6. Abstraktní datový typ

BAB37ZPR – Základy programování

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Abstraktní datové typy

  Seznam

  Zásobník

  Fronta

  Množina

Část I

Abstraktní datové typy

Abstraktní datové typy

Datový typ

- rozsah hodnot, které patří do daného typu
- operace, které je možno s těmito hodnotami provádět

Abstraktní datový typ (ADT)

- rozhraní
- popis operací, které chceme provádět

Konkrétní datová struktura

- implementace
- přesný popis uložení dat v paměti
- definice funkcí pro práci s těmito daty

Dva pohledy na data

Abstraktní

- operace, které budu s daty provádět
- co musí operace splňovat
- například množina: ulož, najdi, vymaž
- tento předmět

Výhody: snadný vývoj, jednodušší přemýšlení o problémech

Riziko: svádí k ignorování efektivity

Implementační

- jak jsou data uložena v paměti
- jak jsou operace implementovány
- například binární vyhledávací strom
- navazující předmět :-)

Nejpoužívanější ADT

- seznam
- zásobník
- fronta
- množina
- slovník (asociativní pole)

I. Abstraktní datové typy

Seznam

Zásobník

Fronta

Množina

Různé varianty seznamu

- obsahuje posloupnost prvků
  - stejného typu
  - různého typu
- přidání prvku
  - na začátek
  - na konec
  - na určené místo
- odebrání prvku
  - ze začátku
  - z konce
  - konkrétní prvek
- test prázdnosti, délky
- případně další operace, např. přístup pomocí indexu

Seznamy v Pythonu – operace

Opakování

- seznamy v Pythonu velmi obecné
- prvky mohou být různých typů
- přístup skrze indexy
- indexování od konce pomocí záporných čísel
- seznamy lze modifikovat na libovolné pozici

```
1 a = ['bacon', 'eggs', 'spam', 42]
2 print(a[1:3]) # ['eggs', 'spam']
3 print(a[-2:-4:-1]) # ['spam', 'eggs']
4 a[-1] = 17
5 print(a) # ['bacon', 'eggs', 'spam', 17]
6 print(len(a)) # 4
```

Seznamy v Pythonu – operace

```
s = [4, 1, 3]  # seznam
s.append(x)    # prida prvek x na konec
s.extend(t)    # prida vsechny prvky t na konec
s.insert(i, x) # prida prvek x pred prvek na pozici i
s.remove(x)    # odstrani prvni prvek rovny x
s.pop(i)       # odstrani (a vrati) prvek na pozici i
s.pop()        # odstrani (a vrati) posledni prvek
s.index(x)     # vrati index prvniho prvku rovneho x
s.count(x)     # vrati pocet vyskytu prvku rovnych x
s.sort()       # seradi seznam
s.reverse()    # obrat seznam
x in s         # test, zda seznam obsahuje x
# (linearni pruchod seznamem!)
```

Seznamy v Pythonu – generátorová notace

- Specialita Pythonu

```
1  s = [x for x in range(1, 7)]
2  print(s)
```

```
[1, 2, 3, 4, 5, 6]
```

```
1  s = [2 * x for x in range(1, 7)]
2  print(s)
```

```
[2, 4, 6, 8, 10, 12]
```

```
1  s = [(a, b) for a in range(1, 2) for b in ["A", "B"]]
2  print(s)
```

```
[(1, 'A'), (1, 'B'), (2, 'A'), (2, 'B')]
```

I. Abstraktní datové typy

Seznam

Zásobník

Fronta

Množina

Zásobník

- strukturovaný/složený datový typ
- obsahuje předem neznámé množství položek, typicky stejného typu *(jako seznam/pole)*
- podporuje následující operace (se složitostí $O(1)$)
  - přidání položky na konec – `push`
  - odebrání položky z konce – `pop`
  - test, jestli je zásobník prázdný – `is_empty`
- položky jsou odebírány v opačném pořadí, než byly přidány.
  LIFO – last in first out
- zásobník může podporovat i další operace
  - nedestruktivní čtení z konce – `peek`, `top`
  - zjištění počtu položek na zásobníku – `size`

Zásobník – příklad použití

```
1  from stack import Stack
3  s=Stack()
5  s.push(1)
6  s.push(2)
7  s.push(3)
9  print(s.pop())
10 print(s.pop())
12 s.push(10)
14 print(s.pop())
15 print(s.is_empty())
16 print(s.pop())
17 print(s.is_empty())
```

Zásobník – implementace pomocí pole

```
1  class Stack:
2    def __init__(self):
3      self.items = []
5    def size(self):
6      return len(self.items)
8    def is_empty(self):
9      return self.size()==0
11   def push(self, item):
12     self.items+=[item]
14   def pop(self):
15     return self.items.pop()
17   def peek(self):
18     return self.items[-1]
```

Příklad – převod do jiné číselné soustavy

```
1  from stack import Stack
3  def to_str(n,base):
4    cislice = "0123456789ABCDEF"
5    assert(n>=0)
6    stack=Stack()
7    while True:
8      stack.push(n % base)
9      n //= base
10     if n==0: break
11   result=""
12   while not stack.is_empty():
13     result+=cislice[stack.pop()]
14   return result
16 print(to_str(67,2))
```

Příklad – kontrola uzávorkování 1/2

```
1  def par_checker(s):
3    L = "([{"  # otevírací závorky
4    R = ")]}"  # uzavírací závorky (stejné pořadí)
5    z = Stack()
7    for c in s:
8      if c in L:
9        z.push(c)
10     else:
11       for i in range(len(R)):
12         if c == R[i]:
13           if z.is_empty() or z.pop()!=L[i]:
14             return False
16   return stack.is_empty()
```

stack_examples.py

Příklad – kontrola uzávorkování 2/2

- Příklady

```
print(par_checker("(4+(3*[a+b]))"))
```

```
True
```

```
print(par_checker("(x+([21*c]-5}*6)"))
```

```
False
```

```
print(par_checker("[(3+4)*7-{}*(((0)+(1))%7)]"))
```

```
True
```

```
print(par_checker("{ { ( [ ] [ ] ) } ( ) }"))
```

```
True
```

Postfixová notace

- Klasická notace je *infixová* (operátor mezi operandy)
- *Postfixová notace* (operátor po argumentech)
  - Nepotřebuje závorky.
  - Snadné vyhodnocení pomocí **zásobníku**

Existují zásobníkové programovací jazyky (FORTH, Postscript, bibtex).

infix	postfix
`12 / 4`	`12 4 /`
`3 * 4 - 2`	`3 4 * 2 -`
`3 * (4 - 2)`	`3 4 2 - *`
`(62-32)*5/9`	`62 32 - 5 * 9 /`

Existuje i notace prefixová: `/ 12 4`, `- * 3 4 2`,...

Postfixová notace a zásobník

Vyhodnocení výrazu:

- *číslo* na vstupu vložíme do zásobníku.
- *operand* vezme dvě čísla ze zásobníku a vloží do zásobníku výsledek operace.

infix	postfix
`12 / 4`	`12 4 /`
`3 * 4 - 2`	`3 4 * 2 -`
`3 * (4 - 2)`	`3 4 2 - *`
`(62-32)*5/9`	`62 32 - 5 * 9 /`

Vyhodnocení postfixového výrazu 1/2

```
1  def eval_postfix(s):
2    stack=Stack()
3    for x in s.split(): # rozděl 's' dle mezer
4      if x=='+':
5        stack.push(stack.pop()+stack.pop())
6      elif x=='-':
7        stack.push(-stack.pop()+stack.pop())
8      elif x=='*':
9        stack.push(stack.pop()*stack.pop())
10     elif x=='/':
11       second=stack.pop()
12       stack.push(stack.pop()/second)
13     else: # 'x' je číslo
14       stack.push(float(x))
15   return stack.pop()
```

Vyhodnocení postfixového výrazu 2/2

- Příklady

```
print(eval_postfix("3 4 *"))
```

```
12.0
```

```
print(eval_postfix("10 6 -"))
```

```
4.0
```

```
print(eval_postfix("20 4 /"))
```

```
5.0
```

```
print(eval_postfix("3 4 * 2 -")) # 3 * 4 - 2
```

```
print(eval_postfix("3 4 2 - *")) # 3 * (4 - 2)
```

Převod infixového na postfixový výraz

Edsger Dijkstra: Shunting yard algorithm *(seřaďovací nádraží)*

1. **Číslo** okopíruj na výstup.
2. **Operátor** ulož do zásobníku operátorů.
   (a) Je-li v zásobníku operátor s vyšší precedencí, přesuň tento operátor nejdřív na výstup.
3. **Otevírací závorku** ulož do zásobníku.
4. Po přečtení **uzavírací závorky** přesouvej ze zásobníku na výstup, dokud nenarazíš na otevírací závorku, kterou zahoď.
5. Nakonec přesuň ze zásobníku na výstup zbývající operátory.

`stack_examples.py`

funkce **infix_to_postfix**

Převod infixového na postfixový výraz

- Příklady

```
print(infix_to_postfix("32+4"))
```

```
32 4 +
```

```
print(infix_to_postfix("3*4-2"))
```

```
3 4 * 2 -
```

```
print(infix_to_postfix("3*(4-2)"))
```

```
3 4 2 - *
```

```
print(infix_to_postfix("(62-32)*5/9"))
```

```
62 32 - 5 * 9 /
```

Vyhodnocování infixových výrazů

- Výraz převedeme na postfixový a vyhodnotíme.

```
1  def eval_infix(s):
2    return eval_postfix(infix_to_postfix(s))
4  print(eval_infix("32+4"))
```

```
36.0
```

```
1  print(eval_infix("3*4-2"))
```

```
10.0
```

```
1  print(eval_infix("3*(4-2)"))
```

```
6.0
```

```
1  print(eval_infix("(62-32)*5/9"))
```

```
16.666666666666668
```

I. Abstraktní datové typy

Seznam

Zásobník

Fronta

Množina

Fronta (Queue)

- strukturovaný/složený datový typ
- obsahuje předem neznámé množství položek, typicky stejného typu *(jako pole)*
- podporuje následující operace (se složitostí $O(1)$)
  - přidání položky na konec *(enqueue, add)*
  - odebrání položky **ze začátku** *(dequeue, top)*
  - test, jestli je fronta prázdná *(is_empty)*
- položky jsou odebírány ve **stejném** pořadí, jako byly přidány. *(FIFO — first in first out)*
- fronta může podporovat i další operace
  - nedestruktivní čtení ze začátku *(peek)*
  - zjištění počtu položek ve frontě *(size)*
- Pokročilejší varianta: prioritní fronta

Fronta – implementace v Pythonu

- implementace pomocí seznamů snadná, ale neefektivní
  - přidávání a odebírání na začátku seznamu vyžaduje přesuny
  - pomalé pro dlouhé fronty
  - přesto si implementaci ukážeme
- použití knihovny `collections`
  - datový typ `deque` (oboustranná fronta)
  - vložení do fronty pomocí `append`
  - odebrání z fronty pomocí `popleft`
  - přední prvek fronty je `[0]`

```
1 from collections import deque
2 q = deque(["Eric", "John", "Michael"])
3 q.append("Terry") # Terry arrives
4 q.append("Graham") # Graham arrives
5 q.popleft() # Eric leaves
6 q.popleft() # John leaves
7 print(q) # deque(['Michael', 'Terry', 'Graham']
```

Fronta – implementace v Pythonu

- Příklad vlastní implementace

```
1  from slowqueue import Queue
3  q=Queue()
4  q.enqueue(1)
5  q.enqueue(2)
6  q.enqueue(3)
7  print(q.dequeue())
8  print(q.dequeue())
9  q.enqueue(10)
10 print(q.dequeue())
11 print(q.is_empty())
12 print(q.dequeue())
13 print(q.is_empty())
```

queue_examples.py

Fronta – použití

- Komunikace mezi procesy *(consumer,producer)*
- Čekání na asynchronní periferie — klávesnice, disk, síť ...
- 'Férový přístup' pro sdílení zdrojů *(policy)*
- Simulace čekání ve frontě
- Některé grafové a třídící algoritmy *(prohledávání do šířky, merge sort)*

Fronta – implementace pomocí seznamu

```
1  class Queue:
3    def __init__(self):
4      self.items = []
6    def is_empty(self):
7      return self.items == []
9    def enqueue(self, item):
10     self.items.insert(0,item)
12   def dequeue(self):
13     return self.items.pop()
15   def size(self):
16     return len(self.items)
```

Problém: složitost vkládání je $O(n)$.

slowqueue.py

Fronta – seznam s realokací 1/2

- Prvky ukládáme do seznamu.
- Vkládání na konec seznamu ($O(1)$).
- Prvky nemažeme, ale posouváme index počátku.
- Pokud je vynechaných prvků hodně, překopírujeme frontu do nového pole.
- Kopírujeme po poklesu využití paměti na 50 %
- Odebrání n prvků vyžaduje $\sim \log_2 n$ kopírování, dohromady $n/2 + n/4 + \cdots \sim n$ prvků $\rightarrow$ amortizovaná složitost odebrání prvku je $O(1)$.
- Nevýhoda – neefektivní využití paměti.

`arrayqueue.py`

Fronta – pole s realokací


```
1  class Queue:
3    def __init__(self):
4      self.front = 0   # index prvního prvku
5      self.items = []
7    def is_empty(self): return len(self.items) == self.front
9    def enqueue(self, item): self.items+=[item]
11   def dequeue(self):
12     el = self.items[self.front]
13     self.front+=1
14     if self.front >= 1024 and self.front >= len(self.items)//2:
15        self.items = self.items[self.front:]
16        self.front = 0
17     return el
19   def size(self): return len(self.items)
```

arrayqueue.py

Fronta – dva zásobníky (Donald Knuth) 1/2

- Prvky ukládáme do zásobníku `inp` ($O(1)$)
- Prvky odebíráme ze zásobníku `out` ($O(1)$)
- Když je `out` prázdný, přesuneme do něj `inp`
- Každý prvek je kopírován jen jednou, složitost zůstává $O(1)$

Fronta – dva zásobníky 2/2

```
1  class Queue:
3    def __init__(self):
4      self.inp = Stack()
5      self.out = Stack()
7    def is_empty(self):
8      return self.size()==0
10   def enqueue(self, item):
11     self.inp.push(item)
13   def dequeue(self):
14     if self.out.is_empty():
15       while not self.inp.is_empty():
16         self.out.push(self.inp.pop())
17     return self.out.pop()
19   def size(self):
20     return self.inp.size() + self.out.size()
```

knuthqueue.py

Příklad: Rozpočítávání 1/3

- Děti v kruhu, rozpočítávadlo má m slabik, začíná se prvním.
- Na koho padne poslední slabika, vypadává.
- Hraje se, dokud nevypadne poslední.
- Děti reprezentujeme pomocí fronty, budeme přesouvat ze začátku na konec.

```
from rozpocitavani import *
v=rozpocitej(["Adam","Bara","Cyril","David","Emma", "Franta","Gabina"],
    3)
print("Vyhral(a): ", v)
```

rozpocitavani.py

Příklad: Rozpočítávání 1/3

```
1  from knuthqueue import Queue
3  def rozpocitej(jmena,m):
4    q=Queue()
5    for j in jmena:   # uloz jmena do fronty
6      q.enqueue(j)
7      while q.size()>1:
8        for i in range(m-1):
9          q.enqueue(q.dequeue())
10         print("Vypadl(a): ",q.dequeue())
12     return q.dequeue()  # vrať vítěze
14 if __name__=="__main__":
15   v = rozpocitej(["Adam", "Bara", "Cyril", "David", "Emma", "Franta",
       "Gabina"],3)
16   print("Vyhral(a): ",v)
```

Příklad: Rozpočítávání 1/3

```
1 v = rozpocitej(["Adam", "Bara", "Cyril", "David", "Emma", "Franta", "
      Gabina"],3)
2 print("Vyhral(a): ",v)
```

```
Vypadl(a):  Cyril
Vypadl(a):  Franta
Vypadl(a):  Bára
Vypadl(a):  Gábina
Vypadl(a):  Emma
Vypadl(a):  Adam
Vyhrál(a):  David
```

Příklad: Tisková fronta

- Máme *n* uživatelů, náhodně posílají požadavky na tisk *(úlohy)* s náhodným počtem stran.
- Tiskárna tiskne danou rychlostí úlohy v pořadí, jak přicházejí *(fronta)*.
- Chceme simulovat a zjistit průměrnou dobu čekání na dokončení úlohy.

`tiskova_fronta.py`

Tisková fronta – implementace (1/4)

- Stav tiskárny a úlohy jsou záznamy *(records)*.

```
1  from knuthqueue import Queue
2  import random
4  class Uloha:
6    def __init__(self,t,s):
7      self.time=t    # čas vytvoření
8      self.stran=s   # počet stran
10 class Tiskarna:
12   def __init__(self):
13     self.uloha=None
14     self.zbyvajici_cas=0
```

Tisková fronta – implementace (2/4)

```
1  def simuluj(num_people,prob_second,max_pages,seconds_per_page,
       simulation_time):
2    """ Nasimuluje chovani tiskove fronty.
3    "num_people" - pocet lidi
4    "prob_second" - pravd. vytvoreni ulohy v dane vterine
5    "max_pages"         - maximalni pocet stran v uloze
6    "seconds_per_page" - pocet sekund na stranku
7    "simulation_time"  - delka simulace v sekundach """
8    q=Queue()       # tiskova fronta
9    t=Tiskarna()    # stav tiskarny
10   casy_cekani=[] # délky čekání na vytisknutí v sekundách
12   for i in range(simulation_time):
13     simuluj_lidi(num_people,prob_second,max_pages,q,i)
14     simuluj_tiskarnu(seconds_per_page,q,t,i,casy_cekani)
15   avg_time=sum(casy_cekani)/len(casy_cekani)
16   print("Prumerna doba cekani  %5.2fs." % avg_time)
17   return avg_time
```

Tisková fronta – implementace (3/4)

```
def simuluj_lidi(num_people,prob_second,max_pages,q,i):
  for j in range(num_people):
    if random.random()<prob_second:
      pocet_stran=random.randrange(1,max_pages+1)
      print("Cas %d, pozadavek na tisk %d stran." %
        (i,pocet_stran))
    q.enqueue(Uloha(i,pocet_stran))
```

Tisková fronta – implementace (4/4)

```
def simuluj_tiskarnu(seconds_per_page,q,t,i,casy_cekani):
  if t.uloha!=None:       # tiskarna tiskne
  t.zbyvajici_cas-=1
  if t.zbyvajici_cas<=0: # hotovo
    print("Cas %d, tisk %d stran/y hotov, cekani %5.1fs."
          % (i,t.uloha.stran,i-t.uloha.time))
    casy_cekani+=[i-t.uloha.time]
    t.uloha=None
  if t.uloha==None:       # tiskarna je volna
    if not q.is_empty():  # ve fronte je uloha
      t.uloha=q.dequeue()
      print("Cas %d, zaciname tisknout ulohu majici %d stran."
                    % (i,t.uloha.stran))
      t.zbyvajici_cas=t.uloha.stran*seconds_per_page
```

Tisková fronta – příklad

```
1 | simuluj(10,1./(60.*60.),10,12,100*60*60)
```

```
Cas 104, pozadavek na tisk 4 stran.
Cas 104, zaciname tisknout ulohu majici 4 stran.
Cas 152, tisk 4 stran/y hotov, cekani  48.0s.
Cas 1201, pozadavek na tisk 8 stran.
Cas 1201, zaciname tisknout ulohu majici 8 stran.
Cas 1297, tisk 8 stran/y hotov, čekáni  96.0s.
Cas 2185, zaciname tisknout ulohu majici 6 stran.
Cas 2188, pozadavek na tisk 7 stran.
Cas 2257, tisk 6 stran/y hotov, cekani  72.0s.
Cas 2257, zaciname tisknout ulohu majici 7 stran.
Cas 2341, tisk 7 stran/y hotov, cekani 153.0s.
...
Prumerna doba cekani  73.30s.
```

I. Abstraktní datové typy

Seznam

Zásobník

Fronta

Množina

Množina

- neuspořádaná kolekce dat bez vícenásobných prvků
- operace
  - insert (vložení)
  - find (vyhledání prvku, test přítomnosti)
  - remove (odstranění)
- použití
  - grafové algoritmy (označené navštívených vrcholů)
  - rychlé vyhledávání
  - výpis unikátních slov

Množina v Pythonu

```
set(alist) # vytvori mnozinu ze seznamu
len(s) # pocet prvku mnoziny s
s.add(x) # pridani prvku do mnoziny
s.remove(x) # odebrani prvku z mnoziny
x in s # test, zda mnozina obsahuje x
s1 <= s2 # test, zda je s1 podmnozinou s2
s1.union(s2) # sjednoceni mnozin s1 a s2
s1 | s2 # -- totez --
s1.intersection(s2) # prunik mnozin s1 a s2
s1 & s2 # -- totez --
s1.difference(s2) # rozdil mnozin s1 a s1
s1 - s2 # -- totez --
s1.symmetric_diference(s2) # symetricky rozdil mnozin
s1 ^ s2 # -- totez --
```

Množina v Pythonu

```
basket = ['apple', 'orange', 'apple', 'orange', 'banana']
fruit = set(basket)
print(fruit) # 'orange', 'apple', 'banana'
print('orange' in fruit) # True
print('tomato' in fruit) # False

a = set("abracadabra")
b = set("engineering")

print(a) # 'a', 'r', 'b', 'c', 'd'
print(b) # 'i', 'r', 'e', 'g', 'n'

print(a | b) # 'a', 'c', 'b', 'e', 'd', 'g', 'i', 'n',
print(a & b) # 'r'
print(a - b) # 'a', 'c', 'b', 'd'
print(a ^ b) # 'a', 'c', 'b', 'e', 'd', 'g', 'i', 'n'
```