním `x` a `y`

Velikost rozptylovací tabulky

- Vhodná velikost je prvočíselná — např. $11, 103, 1009 \ldots$
  - Jinak riziko kolizí pokud $\varphi(x) \in \{k, 2k, 3k, \ldots\}$
- Dynamická realokace:
  - pokud se tabulka naplní ($\lambda > \lambda_{\mathsf{max}}$) — vytvoříme větší tabulku ($m' \approx 2m$)
  - pokud se tabulka vyprázdní ($\lambda < \lambda_{\mathsf{min}}$) — vytvoříme menší tabulku ($m' \approx m/2$)

Možné hodnoty $m_0 = 11$, $\lambda_{\mathsf{max}} = 0.75$, $\lambda_{\mathsf{min}} = 0.25$.

Nalezení prvočíselné velikosti

Najde první prvočíslo větší než n. Pokud takové není, vrátí n a vypíše varování.

```
primes=prvocisla_eratosthenes(100000)

def find_prime_size(n):
  for i in range(len(primes)):
    if primes[i]>n:
      return n
  print("Pozor, tabulka prvočísel je příliš krátká.")
  return n
```

Zrychlování

- Tabulku (vybraných) prvočísel lze předpočítat.
- Vyhledávání lze zrychlit binárním půlením.
- Prvočísla nejsou potřeba všechna.

Soubor `hashing.py`.

Řešení kolizí

Co když dvě položky mají stejný otisk?

- **Zřetězení** *(chaining)*
  - Každá přihrádka je seznam *(nebo pole)*.
  - Zaplnění λ může být > 1.

Řešení kolizí

Co když dvě položky mají stejný otisk?

- **Zřetězení** *(chaining)*
  - Každá přihrádka je seznam *(nebo pole)*.
  - Zaplnění λ může být > 1.
- **Otevřené adresování** *(open addressing)*
  - Kapacita přihrádky je 1. Pokud je přihrádka $m_0 = \varphi(x)$ obsazená, zkusíme jinou (m_1, m_2, ...)
  - Lineární zkoušení *(linear probing)* — zkusíme $m_i = m_0 + i$.

Řešení kolizí

Co když dvě položky mají stejný otisk?

- **Zřetězení** *(chaining)*
  - Každá přihrádka je seznam *(nebo pole)*.
  - Zaplnění λ může být > 1.
- **Otevřené adresování** *(open addressing)*
  - Kapacita přihrádky je 1. Pokud je přihrádka $m_0 = \varphi(x)$ obsazená, zkusíme jinou (m_1, m_2, …)
  - Lineární zkoušení *(linear probing)* — zkusíme $m_i = m_0 + i$.
  - Kvadratické zkoušení *(quadratic probing)* — zkusíme $m_i = m_0 + ai^2 + bi$, např. $a = 1$, $b = 0$.
  - Dvojité rozptylování *(double hashing)* — zkusíme $m_i = m_0 + i\psi(x)$.

Řešení kolizí

Co když dvě položky mají stejný otisk?

- **Zřetězení** *(chaining)*
  - Každá přihrádka je seznam *(nebo pole)*.
  - Zaplnění λ může být > 1.
- **Otevřené adresování** *(open addressing)*
  - Kapacita přihrádky je 1. Pokud je přihrádka $m_0 = \varphi(x)$ obsazená, zkusíme jinou (m_1, m_2, …)
  - Lineární zkoušení *(linear probing)* — zkusíme $m_i = m_0 + i$.
  - Kvadratické zkoušení *(quadratic probing)* — zkusíme $m_i = m_0 + ai^2 + bi$, např. $a = 1$, $b = 0$.
  - Dvojité rozptylování *(double hashing)* — zkusíme $m_i = m_0 + i\psi(x)$.
  - Menší režie než zřetězení.
  - Zaplnění λ nesmí být velké (≈ 0.7).
  - Rozptylovací funkce nesmí vytvářet shluky.

Počet porovnání při hledání

	úspěšné	neúspěšné
zřetězení	$1 + \frac{\lambda}{2}$	λ
otevřené adresování	$\frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$	$\frac{1}{2}\left(1 + \left(\frac{1}{1-\lambda}\right)^2\right)$

Počet přístupů do paměti je větší o 1 + režie přihrádek (např. 2 přístupy na porovnání u spojového seznamu).

Otevřené adresování — příklad

$m = 11$ přihrádek, rozptylovací funkce $\varphi(x) = x \bmod m$

Vložíme čísla

x	54	26	93	17	77	31	44	55	20
$\varphi(x)$	10	4	5	6	0	9	0	0	9

Po vložení 31:

0	1	2	3	4	5	6	7	8	9	10
77				26	93	17			31	54

Otevřené adresování — příklad

$m = 11$ přihrádek, rozptylovací funkce $\varphi(x) = x \bmod m$

Vložíme čísla

x	54	26	93	17	77	31	44	55	20
$\varphi(x)$	10	4	5	6	0	9	0	0	9

Po vložení 44:

0	1	2	3	4	5	6	7	8	9	10
77	44			26	93	17			31	54

Otevřené adresování — příklad

$m = 11$ přihrádek, rozptylovací funkce $\varphi(x) = x \bmod m$

Vložíme čísla

x	54	26	93	17	77	31	44	55	20
$\varphi(x)$	10	4	5	6	0	9	0	0	9

Po vložení 55:

0	1	2	3	4	5	6	7	8	9	10
77	44	55		26	93	17			31	54

Otevřené adresování — příklad

$m = 11$ přihrádek, rozptylovací funkce $\varphi(x) = x \bmod m$

Vložíme čísla

x	54	26	93	17	77	31	44	55	20
$\varphi(x)$	10	4	5	6	0	9	0	0	9

Po vložení 20:

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17			31	54

Otevřené adresování — příklad

$m = 11$ přihrádek, rozptylovací funkce $\varphi(x) = x \bmod m$

Vložíme čísla

x	54	26	93	17	77	31	44	55	20
$\varphi(x)$	10	4	5	6	0	9	0	0	9

Po vložení 20:

0	1	2	3	4	5	6	7	8	9	10
77	44	55	20	26	93	17			31	54

'Prázdné' položky = speciální hodnota.

Implementace např. *Problem Solving with Algorithms and Data Structures*

https://interactivepython.org/runestone/static/pythonds/SortSearch/Hashing.html

Mazání položek

- Zřetězení — smažeme ze seznamu přihrádky.

Mazání položek

- Zřetězení — smažeme ze seznamu přihrádky.
- Otevřené adresování — smazané položky označíme speciální hodnotou ‘přeskoč’.
- Mazání často není potřeba

Implementace rozptylové tabulky

Asociativní mapa, kolizní strategie zřetězení. Podobné rozhraní jako `BinarySearchTree` a `dict`:

- `h=Hashtable(n)` — vytvoření
- `h=put(h,key,value)` — vložení položky
- `get(h,key)` → `value` — nalezení/vyzvednutí hodnoty
- `items(h)` — seznam dvojic (klíč,hodnota)

```
class Hashtable:

  def __init__(self,n=13):  # 'n' je doporučená velikost
    self.size=find_prime_size(n)
    self.keys  =[ [] for i in range(self.size) ]
    self.values=[ [] for i in range(self.size) ]
    self.count=0
```

Soubor `hashing.py`.

Nalezení položky

```
def get(h,key):
  """ Vrátí 'value' prvku s klíčem 'key', jinak None """
  m=hash(key) % h.size      # číslo přihrádky
  i=find_index(h.keys[m],key) # je tam?
  if i is None:             # není
    return None
  return  h.values[m][i]

def find_index(l,x):
  """ Vrátí index 'i' aby l[i]==x nebo 'None'
      pokud 'x' není v 'l' """
  for i,v in enumerate(l): # dvojice index, hodnota
    if v==x:
      return i
  return None
```

V pythonu existuje metoda pole `index`, používá výjimky.

Soubor `hashing.py`.

Vložení položky

```
def put(h,key,value):
    """ Vloží pár 'key'->'value' do tabulky
        a vrátí odkaz na aktualizovanou """
    m=hash(key) % h.size     # číslo přihrádky
    i=find_index(h.keys[m],key)    # je tam?
    if i is not None:              # klíč v tabulce už je
      h.values[m][i]=value
      return h
    h.keys[m].append(key)          # klíč v tabulce není
    h.values[m].append(value)
    h.count+=1
    if h.count>h.size*0.75:        # je tabulka moc plná?
      return grow_table(h)
    return h
```

Zvětšení tabulky

```
def grow_table(h):
  """ Vytvoří větší tabulku, překopíruje tam obsah
      a vrátí ji """
  hnew=Hashtable(2*h.size)
  for i in range(h.size):     # okopíruj vše do hnew
    keys=h.keys[i]
    values=h.values[i]
    for j in range(len(keys)):
      put(hnew,keys[j],values[j])
  return hnew
```

Získání obsahu tabulky

```
def items(h):
  """ Vrátí seznam dvojic klíč,hodnota """
  r=[]
  for i in range(h.size):
    r+=zip(h.keys[i],h.values[i])
  return list(r)
```

Získání obsahu tabulky

```
def items(h):
  """ Vrátí seznam dvojic klíč,hodnota """
  r=[]
  for i in range(h.size):
    r+=zip(h.keys[i],h.values[i])
  return list(r)
```

- Další možná rozhraní — `iter`, `reduce`, iterátor...
- `list` dělá z posloupnosti *(lazy/on-demand)* seznam — volíme dle aplikace

Příklad

```
from hashing import *

t=Hashtable()
t=put(t,'pi',    3.14159)
t=put(t,'e',     2.71828)
t=put(t,'sqrt2',1.41421)
t=put(t,'golden',1.61803)
print(get(t,'pi'))
```

```
3.14159
```

```
print(get(t,'e'))
```

```
2.71828
```

```
print(get(t,'gamma'))
```

```
None
```

Příklad: Počítání frekvence slov

Zjistěte relativní frekvence slov v daném textu (souboru)

- Načtení souboru, rozdělení na slova.
- Spočítání frekvence slov
- Seřazení a vytisknutí

```
def word_frequencies(filename):
  w=read_words(filename)       # seznam slov
  c=word_counts_dictionary(w) # seznam dvojic (slovo,počet)
  print_frequencies(c)
```

Soubor `word_frequencies.py`.

Načtení slov

```
word_pattern=re.compile(r'[A-Za-z]+')

def read_words(filename):
   words=[]
   with open(filename,'rt') as f: # otevři textový soubor
     for line in f.readlines():   # čti řádku po řádce
       line_words=word_pattern.findall(line)
       line_words=map(lambda x: x.lower(),line_words)
       words+=line_words
   return words
```

Spočítání slov (1)

dict

Asociativní mapa count uchovává počet výskytů, klíčem je slovo.

```
def word_counts_dictionary(words):
    """ Vrátí seznam dvojic slov a jejich frekvencí """
    counts={} # slovník
    for w in words:
      if w in counts:
        counts[w]+=1
      else:
        counts[w]=1
    return list(counts.items())
```

Spočítání slov (2)

Rozptylovací tabulka

```python
import hashing

def word_counts_hashtable(words):
    """ Vrátí seznam dvojic slov a jejich frekvencí """
    counts=hashing.Hashtable()
    for w in words:
      value=hashing.get(counts,w)
      if value is None:
        counts=hashing.put(counts,w,1)
      else:
        counts=hashing.put(counts,w,value+1)
    return hashing.items(counts)
```

Spočítání slov (3)

Vyhledávací strom

```
import binary_search_tree as bst

# implementace pomocí vyhledávacího stromu
def word_counts_bst(words):
    """ Vrátí seznam dvojic slov a jejich frekvencí """
    counts=None
    for w in words:
      value=bst.get(counts,w)
      if value is None:
        counts=bst.put(counts,w,1)
      else:
        counts=bst.put(counts,w,value+1)
    return bst.items(counts)
```

Setřídění a tisk

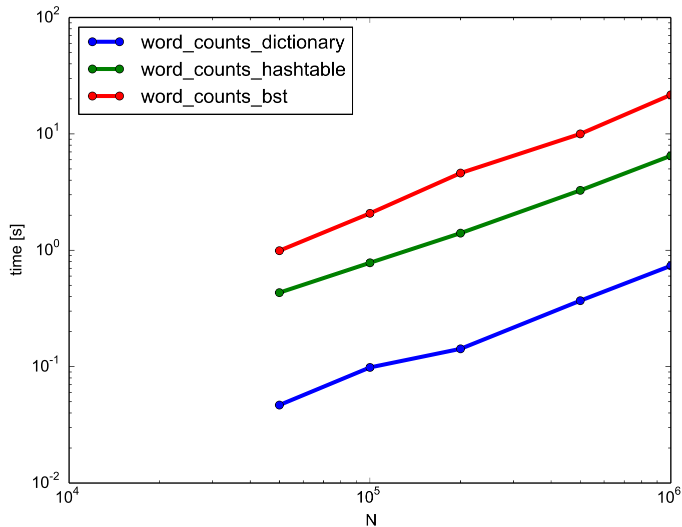
```
def print_frequencies(counts,n=10):
    """ Vytiskne 'n' nečastěji použitých slov dle
        seznamu dvojic (slovo,frekvence) 'counts' """
    # setřiď od nejčastějšího
    counts.sort(key=lambda x: x[1],reverse=True)
    # celkový počet slov
    nwords=functools.reduce(lambda acc,x: x[1]+acc,counts,0)
    for i in range(min(n,len(counts))):
      print("%10s %6.3f%%" %
             (counts[i][0],counts[i][1]/nwords*100.))
```

Frekvence slov — příklad

```
Terminal> python3 word_frequencies.py poe.txt
```

```
 the  7.653%
  of  4.589%
 and  2.605%
  to  2.484%
   a  2.282%
  in  2.096%
   i  1.371%
  it  1.233%
that  1.121%
 was  1.103%
  is  0.912%
with  0.844%
  at  0.812%
  as  0.761%
this  0.732%
```

Porovnání rychlosti
Četnost slov



Rozptylovací tabulky — shrnutí

- Implementace asociativní mapy nebo množiny.
- Velmi rychlé operace vkládání a vyhledávání (v průměru $O(1)$, nejhorší případ $O(n)$).
- Citlivé na volbu rozptylovací funkce a velikost tabulky.
- Potřebuje rozptylovací funkci a test na rovnost.
- Nepotřebuje/neumí porovnávat velikost.

Náměty na domácí práci

- Implementujte otevřenou adresaci.
- Najděte závislost času na zaplnění tabulky.
- Implementujte mazání.
- Implementujte zcela stejné rozhraní jako `dict`
- Implementujte počítání frekvence slov bez asociativní mapy, například na základě třídění. Porovnejte efektivitu.