9. Datové struktury: strom a rozptylovací tabulka

BAB37ZPR – Základy programování

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Strom
  - Binární vyhledávací stromy
  - Množiny a mapy
- Část 2 – Rozptylovací tabulka

Část I

Strom

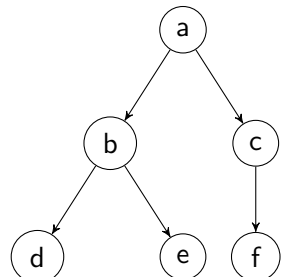
Strom

Strom

- skládá se s *uzlů (nodes)* spojených *hranami (edges)*.
- je souvislý a acyklický

Kořenový strom

- orientovaný graf, má jeden význačný uzel = *kořen (root)*
- z kořene vede do každého jiného uzlu právě jedna orientovaná cesta
- do kořene nevstupuje žádná hrana, do každého jiného uzlu vstupuje právě jedna hrana



Vlastnosti stromů

- každé dva uzly jsou spojeny právě jednou neorientovanou cestou
- počet hran = počet uzlů - 1
- pokud jednu hranu vyjmeme, graf bude nesouvislý
- pokud jednu hranu přidáme, graf bude obsahovat cyklus (kružnici)

Názvosloví

- kořen, list, vnitřní uzel, rodič, (pravý/levý) potomek (syn), sourozenci, stupeň uzlu, hloubka (výška)

Poziční strom

- potomci jsou označeni čísly (=levý/pravý)
- některý potomek může chybět

Binární strom

- poziční strom
- každý uzel má nanejvýš dva potomky.

Úplný binární strom s n uzly

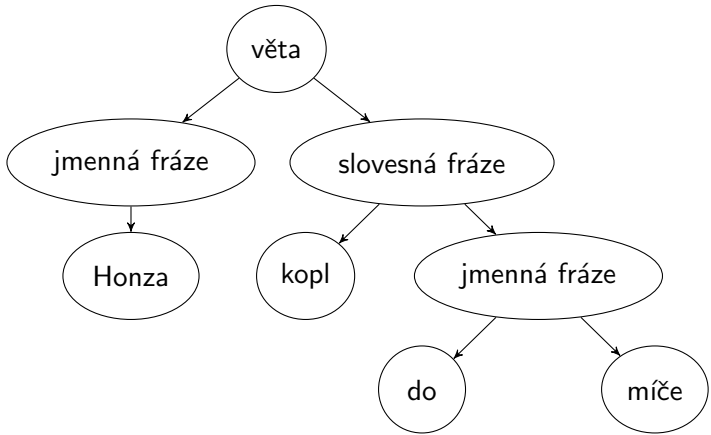
- každý uzel kromě listů má právě dva potomky
- Počet uzlů v hloubce i je 2^i
- $n = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$
- Všechny listy mají hloubku $h = \log_2(n+1) - 1$
- Počet listů je $(n+1)/2$, počet vnitřních uzlů je $(n-1)/2$.

Pro každý binární strom s n uzly a hloubkou h

$$\log_2(n+1) - 1 \leq h \leq n - 1$$

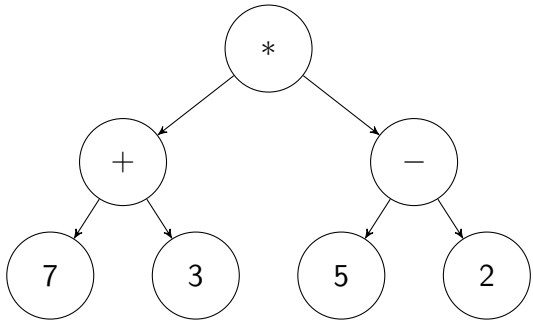
Příklady stromů 1/2

Gramatická struktura věty



Příklady stromů 2/2

Struktura aritmetického výrazu $(7 + 3) * (5 - 2)$



Reprezentace stromu – záznam

```
>>> class BinaryTree:
...   def __init__(self, data, left=None, right=None):
...     self.data = data
...     self.left = left
...     self.right = right
...
```

Reprezentace výrazu $(7 + 3) * (5 - 2)$:

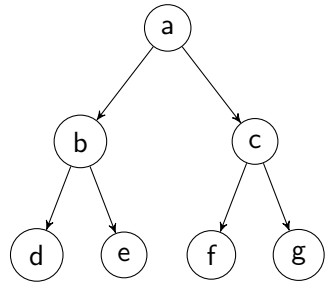
```
>>> t1 = BinaryTree('*',
...         BinaryTree('+', BinaryTree(7), BinaryTree(3)),
...         BinaryTree('-', BinaryTree(5), BinaryTree(2)))
```

V této reprezentaci *strom=kořen*. Prázdný strom = `None`.

binary_tree.py

Procházení stromu

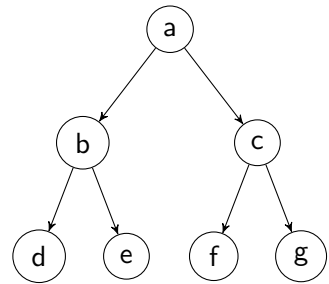
```
>>> t2 = BinaryTree('a',
...          BinaryTree('b',
...              BinaryTree('d'),
...              BinaryTree('e')),
...          BinaryTree('c',
...              BinaryTree('f'),
...              BinaryTree('g')))
```



Procházení stromu – preorder

- nejdřív aktuální uzel, pak oba podstromy
- prefixová notace
- abdecfg

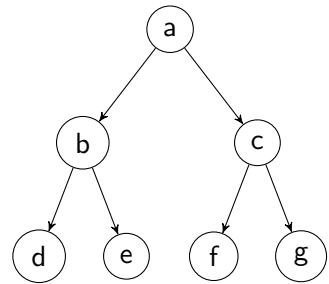
```
>>> def to_string_preorder(tree):
...     return (str(tree.data) + " " +
...         to_string_preorder(tree.left) +
...         to_string_preorder(tree.right)
...         if tree else "")
...
>>> print(to_string_preorder(t2))
a b d e c f g
```



Procházení stromu – postorder

- nejdřív oba podstromy, pak aktuální uzel
- postfixová notace
- debfgca

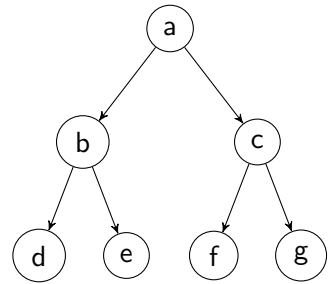
```
>>> def to_string_postorder(tree):
...   return (to_string_postorder(tree.left) +
...     to_string_postorder(tree.right) + " " +
...     str(tree.data)
...     if tree else "")
...
>>> print(to_string_postorder(t2))
 d e b f g c a
```



Procházení stromu – inorder

- levý podstrom, pak aktuální uzel, pak pravý podstrom
- infixová notace
- dbeafcg

```
>>> def to_string_inorder(tree):
...     if not tree:          # prázdný strom
...         return ""
...     if tree.left:  # binární operátor
...         return ("(" + to_string_inorder(tree.left)
...           + str(tree.data)
...           +  to_string_inorder(tree.right) + ")" )
...     return str(tree.data) # jen jedno číslo
...
>>> print(to_string_inorder(t2))
((dbe)a(fcg))
```



Vyhodnocení výrazu

```
>>> def evaluate(tree):
...   """ Vyhodnoti aritmeticky vyraz zadany stromem """
...   if tree.data=='+':
...     return evaluate(tree.left) + evaluate(tree.right)
...   if tree.data=='-':
...     return evaluate(tree.left) - evaluate(tree.right)
...   if tree.data=='*':
...     return evaluate(tree.left) * evaluate(tree.right)
...   if tree.data=='/':
...     return evaluate(tree.left) / evaluate(tree.right)
...   return tree.data # jen jedno číslo
...
>>> # t1 ~ (7+3)*(5-2)
>>> print(evaluate(t1))
30
```

I. Strom

Binární vyhledávací stromy

Množiny a mapy

Binární vyhledávací stromy – motivace

Aktualizovatelná datová struktura pro rychlé vyhledávání *porovnatelných* dat.

- *Setříděné pole* – vkládání $O(n)$, vyhledávání $O(\log n)$
- *Spojový seznam* – vkládání $O(1)$, vyhledávání $O(n)$
- *Vyhledávací strom* – vkládání $O(\log n)$, vyhledávání $O(\log n)$

Množina

Podporované operace

- `add(key)` – vložení prvku
- `delete(key)` – odstranění prvku
- `contains(key)` – obsahuje množina daný prvek?

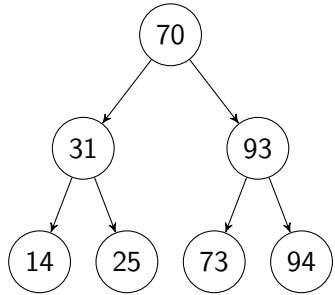
Pomocné funkce: `size` / `len`

Rychlé operace (složitost $O(\log n)$ nebo lepší)

Binární vyhledávací strom

Vlastnosti

- každý uzel obsahuje *klíč*
- klíč v uzlu není menší, než všechny klíče v jeho levém podstromu
- klíč v uzlu není větší, než všechny klíče v jeho pravém podstromu



Reprezentace vyhledávacího stromu

```
>>> class BinarySearchTree:
...     def __init__ (self, key, left=None, right=None):
...         self.key = key
...         self.left = left
...         self.right = right
...
```

- Strom = uzel.
- Prázdný strom reprezentujeme jako `None`.

binary_search_tree.py

Vyhledávání ve stromu

```
>>> def contains(tree, key):
...     """ Je prvek 'key' ve stromu? """
...     if tree:                    # je strom neprázdný?
...         if tree.key == key:     # je to hledaný klíč?
...             return True
...         if tree.key > key:
...             return contains(tree.left, key)
...         else:
...             return contains(tree.right, key)
...     return False
...
```

Vytvoření stromu

```
>>> def from_array(a):
...     """ Sestrojíme strom ze setříděného pole. """
...     def build(a):
...         if len(a) == 0:
...             return None
...         if len(a) == 1:
...             return BinarySearchTree(a[0])
...         m = len(a)//2
...         return BinarySearchTree(a[m],
...                 left = build(a[:m]),
...                 right = build(a[m+1:]))
...     a = sorted(a)
...     return build(a)
...
```

Vytisknutí stromu

```
>>> def print_tree(tree, level=0, prefix=""):
...     if tree:
...         print(" " * (4*level) + prefix + str(tree.key))
...         if tree.left:
...             print_tree(tree.left, level=level+1, prefix = "L:")
...         if tree.right:
...             print_tree(tree.right, level=level+1,prefix = "R:")
...
```

Vyhledávácí strom – příklad

```
>>> t = from_array([21, 16, 19, 87, 34, 92, 66])
>>> print_tree(t)
34
    L:19
        L:16
        R:21
    R:87
        L:66
        R:92
>>> print(contains(t,30))
False
>>> print(contains(t,66))
True
```

Vkládání do stromu

```
>>> def add(tree, key):
...     """ Vlozi 'key' do stromu a vrati novy koren """
...     if tree is None:
...         return BinarySearchTree(key)
...     if key < tree.key:
...         tree.left=add(tree.left, key)
...     elif key > tree.key:
...         tree.right=add(tree.right, key)
...     return tree # hodnota jiz ve stromu je
...
```

Vkládání do stromu — příklad

```
>>> add(t, 30)
<__console__.BinarySearchTree object at 0x000001D33E4F0E80>
>>> print_tree(t)
34
    L:19
        L:16
        R:21
            R:30
    R:87
        L:66
        R:92
```

Příklad: strom jako množina

Vypiš všechny možné součty hodů na dvou kostkách.

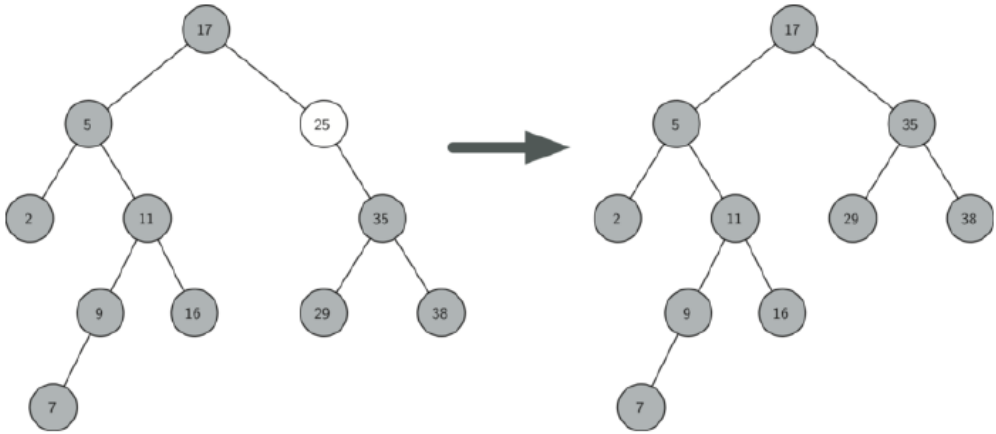
```
>>> import random
>>> s = None
>>> for i in range(1000):
...     s = add(s, random.randrange(1,7) + random.randrange(1,7))
...
>>> print_tree(s)
8
    L:5
        L:4
            L:3
                L:2
        R:6
            R:7
    R:11
        L:10
```

Převod na pole

```
>>> def to_array(tree):
...     ''' Projde uzly stromu podle velikosti a uloží do pole '''
...     a=[]
...     def insert_inorder(t):
...         nonlocal a
...         if t:
...             insert_inorder(t.left)
...             a+=[t.key]
...             insert_inorder(t.right)
...     insert_inorder(tree)
...     return a
...
>>> print(to_array(s))
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

`nonlocal` — přístup k proměnné vnější funkce *(jen Python 3)*

Odstranění prvku ze stromu 1/4



Odstranění prvku – implementace 2/4

```
>>> def delete(tree, key):
...   """ Smaze 'key' za stromu 'tree' a vrati novy koren. """
...   if tree is not None:
...     if key < tree.key: # najdi uzel 'key'
...       tree.left = delete(tree.left, key)
...     elif key > tree.key:
...       tree.right = delete(tree.right, key)
...     else: # uzel nalezen, má syny?
...       if tree.left is None:
...         return tree.right # jen pravý syn nebo nic
...       elif tree.right is None:
...         return tree.left # jen levý syn nebo nic
```

Odstranění prvku – implementace 3/4

```
...         else: # nahradíme uzel maximem levého podstromu
...           w = rightmost_node(tree.left)
...           tree.key = w.key
...           tree.left = delete(tree.left, w.key)
...     return tree
...
>>> def rightmost_node(tree):
...     while tree.right:
...       tree=tree.right
...     return tree
...
```

Odstranění prvku — příklad 4/4

```
>>> t=from_array([21, 16, 19, 87, 34, 92, 66])
```

```
>>> print_tree(t)
34
    L:19
        L:16
        R:21
    R:87
        L:66
        R:92
```

```
>>> t=delete(t,87)
>>> print_tree(t)
34
    L:19
        L:16
        R:21
    R:66
        R:92
```

I. Strom

Binární vyhledávací stromy

Množiny a mapy

Množina

Podporované operace

- `add(key)` – vložení prvku
- `delete(key)` – odstranění prvku
- `contains(key)` – obsahuje množina daný prvek?

Pomocné funkce: `size` / `len`

Rychlé operace (složitost $O(\log n)$ nebo lepší)

Asociativní mapa

Funkce `klíč` → `hodnota` (`key` → `value`)

Podporované operace

- `put(key,value)` – vložení položky
- `delete(key)` – odstranění prvku
- `contains(key)` – obsahuje mapa daný prvek?
- `get(key)` → `value` – nalezení/vyzvednutí hodnoty

Pomocné funkce: `size` / `len`

Rychlé operace (složitost $O(\log n)$ nebo lepší)

Množina je speciální případ mapy.

Reprezentace

```
>>> class BinarySearchTree:
...   def __init__(self, key,value=None,left=None,right=None):
...     self.key = key
...     self.value = value
...     self.left = left
...     self.right = right
...
```

Tisk mapy

```
>>> def print_tree(tree, level=0, prefix=""):
...     if tree:
...         if tree.value:
...             print(" "*(4*level) + prefix + str(tree.key) +
...             " -> " + str(tree.value))
...         else:
...             print(" " * (4*level) + prefix + str(tree.key))
...         if tree.left:
...             print_tree(tree.left, level=level+1, prefix = "L:")
...         if tree.right:
...             print_tree(tree.right, level=level+1,prefix = "R:")
...
```

Vyhledávání v mapě

```
>>> def get(tree,key):
...     """ Vrati 'value' prvku s klicem 'key', jinak None """
...     if tree:              # je strom neprazdny?
...       if tree.key==key: # je to hledany klic?
...         return tree.value
...       if tree.key>key:
...         return get(tree.left,key)
...       else:
...         return get(tree.right,key)
...     return None
...
```

Vkládání do mapy

```
>>> def put(tree,key,value):
...   """ Vlozi par 'key'->'value', vrati novy koren """
...   if tree is None:
...     return BinarySearchTree(key,value=value)
...   if key<tree.key:
...     tree.left=put(tree.left,key,value)
...   elif  key>tree.key:
...     tree.right=put(tree.right,key,value)
...   else:
...     tree.value=value # klíč již ve stromu je
...   return tree
...
```

Mapa – příklad – tabulka symbolů

```
>>> t = None
>>> 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_tree(t)
pi -> 3.14159
    L:e -> 2.71828
        R:golden -> 1.61803
    R:sqrt2 -> 1.41421
```

```
>>> print(get(t,'pi'))
3.14159
>>> print(get(t,'e'))
2.71828
>>> print(get(t,'gamma'))
None
```

Implementace funguje i pro řetězcové klíče.

Vyhledávací stromy

- Datová struktura pro porovnatelné klíče
- Může reprezentovat množinu i mapu.
- Základní operace (vkládání, hledání, mazání) mají složitost $O(\log n)$.
- Vyšší režie (oproti např. poli)
- Stromů je mnoho typů
  - B-stromy
  - k-d stromy, R-stromy
  - prefixové stromy
  - *ropes*...

Část II

Rozptylovací tabulka

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

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 $0 \ldots m-1$
- 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

Průměrný počet položek na přihrádku

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

- velké $\lambda \rightarrow$ hodně kolizí $\rightarrow$ zpomalení operací
- malé $\lambda \rightarrow$ 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 a určíme indexy

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 $P\big[\varphi(x) \neq \varphi(y)\big]$
  - 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**.

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

hashing.py

Rozptylovací funkce v Pythonu

Funkce hash vrací (velké) celé číslo

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

```
>>> print(hash(34))
34
>>> print(hash("les"))
-6203894071973765024
>>> print(hash((7, "pes")))
-2309871195336679876
```

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`

Rozptylovací funkce v Pythonu

Funkce hash vrací (velké) celé číslo

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

```
>>> print(hash(34))
34
>>> print(hash("les"))
-6203894071973765024
>>> print(hash((7, "pes")))
-2309871195336679876
```

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`

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

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

Řešení kolizí

Co když dvě položky mají stejný otisk?

Zřetězení

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

Otevřené adresování

- 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.

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.
- 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 = n
...     self.keys   = [ [] for i in range(self.size) ]
...     self.values = [ [] for i in range(self.size) ]
...     self.count = 0
...
```

hashing.py.

Nalezení položky

```
>>> def get(h, key):
...   """ Vrati 'value' pruku s klicem 'key', jinak None """
...   m=hash(key) % h.size         # cislo prihradky
...   i=find_index(h.keys[m],key) # je tam?
...   if i is None:               # neni
...     return None
...   return h.values[m][i]
...
>>> def find_index(l, x):
...   """ Vrati index 'i' aby l[i]==x nebo 'None' pokud 'x' neni 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.

Vložení položky

```
>>> def put(h,key,value):
...     """ Vlozi par 'key'->'value' do tabulky
...         a vrati odkaz na aktualizovanou """
...     m=hash(key) % h.size      # číslo přihrádky
...     i=find_index(h.keys[m],key)    # je tam?
...     if i is not None:              # klic v tabulce uz je
...       h.values[m][i]=value
...       return h
...     h.keys[m].append(key)          # klic v tabulce neni
...     h.values[m].append(value)
...     h.count+=1
...     if h.count>h.size*0.75:        # je tabulka moc plna?
...       return grow_table(h)
...     return h
...
```

Zvětšení tabulky

```
>>> def grow_table(h):
...     """ Vytvori vetsi tabulku, prekopiruje tam obsah a vrati ji """
...     hnew = Hashtable(2*h.size)
...     for i in range(h.size):      # okopiruj vse 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):
...     """ Vrati seznam dvojic klic, 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

```
>>> 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

```
1  def word_frequencies(filename):
2    w = read_words(filename)       # seznam slov
3    c = word_counts_dictionary(w) # seznam dvojic (slovo, pocet)
4    print_frequencies(c)
```

word_frequencies.py.

Načtení slov

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

3  def read_words(filename):

5      words=[]

7      with open(filename,'rt') as f: # otevri textovy soubor
8        for line in f.readlines():   # cti radku po radce
9          line_words=word_pattern.findall(line)
10         line_words=map(lambda x: x.lower(),line_words)
11         words+=line_words

13     return words
```

Spočítání slov (1) – dict

Asociativní mapa `count` uchovává počet výskytů, klíčem je slovo.

```
1  def word_counts_dictionary(words):
2      """ Vrati seznam dvojic slov a jejich frekvenci """
3
4      counts={} # slovník
5
6      for w in words:
7        if w in counts:
8          counts[w]+=1
9        else:
10         counts[w]=1
11
12     return list(counts.items())
```

Spočítání slov (2) – Rozptylovací tabulka

```
1  import hashing
2
3  def word_counts_hashtable(words):
4      """ Vrati seznam dvojic slov a jejich frekvenci """
5      counts=hashing.Hashtable()
6      for w in words:
7        value=hashing.get(counts,w)
8        if value is None:
9          counts=hashing.put(counts,w,1)
10       else:
11         counts=hashing.put(counts,w,value+1)
12     return hashing.items(counts)
```

Spočítání slov (3) – vyhledávací strom

```
1  import binary_search_tree as bst
2
3  # implementace pomoci vyhledavaciho stromu
4  def word_counts_bst(words):
5      """ Vrati seznam dvojic slov a jejich frekvenci """
6
7      counts=None
8
9      for w in words:
10       value=bst.get(counts,w)
11       if value is None:
12         counts=bst.put(counts,w,1)
13       else:
14         counts=bst.put(counts,w,value+1)
15
16     return bst.items(counts)
```

Setřídění a tisk

```
1  def print_frequencies(counts, n=10):
2      """ Vytiskne 'n' nejcasteji pouzitych slov dle
3          seznamu dvojic (slovo,frekvence) 'counts' """
4
5      # setrid od nejcastejsiho
6      counts.sort(key=lambda x: x[1],reverse=True)
7
8      # celkovy pocet slov
9      nwords=functools.reduce(lambda acc,x: x[1]+acc,counts,0)
10
11     for i in range(min(n,len(counts))):
12     print("%10s %6.3f%%" %
13               (counts[i][0],counts[i][1]/nwords*100.))
```

Frekvence slov – příklad

```
$ 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%
```

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.