

8. Abstraktní datový typ

Základy programování

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Abstraktní datové typy

Seznam

Množina

- Část 2 – Zásobník

P8.1 Kontrola uzávorkování

P8.2 Postfixová notace

- Část 3 – Fronta

P8.3 Rozpočítávání

Datové struktury

Na datové struktury se lze dívat ze dvou pohledů:

- Z pohledu vnitřní realizace
 - **pole** – posloupnost proměnných stejného typu množina,
 - **záznam** – struct, skupina proměnných různého typu
 - **objekt** – datovou strukturu tvoří pouze atributy instancí, atributy třídy jsou globální proměnné pouze formálně přidružené k dané třídě, proto se ukládají obvykle nezávisle na instancích třídy
 - **množina**
 - **řetězec** – string, posloupnost znaků
- Podle logiky přístupu k jednotlivým položkám
 - **strom**
 - **seznam**
 - **zásobník**
 - **fronta**

Další kritéria klasifikace datových struktur

- Přístup k položkám struktury
 - **sekvenční** – např. jednosměrný seznam, uspořádaná posloupnost položek, kde každá položka obsahuje odkaz na následující položku,
 - **přímý** – např. dynamické pole, položky jsou přístupné pomocí indexů.
- Organizace dat
 - **lineární** – v řadě za sebou, pole, seznam, zásobník a fronta,
 - **nelineární** – binární strom.
- Fyzické vymezení prostoru
 - **statické struktury** – mají svou velikost přesně a neměně určenou v okamžiku překladu zdrojového kódu,
 - **dynamické struktury** – mohou měnit počet položek za běhu programu - seznam, dynamické pole.

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
- + snadný vývoj, jednodušší přemýšlení o problémech
- svádí k ignorování efektivity

Implementační

- jak jsou data uložena v paměti
- jak jsou operace implementovány
- například binární vyhledávací strom
- navazující předmět – PRGA

Nejpoužívanější ADT

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

I. Abstraktní datové typy

Seznam

Množina

Seznam

- Seznam (proměnných nebo objektů) patří mezi základní datové struktury

Základní ADT – Abstract Data Type

- Seznam zpravidla nabízí sadu základních operací:
 - `insert()` – vložení prvku
 - `remove()` – odebrání prvku
 - `index_of()` – vyhledání prvku
 - `size()` – aktuální počet prvku v seznamu
- Implementace seznamu může být různá:
 - Pole
 - Indexování je velmi rychlé
 - Vložení prvku na konkrétní pozici může být pomalé
 - Spojové seznamy

Nová alokace a kopírování.

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 prázdnoti, 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

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

```
>>> s = [4, 1, 3] # seznam
>>> x = 10; t = [12, 10]; i = 2
>>> s.append(x)    # prida prvek x na konec
>>> s
[4, 1, 3, 10]
>>> s.extend(t)   # prida vsechny prvky t na konec
>>> s
[4, 1, 3, 10, 12, 10]
>>> s.insert(i, x) # prida prvek x pred prvek na pozici i
>>> s
[4, 1, 10, 3, 10, 12, 10]
>>> s.remove(x)   # odstrani prvnι prvek rovnι x
>>> s
[4, 1, 3, 10, 12, 10]
```

```
>>> s.pop(i)           # odstrani (a vrati) prvek na pozici i
3
>>> s.pop()           # odstrani (a vrati) posledni prvek
10
>>> s.index(x)        # vrati index prvnioho prvku rovnego x
2
>>> s.count(x)        # vrati pocet vyskytu prvku rovných x
1
>>> s.sort()          # seradi seznam
>>> s
[1, 4, 10, 12]
>>> s.reverse()       # obrat seznam
>>> s
[12, 10, 4, 1]
```

Seznamy v Pythonu – generátorová notace

- Specialita Pythonu

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

```
>>> print(s)
```

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

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

```
>>> print(s)
```

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

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

```
>>> print(s)
```

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

I. Abstraktní datové typy

Seznam

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

```
>>> x = [1, 2, 3, 1, 5, 2]
>>> s = set(x)      # vytvoři množinu ze seznamu
>>> s
{1, 2, 3, 5}
>>> len(s)         # počet pruku množiny s
4
>>> s.add(10)      # pridani pruku do množiny
>>> s
{1, 2, 3, 5, 10}
>>> s.remove(2)   # odebrani pruku z množiny
>>> s
{1, 3, 5, 10}
>>> 10 in s       # test, zda množina obsahuje x
True
```

Množina v Pythonu

```
>>> a = set("abracadabra")
>>> b = set("engineering")
>>> a
{'a', 'r', 'd', 'c', 'b'}
>>> b
{'i', 'g', 'r', 'n', 'e'}
>>> a | b      # ekv. a.union(b), sjednoceni
{'i', 'g', 'a', 'r', 'n', 'e', 'd', 'c', 'b'}
>>> a & b      # ekv. a.intersection(b), prunik
{'r'}
>>> a - b      # ekv. a.difference(b), rozdil
{'c', 'b', 'a', 'd'}
>>> a ^ b      # ekv. a.symmetric_difference(b)
{'i', 'g', 'a', 'n', 'e', 'd', 'b', 'c'}
```

Část II

Zásobník

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 – implementace pomocí pole

```
>>> class Stack:
...     def __init__(self):
...         self.items = []
...
...     def size(self):
...         return len(self.items)
...     def is_empty(self):
...         return self.size()==0
...     def push(self, item):
...         self.items+= [item]
...     def pop(self):
...         return self.items.pop()
...     def peek(self):
...         return self.items[-1]
... 
```

Zásobník – příklad použití

```
>>> s = Stack()
>>> s.push(1)
>>> s.push(2)
>>> s.pop()
2
>>> s.push(10)
>>> s.pop()
10
>>> s.is_empty()
False
>>> s.pop()
1
>>> s.is_empty()
True
```

II. Zásobník

P8.1 Kontrola uzávorkování

P8.2 Postfixová notace


```
>>> def par_checker(input):
...     L = "([{" # otevírací závorky
...     R = ")]}" # uzavírací závorky (stejně pořadí)
...     s = Stack()
...
...     for c in input:
...         if c in L:
...             s.push(c)
...         else:
...             for i in range(len(R)):
...                 if c == R[i]:
...                     if s.is_empty() or s.pop() != L[i]:
...                         return False
...     return s.is_empty()
...
...
```

```
>>> 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
```

```
>>> par_checker("ahoj+(svete-(X+Y{}))")
```

```
False
```

II. Zásobník

P8.1 Kontrola uzávorkování

P8.2 Postfixová notace

P8.2 Postfixová notace

- Klasická notace je **infixová** – operátor je mezi operandy
- Někdy je výhodnější **postfixová** notace – operátor je 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,...

P8.1 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
$12 / 4$	$12\ 4\ /$
$3 * 4 - 2$	$3\ 4\ * 2\ -$
$3 * (4 - 2)$	$3\ 4\ 2\ -\ *$
$(62-32)*5/9$	$62\ 32\ -\ 5\ * 9\ /$

```
>>> def eval_postfix(s):
...     stack=Stack()
...     for x in s.split(): # rozděl 's' dle mezer
...         if x=='+':
...             stack.push(stack.pop()+stack.pop())
...         elif x=='-':
...             stack.push(-stack.pop()+stack.pop())
...         elif x=='*':
...             stack.push(stack.pop()*stack.pop())
...         elif x=='/':
...             second=stack.pop()
...             stack.push(stack.pop()/second)
...         else: # 'x' je číslo
...             stack.push(float(x))
...     return stack.pop()
... 
```

```
>>> 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
```

```
10.0
```

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

```
6.0
```

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

1. **Číslo** okopíruj na výstup.
2. **Operátor** ulož do zásobníku operátorů.
 - (a) Je-li v zásobníku operátor s vyšší prioritou, 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 zahod.
5. Nakonec přesuň ze zásobníku na výstup zbývající operátory.

`stack_examples.py`

funkce `infix_to_postfix`


```
>>> def infix_to_postfix(s):
...
...     result = "" # výstupní řetězec
...     op = Stack() # zásobník operátorů
...     i = 0      # index 's'
...
...     while i<len(s):
...         if s[i] in "0123456789":
...             while i<len(s) and s[i] in "0123456789":
...                 result+=s[i]
...                 i+=1
...             result+=" "
...             continue
```

```
...     if s[i]=='(':
...         op.push(s[i])
...     elif s[i]==')':
...         top=op.pop()
...         while top!='(':
...             result+=top+" "
...             top=op.pop()
...     else: # s[i] is +,-,*,/
...         while not op.is_empty() and not higher_prec(s[i],op.peek()):
...             result+=op.pop()+" "
...         op.push(s[i])
...     i+=1
... while not op.is_empty():
...     result+=op.pop()+" "
... return result
...
```

```
>>> def higher_prec(a,b):
...     """ operátor 'a' má vyšší prioritu než 'b' (na vrcholu zásobníku) """
...     return ((a in "*/") and (b in "+-")) or (b=="(")
...
>>> print(infix_to_postfix("32+4"))
3 2 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"))
6 2 3 2 - 5 * 9 /
```

P8.2 Vyhodnocování infixových výrazů

- Výraz převedeme na postfixový a vyhodnotíme.

```
>>> def eval_infix(s):  
...     return eval_postfix(infix_to_postfix(s))  
...
```

```
>>> print(eval_infix("32+4"))  
36.0
```

```
>>> print(eval_infix("3*4-2"))  
10.0
```

```
>>> print(eval_infix("3*(4-2)"))  
6.0
```

```
>>> print(eval_infix("(62-32)*5/9"))  
16.666666666666668
```

Část III

Fronta

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 – 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*)

- Implementace pomocí seznamů snadná, ale neefektivní
 - přidávání a odebrání 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]`


```
>>> from collections import deque
>>> q = deque(["Eric", "John", "Michael"])
>>> q.append("Terry") # přichází Terry
>>> q
deque(['Eric', 'John', 'Michael', 'Terry'])
>>> q.append("Graham") # přichází Graham
>>> q
deque(['Eric', 'John', 'Michael', 'Terry', 'Graham'])
>>> q.popleft()      # odchází Eric
'Eric'
>>> q.popleft()      # odchází John
'John'
>>> q
deque(['Michael', 'Terry', 'Graham'])
```

- Implementace pomocí seznamu, s pomalým vkládáním
- Složitost vkládání je $O(n)$

```
>>> class Queue1:
...     def __init__(self):
...         self.items = []
...     def is_empty(self):
...         return self.items == []
...     def enqueue(self, item):
...         self.items.insert(0,item)
...     def dequeue(self):
...         return self.items.pop()
...     def size(self):
...         return len(self.items)
... 
```

```
>>> q = Queue1()
>>> q.enqueue(1)
>>> q.enqueue(2)
>>> print(q.dequeue())
1
>>> q.enqueue(10)
>>> print(q.dequeue())
2
>>> print(q.is_empty())
False
>>> print(q.dequeue())
10
>>> print(q.is_empty())
True
```

- 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 $n/2 + n/4 + \dots \sim n$ prvků
→ amortizovaná složitost odebrání prvku je $O(1)$.
- Nevýhoda – neefektivní využití paměti.

arrayqueue.py

```
>>> class Queue2:
...     def __init__(self):
...         self.front = 0    # index prvního prvku
...         self.items = []
...     def is_empty(self):
...         return len(self.items) == self.front
...     def enqueue(self, item):
...         self.items+= [item]
...     def dequeue(s):
...         el = s.items[s.front];
...         s.front+=1
...         if s.front >= 1024 and s.front >= len(s.items)//2:
...             s.items = s.items[s.front:];
...             s.front = 0
...         return el
```

```
...     def size(self):
...         return len(self.items)
...
>>> q = Queue2()
>>> q.enqueue(1)
>>> q.enqueue(2)
>>> print(q.dequeue())
1
>>> print(q.is_empty())
False
>>> print(q.dequeue())
2
>>> print(q.is_empty())
True
```

- 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 $O(1)$

```
>>> class Queue3:
...     def __init__(self):
...         self.inp = Stack()
...         self.out = Stack()
...     def is_empty(self):
...         return self.size()==0
...     def enqueue(self, item):
...         self.inp.push(item)
...     def dequeue(self):
...         if self.out.is_empty():
...             while not self.inp.is_empty():
...                 self.out.push(self.inp.pop())
...         return self.out.pop()
...     def size(self):
...         return self.inp.size() + self.out.size()
... 
```


P8.3 Rozpočítávání

- Děti v kruhu, rozpočítávací 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.

```
>>> def rozpocitej(jmena, m):
...     p = Queue1()
...     for j in jmena: # uloz jmena do fronty
...         p.enqueue(j)
...     while p.size() > 1:
...         for i in range(m-1):
...             p.enqueue(p.dequeue())
...         print("Vypadl(a): ", p.dequeue())
...     return p.dequeue() # vrať vítěze
... 
```

```
>>> v = rozpocitej(["Adam", "Bara", "Cyril", "David", "Emma",
... "Franta", "Gabina"], 3)
Vypadl(a): Cyril
Vypadl(a): Franta
Vypadl(a): Bara
Vypadl(a): Gabina
Vypadl(a): Emma
Vypadl(a): Adam
>>> print("Vyhrál(a): ", v)
Vyhrál(a): David
```

Část IV

Spojový seznam

Uchování prvků

Základním datovým typem pro uchování množiny prvků (proměnných/struktur) je **pole**, v Pythonu nejčastěji realizované **seznamem**.

- Jedná se o kolekci položek (proměnných) stejného nebo rozdílného typu
- + Umožňuje jednoduchý přístup k položkám indexací prvku
- Některé operace mohou být časově náročné
 - Přidávání prvků na začátek seznamu
 - Vkládání nových prvků
 - Změna velikosti
- V případě řazení pole přesouváme položky

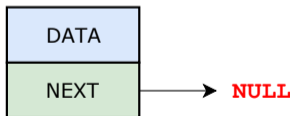
Spojové seznamy

- Datová struktura realizující seznam dynamické délky
- Každý prvek (uzel, node) seznamu obsahuje
 - Datovou část (hodnota proměnné / objekt / ukazatel na data)
 - Odkaz (ukazatel) na další prvek v seznamu

NULL nebo vhodnou zarážku v případě posledního prvku seznamu.

- První prvek seznamu se zpravidla označuje jako **head** nebo **start**.

Realizujeme jej jako ukazatel odkazující na první prvek seznamu.



Základní operace se spojovým seznamem

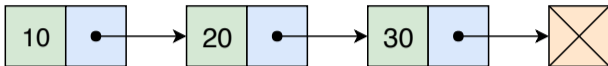
- Vložení prvku
 - Předchozí prvek odkazuje na nový prvek
 - Nový prvek může odkazovat na předchozí prvek, který na něj odkazuje
- Odebrání prvku
 - Předchozí prvek aktualizuje hodnotu odkazu na následující prvek
 - Předchozí prvek tak nově odkazuje na následující hodnotu, na kterou odkazoval odebíraný prvek

Obousměrný spojový seznam.

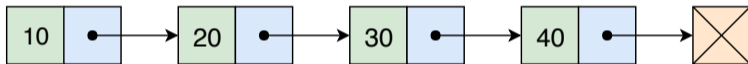
Základní implementací spojového seznamu je tzv. jednosměrný spojový seznam

Jednosměrný spojový seznam

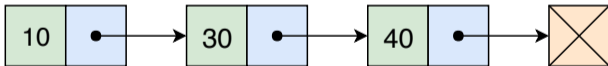
- Příklad spojového seznamu pro uložení číselných hodnot



- Přidání prvku 40 na konec seznamu



- Odebrání prvku 20 ze seznamu



1. Nejdříve sekvenčně najdeme prvek s hodnotou 30

Prvek, na který odkazuje NEXT odebíraného prvku.

2. Následně vyjmeme a napojíme prvek 10 na prvek 30

Hodnotu NEXT prvku 10 nastavíme na adresu prvku 30.

Reprezentace uzlu

```
>>> class Node:    # uzel
...     def __init__(self,data):
...         self.data = data
...         self.next = None # odkaz na dalsi uzel
... 
```

```
>>> class ListStack:    # seznam
...     def __init__(self):
...         self.head = None
...     def is_empty(self):
...         return self.head is None
...     def push(self,item):
...         node=Node(item)
...         node.next=self.head
...         self.head=node
...     def pop(self):
...         item=self.head.data
...         self.head=self.head.next
...         return item
...     def peek(self):
...         return self.head.data
... 
```

```
>>> s = ListStack()
>>> s.push(1)
>>> s.push(2)
>>> print(s.pop())
2
>>> print(s.pop())
1
>>> s.push(10)
>>> print(s.is_empty())
False
>>> print(s.pop())
10
>>> print(s.is_empty())
True
```

```
>>> class ListQueue(ListStack):
...     def __init__(self):
...         self.head = None
...         self.last = None # last item
...         self.count = 0
...     def pop(self):          # odeber ze začátku
...         item=self.head.data
...         self.head=self.head.next
...         if self.head is None:
...             self.last=None
...             self.count-=1
...         return item
...     def dequeue(self):
...         return self.pop()
```

```
...     def push(self,item):    # přidej na začátek
...         node=Node(item)
...         node.next=self.head
...         if self.head is None:
...             self.last=node
...             self.head=node
...             self.count+=1
...     def enqueue(self,item): # přidej na konec
...         node=Node(item)
...         if self.head is None: # seznam je prázdný
...             self.head=node
...             self.last=node
...         else:
...             self.last.next=node
...             self.last=node
...             self.count+=1
```

```
>>> q = ListQueue()
>>> q.enqueue(1)
>>> q.enqueue(2)
>>> print(q.dequeue())
1
>>> q.enqueue(10)
>>> print(q.dequeue())
2
>>> print(q.is_empty())
False
>>> print(q.dequeue())
10
>>> print(q.is_empty())
True
```

Procházení prvků fronty

```
>>> def iter(self, f):
...     # spusti f(x) pro vsechna 'x' ve fronte
...     node = self.head
...     while node is not None:
...         f(node.data)
...         node = node.next
...
>>> def reduce(self, f, acc):
...     # spusti acc=f(x, acc) pro vsechna 'x' ve fronte
...     node = self.head
...     while node is not None:
...         acc = f(node.data, acc)
...         node = node.next
...     return acc
...
```

Konverze na pole a z pole

```
>>> def to_array(self):
...     a = []
...     self.iter(lambda x: a.append(x))
...     return a
...
>>> def array_to_queue(a):
...     q = ListQueue()
...     for x in a:
...         q.enqueue(x)
...     return q
...
```

linkedlistqueue.py

Příklady

```
>>> ListQueue.array_to_queue = array_to_queue
```

```
>>> ListQueue.reduce = reduce
```

```
>>> ListQueue.iter = iter
```

```
>>> ListQueue.to_array = to_array
```

```
>>> q=array_to_queue([4,2,7,3])
```

```
>>> print(q.reduce(max,0))
```

```
7
```

```
>>> print(q.reduce(lambda x,acc: acc+x,0))
```

```
16
```

```
>>> print(q.to_array())
```

```
[4, 2, 7, 3]
```

Vyhledávání v seznamu

```
>>> def contains(self,x): # vraci True, pokud seznam obsahuje 'x'
...     node=self.head
...     while node is not None:
...         if node.data==x:
...             return True
...         node=node.next
...     return False
...
>>> ListQueue.contains = contains
>>> q=array_to_queue([3, 52, 69, 17, 19])
>>> print(q.contains(17))
True
>>> print(q.contains(20))
False
```

Mazání v seznamu

```
>>> def remove(self, x): # vyradi prvek x, pokud je v seznamu
...     node = self.head
...     prev = None
...     while node is not None:
...         if node.data == x:
...             if prev is None:
...                 self.head = node.next
...                 if self.head is None:
...                     self.last = None
...             else:
...                 prev.next = node.next
...     prev = node
...     node = node.next
... 
```

Uspořádaný spojový seznam – řazení

- Seznam budeme udržovat seřazený.
- Seznam lze procházet jen odpředu (od `self.head`).
- Vkládání (*insert*) dopředu je rychlejší.
- Prvky mohou často přicházet srovnané vzestupně (*insertion sort*)
- Seznam budeme řadit sestupně, aby větší prvky mohly zůstat vpředu.

Uspořádaný spojový seznam – vkládání

```
>>> class OrderedList(ListQueue):
...     def insert(self,x):
...         newnode = Node(x); prev = None; node = self.head
...         while node is not None and x<node.data:
...             prev = node; node = node.next
...         if node is None: # newnode patří na konec
...             if self.head is None: # seznam je prázdný
...                 self.head=newnode
...             else:
...                 self.last.next=newnode
...                 self.last=newnode
...         else:
...             if prev is None: # newnode patří na začátek
...                 self.head=newnode
...             else: # newnode patří mezi prev a node
...                 prev.next=newnode
...                 newnode.next=node
...         self.count+=1
```

Řazení vkládáním a spojové seznamy

```
>>> def insertion_sort_linkedlist(a):
...     """ serad vzestupne pole 'a' v miste """
...     q = OrderedList()
...     for x in a:
...         q.insert(x)
...     for i in range(len(a)-1, -1, -1):
...         a[i] = q.pop() # from the highest value
...     print(a)
...
>>> insertion_sort_linkedlist([3, 2, 6, 7, 4, 8, 10])
[2, 3, 4, 6, 7, 8, 10]
```

`insertion_sort_linkedlist.py`

Spojování seznamů v konstantním čase

```
>>> def concatenate(self,l):
...     """ destruktivne pridej seznam 'l' na konec """
...     if l.last is None:
...         return
...     if self.last is None:
...         self.head=l.head
...     else:
...         self.last.next=l.head
...     self.last=l.last
...     self.count+=l.count
...     l.head=None # smaž list 'l'
...     l.last=None
...     l.count=0
... 
```

linkedlistqueue.py.

Spojování seznamů – příklad

```
>>> ListQueue.concatenate = concatenate
```

```
>>> q=array_to_queue([1, 2, 3])
```

```
>>> r=array_to_queue([4, 5])
```

```
>>> q.concatenate(r)
```

```
>>> print(q.to_array())
```

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

`linkedlist_examples.py`

Oboustranná fronta

Lineární datová struktura kombinující frontu a zásobník.

operace	zásob.	fronta	oboustranná fronta	spoj.sez.	
				jedn.	dvoj.
přidej na začátek	$O(1)$		$O(1)$	$O(1)$	$O(1)$
přidej na konec		$O(1)$	$O(1)$	$O(1)$	$O(1)$
odeber ze začátku	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
odeber z konce			$O(1)$	$O(n)^\#$	$O(1)$
iterace vpřed				ano	ano
iterace vzad					ano

`začátek = self.head`

`konec = self.last`

$^\# O(1)$, pokud máme odkaz `i` na předchozí uzel.

Aplikace oboustranné fronty

- Zásobník s omezenou délkou
 - Seznam navštívených stránek v prohlížeči
 - Undo/redo operace v textovém či grafickém editoru
- Rozvrhování pro více procesorů — volné procesory mohou 'ukrást' proces jiným.
- Nalezení maxima všech souvislých podsekvencí dané délky.

Spojový seznamu – shrnutí

- Podporuje mnoho operací v čase $O(1)$...
- ...za cenu větších časových a paměťových nároků (konstantní faktor)
- Pomocí spojového seznamu můžeme implementovat zásobník i frontu.
- *Dvojitě zřetězený* spojový seznam umí rychle více operací (iterace vzad, vypuštění prvku uprostřed) za cenu opět větších časových a paměťových nároků (konstantní faktor).