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 tabuklka

Část I

Strom

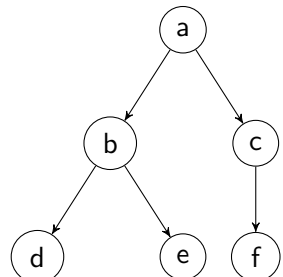
Strom

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.

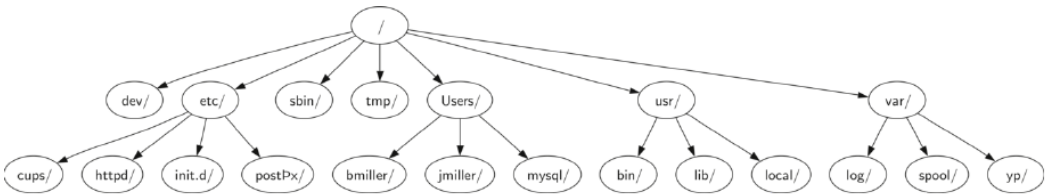
Úplný binární strom s n uzly

- 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

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

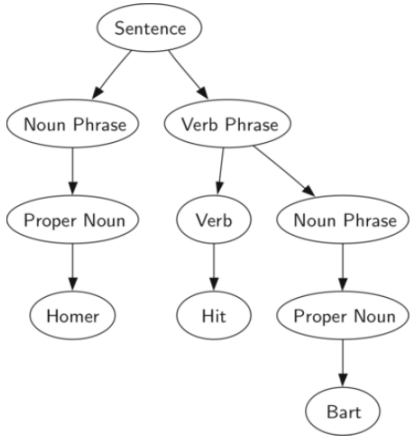
Příklady stromů



Unixová struktura adresářů

Images courtesy of Brad Miller, David Ranum.

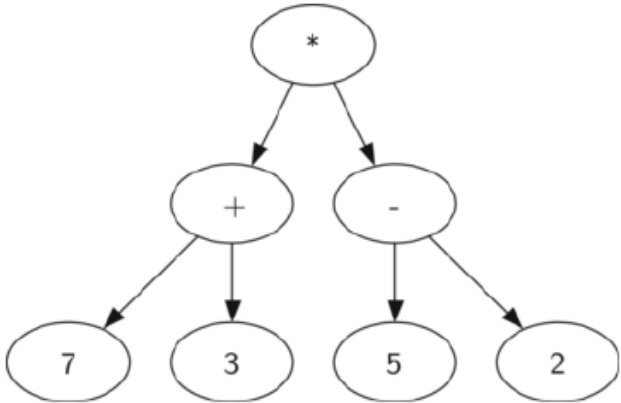
Příklady stromů



Gramatická struktura věty.

Images courtesy of Brad Miller, David Ranum.

Příklady stromů



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

Images courtesy of Brad Miller, David Ranum.

Reprezentace stromu – záznam

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

Reprezentace výrazu $(7+3)*(5-2)$:

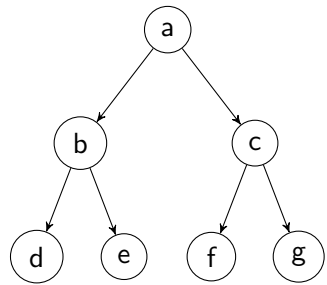
```
t=BinaryTree('*',
    BinaryTree('+',BinaryTree(7),BinaryTree(3)),
    BinaryTree('-',BinaryTree(5),BinaryTree(2)))
```

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

binary_tree.py

Procházení stromu

- *preorder*
  - nejdřív aktuální uzel, pak oba podstromy
  - prefixová notace
  - `abdecfg`
- *inorder*
  - levý podstrom, pak aktuální uzel, pak pravý podstrom
  - infixová notace
  - `dbeafcg`
- *postorder*
  - nejdřív oba podstromy, pak aktuální uzel
  - postfixová notace
  - `debfgca`



Procházení stromu – implementace 1/2

```
1  def to_string_preorder(tree):
2    return (str(tree.data) + " " +
3        to_string_preorder(tree.left) +
4      to_string_preorder(tree.right)
5      if tree else " ")

print(to_string_preorder(t))
```

```
1  def to_string_postorder(tree):
2    return ( to_string_postorder(tree.left) +
3             to_string_postorder(tree.right) + " " +
4             str(tree.data)
5             if tree else "" )

print(to_string_postorder(t))
```

Procházení stromu – implementace 2/2

```
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(t))
```

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


print(evaluate(t))
```

binary_tree.py

I. Strom

Binární vyhledávací stromy

Množiny a mapy

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?

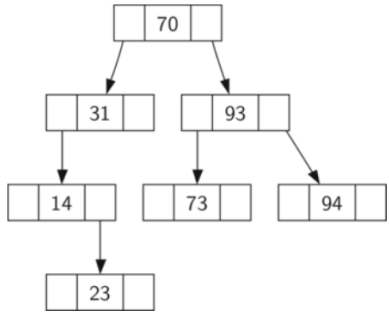
Pomocné funkce: `size` / `len`

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

```
1  class BinarySearchTree:
2    def __init__ (self, key, left=None, right=None):
3        self.key = key
4        self.left = left
5        self.right = right
```

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

binary_search_tree.py

Vyhledávání ve stromu

```
1  def contains(tree,key):
2    """ Je prvek 'key' ve stromu? """
4      if tree:              # je strom neprázdný?
5        if tree.key==key: # je to hledaný klíč?
6            return True
7          if tree.key>key:
8            return contains(tree.left,key)
9          else:
10           return contains(tree.right,key)
12     return False
```

Vytvoření stromu

Hledání ve stromu je ekvivalentní binárnímu vyhledávání.
Sestrojíme strom ze setříděného pole.

```
1  def from_array(a):
2      """ Build a tree (containing only keys) from an array """
3
4      def build(a):
5        if len(a) == 0:
6            return None
7        if len(a) == 1:
8            return BinarySearchTree(a[0])
9        m = len(a)//2
11       return BinarySearchTree(a[m], left=build(a[:m]),
12                               right=build(a[m+1:]))
13   a = sorted(a)
14   return build(a)
```

Vytisknutí stromu

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

Vyhledávácí strom – příklad

```
1  import binary_search_tree as bst
2  t = bst.from_array([21, 16, 19, 87, 34, 92, 66])
3  bst.print_tree(t)
```

```
print(contains(t,30))
```

```
False
```

```
print(contains(t,66))
```

```
True
```

Vkládání do stromu

```
1  def add(tree,key):
2      """ Vlozi 'key' do stromu a vrati novy koren """
3
4      if tree is None:
5        return BinarySearchTree(key)
6      if key < tree.key:
7        tree.left=add(tree.left,key)
8      elif key > tree.key:
9        tree.right=add(tree.right,key)
10
11   return tree # hodnota jiz ve stromu je
```

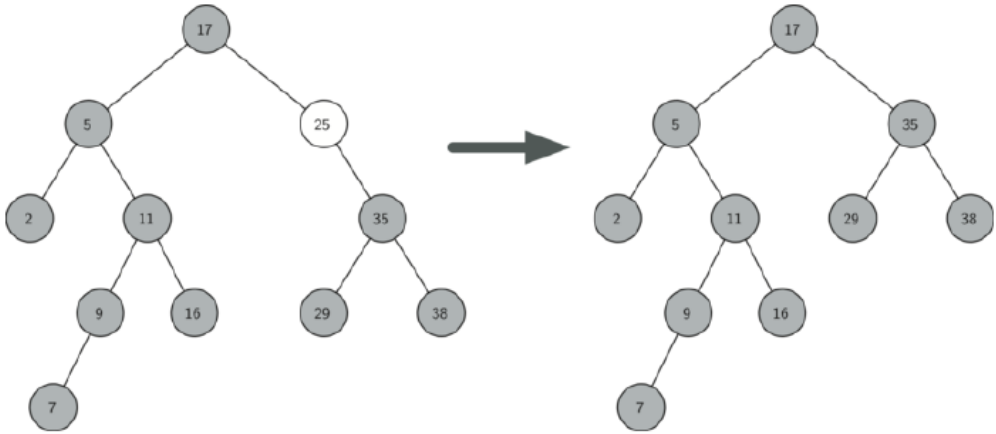
Převod na pole

Projde uzly stromu podle velikosti a uloží do pole.

```
1  def to_array(tree):
2    a=[]
3    def insert_inorder(t):
4      nonlocal a
5      if t:
6        insert_inorder(t.left)
7        a+=[t.key]
8        insert_inorder(t.right)
9    insert_inorder(tree)
10   return a
12 print(to_array(s))
```

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

Odstranění prvku ze stromu



Odstranění prvku — implementace

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

Odstranění prvku (2)

```
def rightmost_node(tree):
  while tree.right:
    tree=tree.right
  return tree
```

I. Strom

Binární vyhledávací stromy

Množiny a mapy

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

```
1  class BinarySearchTree:
2    def __init__(self, key,value=None,left=None,right=None):
3      self.key = key
4      self.value = value
5      self.left = left
6      self.right = right
```

binary_search_tree.py

Vyhledávání v mapě

```
1  def get(tree,key):
2      """ Vrati 'value' prvku s klicem 'key', jinak None """
3      if tree:              # je strom neprazdny?
4        if tree.key==key: # je to hledany klic?
5          return tree.value
6        if tree.key>key:
7          return get(tree.left,key)
8        else:
9          return get(tree.right,key)
10     return None
```

Vkládání do mapy

```
1  def put(tree,key,value):
2    """ Vlozi par 'key'->'value', vrati novy koren """
3    if tree is None:
4      return BinarySearchTree(key,value=value)
5    if key<tree.key:
6      tree.left=put(tree.left,key,value)
7    elif  key>tree.key:
8      tree.right=put(tree.right,key,value)
9    else:
10     tree.value=value # klíč již ve stromu je
11   return tree
```

Mapa – příklad – tabulka symbolů

```
1  from binary_search_tree import *
3  t=None
4  t=put(t,'pi',     3.14159)
5  t=put(t,'e',      2.71828)
6  t=put(t,'sqrt2',  1.41421)
7  t=put(t,'golden',1.61803)
8  print_tree(t)
```

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

print(get(t,'e'))

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

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*...

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

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**.

```
1  def hash_string(x,m):
2    h=0
3    for c in x:
4      h=((h*67)+ord(c)) % m
5    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"))
```

```
-1710257127033717405
```

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

```
-2038261560997172772
```

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.

```
1  primes=prvocisla_eratosthenes(100000)
3  def find_prime_size(n):
4    for i in range(len(primes)):
5      if primes[i]>n:
6        return n
7    print("Pozor, tabulka prvocisel je prilis kratka.")
8    return n
```

Zrychlování

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

hashing.py

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.

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.

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.

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.
- Otevřené adresování — smazané položky označíme speciální hodnotou ‘přeskoč’.
- Mazání často není potřeba

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)

```
1  class Hashtable:
3    def __init__(self,n=13):  # 'n' je doporučená velikost
4      self.size = find_prime_size(n)
5      self.keys   = [ [] for i in range(self.size) ]
6      self.values = [ [] for i in range(self.size) ]
7      self.count = 0
```

hashing.py.

Nalezení položky

```
1  def get(h,key):
2    """ Vrati 'value' prvku s klicem 'key', jinak None """
3    m=hash(key) % h.size        # cislo prihradky
4    i=find_index(h.keys[m],key) # je tam?
5    if i is None: return None   # neni
6    return  h.values[m][i]
8  def find_index(l,x):
9    """ Vrati index 'i' aby l[i]==x nebo 'None'
10       pokud 'x' neni v 'l' """
11   for i,v in enumerate(l): # dvojice index, hodnota
12     if v==x: return i
13   return None
```

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

hashing.py

Vložení položky

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

Zvětšení tabulky

```
1  def grow_table(h):
2    """ Vytvori vetsi tabulku, prekopiruje tam obsah
3        a vrati ji """
5    hnew = Hashtable(2*h.size)
7    for i in range(h.size):      # okopiruj vse do hnew
8      keys=h.keys[i]
9      values=h.values[i]
10     for j in range(len(keys)):
11       put(hnew,keys[j],values[j])
13   return hnew
```

Získání obsahu tabulky

```
1  def items(h):
2    """ Vrati seznam dvojic klic, hodnota """
3    r=[]
4    for i in range(h.size):
5      r+=zip(h.keys[i],h.values[i])
6    return list(r)
```

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

Příklad

```
1  from hashing import *
3  t=Hashtable()
4  t=put(t,'pi',   3.14159)
5  t=put(t,'e',    2.71828)
6  t=put(t,'sqrt2',1.41421)
7  t=put(t,'golden',1.61803)
8  print(get(t,'pi'))

print(get(t,'e'))

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

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 """
4      counts={} # slovník
6      for w in words:
7        if w in counts:
8          counts[w]+=1
9        else:
10         counts[w]=1
12     return list(counts.items())
```

Spočítání slov (2) – Rozptylovací tabulka

```
1  import hashing
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
3  # implementace pomoci vyhledavaciho stromu
4  def word_counts_bst(words):
5      """ Vrati seznam dvojic slov a jejich frekvenci """
7      counts=None
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)
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' """
5      # setrid od nejcastejsiho
6      counts.sort(key=lambda x: x[1],reverse=True)
8      # celkovy pocet slov
9      nwords=functools.reduce(lambda acc,x: x[1]+acc,counts,0)
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.