Abstraktní datové typy

KIV/ADT – 3. přednáška

Miloslav Konopík, Libor Váša

1. března 2024

Obsah

Generické programy

Generické programování

Význam

Generické = obecné, společné

Generické programování

Význam

Generické = obecné, společné

snaha podchytit společné vlastnosti různých algoritmů/datových struktur

Generické programování

Význam

Generické = obecné, společné

snaha podchytit společné vlastnosti různých algoritmů/datových struktur

Antonymum

Specifické = konkrétní, speciální, zvláštní

Generické programování

Význam

Generické = obecné, společné

snaha podchytit společné vlastnosti různých algoritmů/datových struktur

Antonymum

Specifické = konkrétní, speciální, zvláštní

Cíle:

- umožnit záměnu různých specifických implementací splňujících nějaká generická kritéria
- umožnit sdílení generického kódu různými specifickými implementacemi

Problém

- Různé algoritmy řazení (bubblesort, insertsort, ...).
- Není dopředu jasné, který bude lepší použít.

Problém

- Různé algoritmy řazení (bubblesort, insertsort, ...).
- Není dopředu jasné, který bude lepší použít.
- Implementujeme oba/všechny a vyzkoušíme na reálných datech.
- Budeme chtít snadno přecházet od jednoho algoritmu k druhému.

Možné řešení

Vytvoříme pro každý algoritmus třídu, která ho svými metodami implementuje

```
1  def bubbleSort(self, data: list) -> None:
2      ...

3  def insertSort(self, data: list) -> None:
4      ...


5  ...
6  data1: list[int] = generateData1()
7  data2: list[int] = generateData2()
8  ...
9  bubbleSort(data1)
10 bubbleSort(data2)
11 ...
```

Možné řešení

Vytvoříme pro každý algoritmus třídu, která ho svými metodami implementuje

```
1  def bubbleSort(self, data: list) -> None:
2      ...

3  def insertSort(self, data: list) -> None:
4      ...
```

```
5  ...
6  data1: list[int] = generateData1()
7  data2: list[int] = generateData2()
8  ...
9  bubbleSort(data1)
10 bubbleSort(data2)
11 ...
```

```
12 ...
13 data1: list[int] = generateData1()
14 data2: list[int] = generateData2()
15 ...
16 insertSort(data1)
17 insertSort(data2)
18 ...
```

Lepší řešení

Dáme metodám se stejným významem stejnou signaturu (Duck typing).

```
1  def bubbleSort(data: list) -> None:
2      ...

3  def insertSort(data: list) -> None:
4      ...


5  ...
6  data1: list[int] = generateData1()
7  data2: list[int] = generateData2()
8  ...
9  sorter = bubbleSort
10 sorter(data1)
11 sorter(data2)
12 ...
```

Lepší řešení

Dáme metodám se stejným významem stejnou signaturu (Duck typing).

```
1  def bubbleSort(data: list) -> None:
2      ...

3  def insertSort(data: list) -> None:
4      ...
```

```
5  ...
6  data1: list[int] = generateData1()
7  data2: list[int] = generateData2()
8  ...
9  sorter = bubbleSort
10 sorter(data1)
11 sorter(data2)
12 ...
```

```
13 ...
14 data1: list[int] = generateData1()
15 data2: list[int] = generateData2()
16 ...
17 sorter = insertSort
18 sorter(data1)
19 sorter(data2)
20 ...
```

Rozhraní

Rozhraní:

- Popisuje, co nějaká metoda nebo třída umí.
- Definuje, co musí být splněno.
- Využíváme Duck typing – požadujeme nějaké vlastnosti nebo existenci funkce / metody.
- Definováno v dokumentaci nebo využitím dekorátorů `@` (později v přednáškách).

Příklad

Funkce pro řazení seznamu umí seřadit vstupní seznam. Vyžaduje vstupně-výstupní parametr seznamu s prvky, které lze porovnávat.

Abstraktní datové typy

Abstraktní

Abstraktní = nezabývají se rozdíly, ale společnými vlastnostmi

Abstraktní

Abstraktní = nezabývají se rozdíly, ale společnými vlastnostmi

Příklad

seznam pacientů v kartotéce

vs

seznam studentů

Abstraktní

Abstraktní = nezabývají se rozdíly, ale společnými vlastnostmi

Příklad

seznam pacientů v kartotéce
vs
seznam studentů

Společné:

- položky mají jméno
- umožňují přidávat a odebírat
- můžeme chtít řadit
- ...

Abstraktní

Abstraktní = nezabývají se rozdíly, ale společnými vlastnostmi

Příklad

seznam pacientů v kartotéce
vs
seznam studentů

Společné:

- položky mají jméno
- umožňují přidávat a odebírat
- můžeme chtít řadit
- ...

Rozdílné

- ne/mají chorobopis
- ne/mají seznam zapsaných předmětů
- ...

Absraktní datové typy

- definují možné operace nad daty
- nedefinují způsob uložení dat
- nedefinují způsob provedení operací (implementaci)

Absraktní datové typy

- definují možné operace nad daty
- nedefinují způsob uložení dat
- nedefinují způsob provedení operací (implementaci)

Příklad

ADT číslo:

- čísla lze sčítat, násobit, dělit...,
- `int`, `float`, `complex` ... jsou implementace tohoto ADT

ADT vs. rozhraní

ADT je obecný myšlenkový koncept, přesahující konkrétní jazyk

ADT vs. rozhraní

ADT je obecný myšlenkový koncept, přesahující konkrétní jazyk
rozhraní definuje ADT

ADT vs. rozhraní

ADT je obecný myšlenkový koncept, přesahující konkrétní jazyk
rozhraní definuje ADT

- říká co umí: hlavičky metod
- neříká jak se to dělá: těla metod

ADT vs. rozhraní

ADT je obecný myšlenkový koncept, přesahující konkrétní jazyk
rozhraní definuje ADT

- říká co umí: hlavičky metod
- neříká jak se to dělá: těla metod

Rozdíl:

ADT

součástí je sémantika, význam!

ADT vs. rozhraní

ADT je obecný myšlenkový koncept, přesahující konkrétní jazyk
rozhraní definuje ADT

- říká co umí: hlavičky metod
- neříká jak se to dělá: těla metod

Rozdíl:

ADT

součástí je sémantika, význam!

rozhraní

- jen říká jaká data jdou dovnitř a jaká ven
- co se s daty má stát musíme vědět
- tuto informaci nám dodá právě znalost ADT

ADT v ADT :)

- Budeme řešit především struktury uchovávající sady prvků, tzv. kolekce.
- Většina moderních jazyků poskytuje nějakou implementaci kolekcí.
- Musíme ale vědět, jaké mají kolekce vlastnosti.

Složitost práce s kolekcemi

```
1  class MyCollection:
2      def add(self, elem):
3          ...
4      def contains(self, elem):
5          ...

6  def containsDuplicates2(a: list[int]) -> bool:
7      c = MyCollection()
8      c.add(a[0])
9      for i in range(1, len(a)):
10         if c.contains(a[i]):
11             return True
12         c.add(a[i])
13     return False
```

Složitost práce s kolekcemi

```
1  class MyCollection:
2      def add(self, elem):
3          ...
4      def contains(self, elem):
5          ...

6  def containsDuplicates2(a: list[int]) -> bool:
7      c = MyCollection()
8      c.add(a[0])
9      for i in range(1, len(a)):
10         if c.contains(a[i]):
11             return True
12         c.add(a[i])
13     return False
```

Otázka

Lineární nebo kvadratický algoritmus?

Složitost práce s kolekcemi

```
1  class MyCollection:
2      def add(self, elem):
3          ...
4      def contains(self, elem):
5          ...

6  def containsDuplicates2(a: list[int]) -> bool:
7      c = MyCollection()
8      c.add(a[0])
9      for i in range(1, len(a)):
10         if c.contains(a[i]):
11             return True
12         c.add(a[i])
13     return False
```

Otázka

Lineární nebo kvadratický algoritmus?

Odpověď

Záleží na složitosti `add()` a `contains()`!

Výběr ADT

Konkrétní ADT:

- určuje složitost operací,
- obvykle reprezentována třídou,
- složitost je někdy dána také implementací.
  - metody třídy = operace nad ADT

Výběr ADT

Konkrétní ADT:

- určuje složitost operací,
- obvykle reprezentována třídou,
- složitost je někdy dána také implementací.
  - metody třídy = operace nad ADT

Úkol programátora

- vybrat vhodnou ADT,
- vybrat vhodnou implementaci ADT,
- vědět co dělá (jaká je složitost operací), když používá konkrétní ADT.

Kolekce

Kolekce

Otázka

Co bychom mohli od kolekce chtít?

Kolekce

Otázka

Co bychom mohli od kolekce chtít?

- přidat prvek (add)
  - na začátek
  - na konec
  - za/před nějaký prvek
  - s nějakým klíčem

Kolekce

Otázka

Co bychom mohli od kolekce chtít?

- přidat prvek (add)
  - na začátek
  - na konec
  - za/před nějaký prvek
  - s nějakým klíčem

Pozorování

Kolekce se může chovat jako seznam prvků s číselným indexem (řada), nebo jako množina (nemá indexy, nemá pořadí)

Operace nad kolekcí

Otázka

Co bychom mohli od kolekce chtít?

- vybrat prvek (get)
  - na začátku
  - na konci
  - na nějakém indexu
  - na nějakém místě (máme nějaké "ukazovátko")
  - s nějakým klíčem
  - s extrémním (nejvyšším, nejnižším) klíčem
- zjistit, zda kolekce obsahuje nějaký prvek

Operace nad kolekcí

Otázka

Co bychom mohli od kolekce chtít?

- odebrat prvek (remove)
  - na začátku
  - na konci
  - na nějakém indexu
  - na nějakém místě
  - s nějakým klíčem
  - s extrémním (nejvyšším, nejnižším) klíčem
  - všechny

Krátká terminologická poznámka

Vybrat (get)

- získáme hodnotu
- prvek v kolekci zůstává

Odebrat (remove)

- prvek je z kolekce odstraněn
- často nezískáme jeho hodnotu (někdy ale ano)

Pozor!

V češtině to zní podobně, buďme důslední v terminologii!

Kolekce

Otázka

Co bychom mohli od kolekce chtít?

- Procházet (iterovat) prvky
  - musí být definované pořadí (uspořádané kolekce)
  - náhodné pořadí (pokaždé jiné pořadí prvků, neuspořádané kolekce)
  - více možností (grafy, stromy, ...)
  - měnit kolekci během procházení (spojový seznam)

Kolekce

Otázka

Co bychom mohli od kolekce chtít?

- Procházet (iterovat) prvky
  - musí být definované pořadí (uspořádané kolekce)
  - náhodné pořadí (pokaždé jiné pořadí prvků, neuspořádané kolekce)
  - více možností (grafy, stromy, ...)
  - měnit kolekci během procházení (spojový seznam)

Pozorování

Procházení kolekcí je vlastnost daného ADT a nemusí být definováno pro všechny kolekce.

Více o procházení kolekcí v kapitole 4 – Iterátor .

Ideální kolekce

Ideální kolekce

- poskytuje všechny tyto možnosti v čase $\Theta(1)$
  - čas nezávisí na počtu prvků, které už v kolekci jsou

Ideální kolekce

Ideální kolekce

- poskytuje všechny tyto možnosti v čase $\Theta(1)$
  - čas nezávisí na počtu prvků, které už v kolekci jsou
- bohužel neexistuje

Ideální kolekce

Ideální kolekce

- poskytuje všechny tyto možnosti v čase $\Theta(1)$
  - čas nezávisí na počtu prvků, které už v kolekci jsou
- bohužel neexistuje
- v praxi ale většinou všechny možnosti nepotřebujeme

Ideální kolekce

Ideální kolekce

- poskytuje všechny tyto možnosti v čase $\Theta(1)$
  - čas nezávisí na počtu prvků, které už v kolekci jsou
- bohužel neexistuje
- v praxi ale většinou všechny možnosti nepotřebujeme

Abstraktní datový typ (ADT)

- určuje podmnožinu operací, které je možno provádět
- neurčuje implementaci
- složitost operací není určená

Obecné vlastnosti kolekcí

Pořadí:

- Uspořádaná kolekce udržuje pořadí svých prvků.
- Neuspořádaná kolekce nemá žádné specifické pořadí svých prvků.

Měnitelné nebo neměnitelné:

- Měnitelná kolekce umožňuje úpravu svých prvků po vytvoření.
- Neměnitelná kolekce nedovoluje žádné změny svých prvků po vytvoření.

Další vlastnosti:

- S náhodným přístupem (indexované nebo neindexované).
- Povolení duplicitních prvků nebo ano / ne.
- Pevná / dynamická velikost.

Iterátor

Iterátor

Iterátor v Pythonu 3 je objekt, který umožňuje postupný přístup k prvkům kolekce (seznam, slovník, množina apod.) nebo k prvkům nějakého jiného objektu, který podporuje postupné procházení.

- Příklad návrhového vzoru (více v poslední přednášce).
- Vytvořen voláním funkce `iter()` na třídu.
- Je možné vytvořit vlastní iterátor bez závislosti na jiné třídě.
- Třída iterátoru musí implementovat metody `iter()` a `next()`.
- Po vyčerpání dat vyvolá výjimku `StopIteration`.
- Nemusíme data do paměti najednou, ale můžeme je načítat postupně (lazzy loading).

Iterátor – příklad

```
1  from typing import Iterator

2  my_list: list[int] = [4, 7, 0]

3  # vytvořit iterátor ze seznamu
4  iterator: Iterator[int] = iter(my_list)

5  # získat první prvek iterátoru
6  print(next(iterator)) # vypíše 4

7  #získat druhý prvek iterátoru
8  print(next(iterator)) # vypíše 7

9  # získat třetí prvek iterátoru
10 print(next(iterator)) # vypíše 0
```

Vlastní iterátor – List

```
1  class ListIterator:
2      def __init__(self, items: list) -> None:
3          self.items: list = items
4          self.index: int = 0

5      def __iter__(self) -> Iterator:
6          return self

7      def __next__(self):
8          if self.index >= len(self.items):
9              raise StopIteration
10         value = self.items[self.index]
11         self.index += 1
12         return value
```

Vlastní iterátor – Soubory

```
1  class IntFileIterator:
2      def __init__(self, file_name: str) -> None:
3          self.file_name = file_name
4      def __iter__(self) -> Iterator[int]:
5          self.file = open(self.file_name)
6          return self
7      def __next__(self) -> int:
8          line: str = self.file.readline()
9          if not line:
10             self.file.close()
11             raise StopIteration
12         return int(line.strip())
```

Vlastní iterátor – použití

```
1  # vytvoř iterátor pro soubor 'numbers.txt'
2  iterator = IntFileIterator('numbers.txt')

3  #  vypiš pruky z iterátoru
4  for number in iterator:
5      print(number)
```

Generátor

Stejná funkce jako Iterátor, ale zjednodušená syntax a odstranění boilerplate. Zavádí speciální klíčové slovo `yield`.

```
1  def int_file_generator(file_name: str) -> Iterator[int]:
2      with open(file_name) as file:
3          for line in file:
4              yield int(line.strip())

5  #  vypiš pruky z generátoru
6  for number in int_file_generator("numbers.txt"):
7      print(number)
```

Operace s interátory a generátory I

- **Comprehensions** – operace umožňující efektivní zpracování iterátorů a generátorů.
  - Syntaxe: `[výraz for položka in iterovatelný objekt if podmínka]`
- **Příklad:**
  - `ctverce = [x**2 for x in range(10) if x % 2 == 0]`
  - Vytvoří seznam druhých mocnin sudých čísel od 0 do 9.
- **Operace s generátory** – poskytují podobnou funkčnost pro generátory.
  - Syntaxe: `(výraz for položka in iterovatelný objekt if podmínka)`
- **Příklad:**
  - `ctverce_gen = (x**2 for x in range(10))`
  - Vytváří generátor čtverců čísel od 0 do 9.

Operace s interátory a generátory II

- Operace `zip` slouží k agregaci (paralelnímu procházení) prvků z dvou nebo více iterovatelných objektů (např. seznamů, n-tic).
- Výsledkem je iterátor n-tic, kde i-tá n-tice obsahuje i-té prvky z každého z vstupních iterovatelných objektů.
- Pokud mají vstupní iterovatelné objekty různé délky, `zip` vytvoří n-tice až do délky nejkratšího vstupu.
- Příklad použití:

```
seznam1 = [1, 2, 3, 4]
seznam2 = ['a', 'b', 'c']
vysledek = list(zip(seznam1, seznam2))
# Vysledek: [(1, 'a'), (2, 'b'), (3, 'c')]
```

ADT Seznam

Abstraktní datová struktura Seznam

Seznam je uspořádaná, měnitelná kolekce prvků s duplicitami.

Podporované operace:

- umožňuje vložení prvku na jakoukoli pozici,
- umožňuje vybrání / odebrání prvku z jakékoli pozice,
- procházení seznamu.

Abstraktní datová struktura Seznam

Seznam je uspořádaná, měnitelná kolekce prvků s duplicitami.

Podporované operace:

- umožňuje vložení prvku na jakoukoli pozici,
- umožňuje vybrání / odebrání prvku z jakékoli pozice,
- procházení seznamu.

Běžné implementace:

- polem (`list`),
- spojovým seznamem `from collections import deque`.

ADT Zásobník

ADT zásobník (Stack)

Speciální verze ADT Seznam.
Operace (ideálně v $\Theta(1)$):

- přidej prvek na konec
- vyber prvek na konci
- odeber prvek z konce

ADT zásobník (Stack)

Speciální verze ADT Seznam.
Operace (ideálně v $\Theta(1)$):

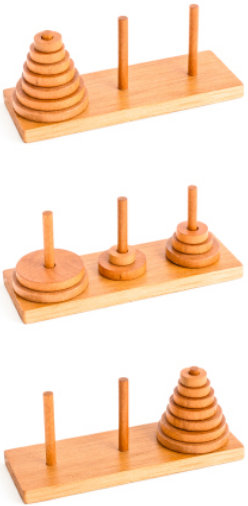
- přidej prvek na konec
- vyber prvek na konci
- odeber prvek z konce

Použití:

- seznamy věcí, které je třeba vyřídit (když chceme nové úkoly řešit jako první)
- pro uložení hodnot, které potřebujeme dočasně změnit, ale ke kterým se budeme chtít vrátit
- Hanojské věže.
- LIFO: Last In, First Out

Zásobník

Zásobník



ADT zásobník – příklad v Pythonu

```
1  from collections import deque

2  stack = deque(['jablko', 'hruška', 'banán'])

3  # přidat hodnotu do zásobníku
4  stack.append('pomeranč')
5  stack.append('kiwi')

6  # odebrat prvek z konce
7  top_value = stack.pop()          # 'kiwi'
8  print(top_value)

9  # vybrat prvek z konce
10 top_value = stack[-1]            # 'pomeranč'
11 print(top_value)
```

ADT Fronta

Fronta (Queue)

Struktura podobná ADT Seznam a Zásobník.
Operace (ideálně v $\Theta(1)$):

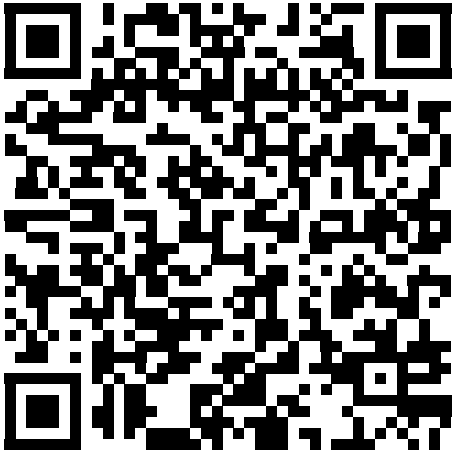
- přidání prvku na konec
- vybrání prvku na začátku
- odebrání prvku ze začátku
- místo posledního umožňuje vybrat/odebrat první prvek
  - ten který se v kolekci nachází nejdelší dobu
- přirozená reprezentace pro odpovídající situaci:
  - vyřizování požadavků v tom pořadí, v jakém vznikly
- FIFO: First In, First Out

Fronta (Queue)

ADT Fronta – příklad v Pythonu

```
1  from collections import deque

2  queue = deque(['jablko', 'hruška', 'banán'])

3  # přidání pruku na konec
4  queue.append('pomeranč')
5  queue.append('kiwi')

6  # odebrání pruku ze začátku
7  front_value = queue.popleft() # 'jablko'
8  print(front_value)

9  # vybrání pruku na začátku
10 front_value = queue[0] # 'hruška'
11 print(front_value)
```

Kvíz I



ADT Množina

ADT Množina

Množina (Set) je kolekce jedinečných, neuspořádaných prvků. Základní operace nad množinou:

- Přidání prvku: Prvek může být přidán do množiny pouze tehdy, pokud v Množině dosud nebyl přítomen.
- Odebrání prvku: Prvek může být odebrán z množiny.
- Kontrola přítomnosti prvku: Množina umožňuje zkontrolovat, zda je daný prvek přítomen v Množině nebo ne.

ADT Množina

Další operace nad množinou:

- Sjednocení množin: Dvě množiny mohou být spojeny (sjednoceny) tak, že vznikne nová Množina obsahující všechny prvky obou původních Množin.
- Průnik: Průnik dvou Množin je nová Množina obsahující pouze prvky, které jsou společné pro obě množiny.
- Rozdíl: Rozdíl mezi dvěma Množinami je nová Množina obsahující pouze prvky, které jsou přítomny v jedné Množině, ale nevyskytují se v druhé.
- Podmnožina: Množina je podmnožinou jiné množiny, pokud jsou všechny její prvky přítomny v té druhé Množině.
- Kardinalita: Počet prvků v Množině se nazývá kardinalita.

ADT Množina

Proč používat Množinu:

- Kontrola přítomnosti prvku má složitost $\Theta(1)$.
- Vysvětlení: další přednáška.

ADT Tabulka

ADT Tabulka

- jednosměrný vztah mezi množinou klíčů a hodnot
- tabulka reprezentuje přiřazení jedné nebo žádné hodnoty každému klíči

Klíč

- diskrétní datový typ
  - `int`
  - `String`

Klíč

- diskrétní datový typ
  - `int`
  - `String`
  - ne `float`!

Klíč

- diskrétní datový typ
  - `int`
  - `String`
  - ne `float`!

Nejdůležitější vlastnost

Musí být možné zjistit, zda jsou si dvě hodnoty rovny

Klíč

- diskrétní datový typ
  - `int`
  - `String`
  - ne `float`!

Nejdůležitější vlastnost

Musí být možné zjistit, zda jsou si dvě hodnoty rovny

Při vhodné implementaci vyhodnocení rovnosti je možné použít i složitější struktury (např. množina čísel $\{0, 1, 2\}$ je shodná s množinou $\{2, 0, 1\}$, množina celých čísel tedy může být klíčem)

Hodnota

- libovolný datový typ
- často reference na instanci třídy
  - instance obsahuje celou sadu dat
  - instance může být obsažena v několika různých datových strukturách (např. v Tabulce a zároveň v Seznamu)

Příklad 1 (tabulka studentů)

Klíč: studijní číslo (`String`)
Hodnota: záznam studenta (reference na instanci třídy `Student`)

Klíč	Hodnota
A16N0123P	František Vonásek Křepelková 5 Hronov
A17N0321P	Kateřina Čumáčková Bezbřehá 8 Srnčí nad Smetanou
A17P0314P	Tomáš Marný Polévková 13 Želvice

Příklad 2 (tabulka faktur)

Klíč: číslo faktury (`int`)
Hodnota: reference na instanci třídy reprezentující fakturu

Klíč	Hodnota
2019002	Plátce: AAA Auto Celková cena: 1234,- Položky: ...
2019004	Plátce: BBB Bedly Celková cena: 4321,- Položky: ...
2018001	Plátce: Plyšmyši, s.r.o Celková cena: 16,50 Položky: ...

Příklad 3 (jídelní lístek)

Klíč: název položky (`String`)
Hodnota: cena (`double`)

Klíč	Hodnota
"Dršťková"	65,-
"Utopenec"	45,-
"Šunknfleky"	75,90

Příklad 3 (jídelní lístek)

Klíč: název položky (`String`)
Hodnota: cena (`double`)

Klíč	Hodnota
"Dršťková"	65,-
"Utopenec"	45,-
"Šunknfleky"	75,90

Pozor!

Takováto tabulka není vhodná pro reprezentaci účtu (proč?)

Vlastnosti ADT Tabulka

- není určeno pořadí prvků
  - je možné ho odvodit seřazením podle klíče
  - klíče samy řazení nemusí nutně podporovat (např. množina)

Vlastnosti ADT Tabulka

- není určeno pořadí prvků
  - je možné ho odvodit seřazením podle klíče
  - klíče samy řazení nemusí nutně podporovat (např. množina)
- některým (mnoha) možným hodnotám klíče nemusí být přiřazena žádná hodnota

Operace nad tabulkou

- přidání přiřazení (klíč + hodnota)

Operace nad tabulkou

- přidání přiřazení (klíč + hodnota)
- získání hodnoty pro daný klíč

Operace nad tabulkou

- přidání přiřazení (klíč + hodnota)
- získání hodnoty pro daný klíč
- zjištění, zda je klíči přiřazena hodnota

Operace nad tabulkou

- přidání přiřazení (klíč + hodnota)
- získání hodnoty pro daný klíč
- zjištění, zda je klíči přiřazena hodnota
- zrušení přiřazení (odebrání klíče)

Operace nad tabulkou

- přidání přiřazení (klíč + hodnota)
- získání hodnoty pro daný klíč
- zjištění, zda je klíči přiřazena hodnota
- zrušení přiřazení (odebrání klíče)
- získání všech klíčů, kterým je přiřazena hodnota

ideálně $\Theta(1)$ složitost operací (kromě získání všech klíčů)

Kvíz II