

Záznam, zásobník a fronta

Jan Kybic

<http://cmp.felk.cvut.cz/~kybic>
kybic@fel.cvut.cz

2016–2019



Záznam

Zásobník

Fronta

Záznam

(Record)

Záznam (obecně)

- ▶ strukturovaný/složený datový typ
- ▶ obsahuje položky (*fields*) / prvky (*elements*)/ členy
- ▶ položek je (obvykle) pevný počet, mají každá svůj typ
- ▶ položky jsou identifikované jménem
- ▶ typ záznamu má své jméno

Záznam

(Record)

Záznam (obecně)

- ▶ strukturovaný/složený datový typ
- ▶ obsahuje položky (*fields*) / prvky (*elements*)/ členy
- ▶ položek je (obvykle) pevný počet, mají každá svůj typ
- ▶ položky jsou identifikované jménem
- ▶ typ záznamu má své jméno

Příklady

- ▶ Datum = rok, měsíc, den
- ▶ Osoba = jméno, příjmení, datum narození
- ▶ Adresa = jméno ulice, číslo popisné, město, PSČ
- ▶ Bod = x , y
- ▶ Kruh = střed, poloměr

Abstrakce

- ▶ **funkční abstrakce** (*function abstraction*)
 - ▶ Klient ví, co funkce dělá. (*rozhraní, interface*)
 - ▶ Klient nemusí vědět, jak je funkce implementována.
- ▶ **datová abstrakce** (*data abstraction*)
 - ▶ Klient ví, co datový typ představuje, jak ho vytvořit a jaké operace podporuje.
 - ▶ Klient nemusí vědět, jaká je vnitřní struktura.
- ▶ Klient musí znát rozhraní (*interface*).
- ▶ Implementace je skrytá, lze ji změnit.
- ▶ Implementace je rozdělena mezi klientský kód a knihovny (např. ve formě modulů).
- ▶ Znovupoužitelnost kódu, možnost nezávislého vývoje.
- ▶ Zjednodušení psaní klientského kódu.

Záznam v Pythonu

V Pythonu je to jinak

V Pythonu záznamy nejsou...

Záznam v Pythonu

V Pythonu je to jinak

V Pythonu záznamy nejsou. . . v Pythonu jsou (dynamické) objekty

objekt = *záznam* + *metody*

Objekt (*object*)

- ▶ Obsahuje datové položky / atributy / proměnné (*fields*, *attributes*, *instance variables*)
- ▶ Může obsahovat *metody* = funkce pracující s datovými položkami
- ▶ Strukturu definuje **třída** (*class*), objekty jsou instance třídy.
- ▶ Základ *objektově orientovaného programování* (*object oriented programming* — *OOP*).

Objekty v Pythonu

Definice třídy (*class*):

```
class Osoba:  
    pass # prázdná třída
```


Objekty v Pythonu

Definice třídy (*class*):

```
class Osoba:  
    pass # prázdná třída
```

Vytvoření objektu (*object instantiation*):

```
o=Osoba()
```

Objekty v Pythonu

Definice třídy (*class*):

```
class Osoba:  
    pass # prázdná třída
```

Vytvoření objektu (*object instantiation*):

```
o=Osoba()
```

Přidání a použití datových položek (atributů):

```
o.jmeno="Karel"  
o.prijmeni="Novák"
```

```
print(o.jmeno+" "+o.prijmeni)
```

```
Karel Novák
```

Objekty v Pythonu

Definice třídy (*class*):

```
class Osoba:  
    pass # prázdná třída
```

Vytvoření objektu (*object instantiation*):

```
o=Osoba()
```

Přidání a použití datových položek (atributů):

```
o.jmeno="Karel"  
o.prijmeni="Novák"
```

```
print(o.jmeno+" "+o.prijmeni)
```

Karel Novák

- ▶ V Pythonu jsou datové položky přidávané *dynamicky*.
- ▶ K datovým položkám přistupujeme pomocí *tečkové notace*.

Metody

```
class Osoba:  
    def print(self):  
        print(self.jmeno+" "+self.prijmeni)
```

```
o.print()
```

Karel Novák

Metody

```
class Osoba:  
    def print(self):  
        print(self.jmeno+" "+self.prijmeni)
```

`o.print()`

Karel Novák

- ▶ Metody jsou funkce definované uvnitř třídy.
- ▶ Pracují s konkrétním objektem. Mají přístup k proměnným objektu pomocí prvního parametru.
- ▶ Voláme je na objekt tečkovou notací.

Vytvoření objektu

Objekt s konstruktorem (*constructor*):

```
class Osoba:
    def __init__(self, jmeno, prijmeni):
        self.jmeno=jmeno
        self.prijmeni=prijmeni

    def print(self):
        print(self.jmeno+" "+self.prijmeni)
```

Vytvoření objektu

Objekt s konstruktorem (*constructor*):

```
class Osoba:
    def __init__(self, jmeno, prijmeni):
        self.jmeno=jmeno
        self.prijmeni=prijmeni

    def print(self):
        print(self.jmeno+" "+self.prijmeni)
```

```
o=Osoba("Karel", "Novák")
o.print()
```

Karel Novák

Vytvoření objektu

Objekt s konstruktorem (*constructor*):

```
class Osoba:
    def __init__(self, jmeno, prijmeni):
        self.jmeno=jmeno
        self.prijmeni=prijmeni

    def print(self):
        print(self.jmeno+" "+self.prijmeni)
```

```
o=Osoba("Karel", "Novák")
o.print()
```

Karel Novák

- ▶ Konstruktor má speciální jméno `__init__`.
- ▶ Je volán při vytvoření objektu.

Příklad: Body ve 2D

```
class Point:  
    """ 2D point with attributes 'x' and 'y' """  
    def __init__(self,x,y):  
        self.x=x  
        self.y=y
```

Příklad: Body ve 2D

```
class Point:
    """ 2D point with attributes 'x' and 'y' """
    def __init__(self,x,y):
        self.x=x
        self.y=y
```

```
r=Point(10,20)
s=Point(13,24)
print("r.x=",r.x)
```

```
r.x= 10
```

```
s.y=20
print("s.y=",s.y)
```

```
s.y= 20
```

Příklad: Body ve 2D (2)

Funkce s argumenty typu bod (Point):

```
def distance(r,s):  
    """ Vypočítá vzdálenost dvou bodů 'r', 's' typu Point """  
    return math.sqrt((r.x-s.x)**2+(r.y-s.y)**2)
```

Příklad: Body ve 2D (2)

Funkce s argumenty typu bod (Point):

```
def distance(r,s):  
    """ Vypočítá vzdálenost dvou bodů 'r','s' typu Point """  
    return math.sqrt((r.x-s.x)**2+(r.y-s.y)**2)
```

```
r=Point(10,20)  
s=Point(13,24)  
print(distance(r,s))
```

5.0

Záznam

Zásobník

Fronta

Zásobník

(Stack)

- ▶ 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 (*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*)
 - ▶ zjištění počtu položek na zásobníku (*size*)

Zásobník — příklad použití

```
from stack import Stack

>>> s=Stack()
>>> s.push(1)
>>> s.push(2)
>>> s.push(3)
>>> print(s.pop())
3
>>> print(s.pop())
2
>>> s.push(10)
>>> print(s.pop())
10
>>> print(s.is_empty())
False
>>> print(s.pop())
1
>>> print(s.is_empty())
True
```

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]
```


Příklad: Převod do jiné číselné soustavy

Zásobník často nahrazuje explicitní rekurzi:

```
def to_str(n,base):
    cislice = "0123456789ABCDEF"
    assert(n>=0)
    stack=Stack()
    while True:
        stack.push(n % base)
        n//=base
        if n==0:
            break
    result=""
    while not stack.is_empty():
        result+=cislice[stack.pop()]
    return result

print(to_str(67,2))
```

1000011

Quicksort rekurzivní

```
def quick_sort(a):  
    """ Setřídí pole 'a' na místě """  
    quick_sort_helper(a,0,len(a)-1)  
  
def quick_sort_helper(a,first,last):  
    """ setřídí podpole a[first]..a[last] """  
    if first < last:  
        m = partition(a,first,last)  
        quick_sort_helper(a,first,m-1)  
        quick_sort_helper(a,m+1,last)
```

Quicksort nerekurzivní

```
def quick_sort(a):  
    """ Setřídí pole 'a' na místě """  
    stack=Stack()  
    stack.push((0, len(a)-1)) # první a poslední index intervalu  
    while not stack.is_empty():  
        (first, last)=stack.pop()  
        m = partition(a, first, last)  
        if first < m-1:  
            stack.push((first, m-1))  
        if m+1 < last:  
            stack.push((m+1, last))
```

Quicksort nerekurzivní

```
def quick_sort(a):  
    """ Setřídí pole 'a' na místě """  
    stack=Stack()  
    stack.push((0,len(a)-1)) # první a poslední index intervalu  
    while not stack.is_empty():  
        (first,last)=stack.pop()  
        m = partition(a,first,last)  
        if first<m-1:  
            stack.push((first,m-1))  
        if m+1<last:  
            stack.push((m+1,last))
```

```
a=[random.randrange(100) for i in range(10)]  
quick_sort(a)  
print(a)
```

[11, 29, 49, 53, 77, 86, 87, 90, 92, 99]

Příklad: kontrola uzávorkování

Vrať `True` pokud je řetězec s správně uzávorkovaný.

```
def par_checker(s):
    lparens="([{ " # otevírací závorky
    rparens=")]}" # uzavírací závorky (ve stejném pořadí)
    stack=Stack()
    for c in s:
        if c in lparens:
            stack.push(c)
        else:
            for i in range(len(rparens)):
                if c==rparens[i]:
                    if stack.is_empty() or stack.pop()!=lparens[i]:
                        return False
    return stack.is_empty()
```

Soubor `stack_examples.py`

Kontrola uzávorkování — 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

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

False

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 /

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

Vyhodnoť výraz `s` v postfixové notaci (řetězec, oddělení mezerami)

```
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()
```

Vyhodnocení postfixového výrazu

Vyhodnoť výraz `s` v postfixové notaci (řetězec, oddělení mezerami)

```
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()
```



Python vyhodnocuje výrazy zleva doprava.

Vyhodnocení postfixového výrazu (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
```

10.0

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

6.0

Vyhodnocení postfixového výrazu (3)

Cvičení:

- ▶ Odstraňte nutnost oddělování mezerami.
- ▶ Přidejte chybová hlášení.
- ▶ Přidejte operace (**sin**, **cos**, $\sqrt{\quad}$, ...)

Převod infixového na postfixový výraz

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 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 zahod'.
5. Nakonec přesuň ze zásobníku na výstup zbývající operátory.

Soubor `stack_examples.py`, funkce `infix_to_postfix`.

Převod infixového na postfixový výraz — příklady

```
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 /
```

Vyhodnocování infixových výrazů

- ▶ Výraz převedeme na postfixový a vyhodnotíme.
- ▶ Jde to i rovnou.

```
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

Převod infixového na postfixový výraz (2)

Cvičení:

- ▶ Dovolte mezery ve vstupním výrazu.
- ▶ Dovolte ve vstupním výrazu různé typy závorek — jejich párování musí odpovídat.
- ▶ Implementujte vyhodnocování infixových výrazů bez převodu na jinou formu.
- ▶ Implementujte vyhodnocování prefixových výrazů.
- ▶ Implementujte převodník infixových na prefixové výrazy.

Záznam

Zásobník

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*)

Zásobník

(Stack)

- ▶ 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 (*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*)
 - ▶ zjištění počtu položek na zásobníku (*size*)

Fronta — příklad použití

```
from slowqueue import Queue

>>> q=Queue()
>>> q.enqueue(1)
>>> q.enqueue(2)
>>> q.enqueue(3)
>>> print(q.dequeue())
1
>>> print(q.dequeue())
2
>>> q.enqueue(10)
>>> print(q.dequeue())
3
>>> print(q.is_empty())
False
>>> print(q.dequeue())
10
>>> print(q.is_empty())
True
```

Zásobník — příklad použití

```
from stack import Stack

>>> s=Stack()
>>> s.push(1)
>>> s.push(2)
>>> s.push(3)
>>> print(s.pop())
3
>>> print(s.pop())
2
>>> s.push(10)
>>> print(s.pop())
10
>>> print(s.is_empty())
False
>>> print(s.pop())
1
>>> print(s.is_empty())
True
```

- ▶ 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 polí

```
class Queue:
    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)
```

Soubor `slowqueue.py`

Fronta — implementace pomocí polí

```
class Queue:
    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)
```

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

Soubor `slowqueue.py`

Fronta — pole s realokací

- ▶ Prvky ukládáme do pole.
- ▶ Vkládání na konec pole ($O(1)$).
- ▶ 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 \rightarrow amortizovaná složitost odebrání prvku je $O(1)$.
- ▶ Nevýhoda — neefektivní využití paměti.

Fronta — pole s realokací (2)

```
class Queue:
    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(self):
        el=self.items[self.front]
        self.front+=1
        if self.front >= 1024 and self.front>=len(self.items)//2:
            self.items=self.items[self.front:]
            self.front=0
        return el
    def size(self):
        return len(self.items)
```

Fronta — dva zásobníky

Donald Knuth

- ▶ 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)

```
class Queue:
    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()
```

Soubor knuthqueue.py

Fronta — dva zásobníky (2)

```
class Queue:
    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()
```

Soubor knuthqueue.py

Cvičení: Porovnejte rychlost jednotlivých implementací.

Příklad: 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.

Příklad: Rozpočítávání

- ▶ Děti v kruhu, rozpočítávací hadlo 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.

```
v=rozpocitej(["Adam","Bára","Cyril","David","Emma",  
             "Franta","Gábina"],3)  
print("Vyhrál(a): ",v)
```

Soubor rozpocitavani.py

Příklad: Rozpočítávání (2)

```
from knuthqueue import Queue

def rozpocitej(jmena,m):
    q=Queue()
    for j in jmena:    # ulož jména do fronty
        q.enqueue(j)
    while q.size()>1:
        for i in range(m-1):
            q.enqueue(q.dequeue())
        print("Vypadl(a): ",q.dequeue())

    return q.dequeue()    # vrať vítěze

if __name__=="__main__":
    v=rozpocitej(["Adam","Bára","Cyril",
                 "David","Emma","Franta","Gábina"],3)
    print("Vyhrál(a): ",v)
```

```
v=rozpocitej(["Adam","Bára","Cyril","David","Emma",  
             "Franta","Gábina"],3)  
print("Vyhrál(a): ",v)
```

```
Terminal> python3 rozpocitavani.py
```

```
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.

Soubor `tiskova_fronta.py`

Tisková fronta – implementace

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

```
from knuthqueue import Queue
import random

class Uloha:
    def __init__(self, t, s):
        self.time=t    # čas vytvoření
        self.stran=s  # počet stran

class Tiskarna:
    def __init__(self):
        self.uloha=None
        self.zbyvajici_cas=0
```

Tisková fronta – implementace (2)

```
def simuluj(num_people,prob_second,max_pages,seconds_per_page,
            simulation_time):
    """ Nasimuluje chování tiskové fronty.
        "num_people" - počet lidí
        "prob_second" - pravdř. vytvoření úlohy v dané vteřině
        "max_pages" - maximální počet stran v úloze
        "seconds_per_page" - počet sekund na stránku
        "simulation_time" - délka simulace v sekundách """
    q=Queue() # tisková fronta
    t=Tiskarna() # stav tiskárny
    casy_cekani=[] # délky čekání na vytisknutí v sekundách
    for i in range(simulation_time):
        simuluj_lidi(num_people,prob_second,max_pages,q,i)
        simuluj_tiskarnu(seconds_per_page,q,t,i,casy_cekani)
    avg_time=sum(casy_cekani)/len(casy_cekani)
    print("Průměrná doba čekání %5.2fs." % avg_time)
    return avg_time
```

Tisková fronta – implementace (3)

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

Tisková fronta – implementace (4)

```
def simuluj_tiskarnu(seconds_per_page,q,t,i,casy_cekani):
    if t.uloha!=None:          # tiskárna tiskne
        t.zbyvajici_cas-=1
        if t.zbyvajici_cas<=0: # hotovo
            print("Čas %d, tisk %d stran/y hotov, čekání %5.1fs."
                  % (i,t.uloha.stran,i-t.uloha.time))
            casy_cekani+=[i-t.uloha.time]
            t.uloha=None
    if t.uloha==None:         # tiskárna je volná
        if not q.is_empty():  # ve frontě je úloha
            t.uloha=q.dequeue()
            print("Čas %d, začínáme tisknout úlohu mající %d stran."
                  % (i,t.uloha.stran))
            t.zbyvajici_cas=t.uloha.stran*seconds_per_page
```

Tisková fronta – příklad

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


Tisková fronta – příklad

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

Čas 104, požadavek na tisk 4 stran.

Čas 104, začínáme tisknout úlohu mající 4 stran.

Čas 152, tisk 4 stran/y hotov, čekání 48.0s.

Čas 1201, požadavek na tisk 8 stran.

Čas 1201, začínáme tisknout úlohu mající 8 stran.

Čas 1297, tisk 8 stran/y hotov, čekání 96.0s.

Čas 2185, požadavek na tisk 6 stran.

Čas 2185, začínáme tisknout úlohu mající 6 stran.

Čas 2188, požadavek na tisk 7 stran.

Čas 2257, tisk 6 stran/y hotov, čekání 72.0s.

Čas 2257, začínáme tisknout úlohu mající 7 stran.

Čas 2341, tisk 7 stran/y hotov, čekání 153.0s.

...

Průměrná doba čekání 73.30s.

Cvičení:

- ▶ Zjistěte závislost doby čekání na počtu lidí, vykreslete graf.
- ▶ Vypočítejte a nakreslete histogram doby čekání.
- ▶ Přidejte identifikátor úlohy.
- ▶ Na konci simulace vypište, kolik úloh čeká ve frontě.
- ▶ Vykreslete statistiku délky fronty.
- ▶ Implementujte více tiskáren různé rychlosti. Vykreslete graf závislosti doby čekání na počtu tiskáren.

Nedestruktivní čtení

(peek)

```
class Queue:
    def __init__(self):
        self.inp = Stack()
        self.out = Stack()
    def enqueue(self, item):
        self.inp.push(item)
    def peek(self):
        self.check_out()
        return self.out.peek()
    def dequeue(self):
        self.check_out()
        return self.out.pop()
    def check_out(self):
        if self.out.is_empty():
            while not self.inp.is_empty():
                self.out.push(self.inp.pop())
    ...
```

Merge sort a fronty

- ▶ *Merge sort* přistupuje pouze na začátek nebo na konec pole.
- ▶ → může místo polí pracovat s frontami.
- ▶ Asymptotická časová složitost zůstává stejná.
- ▶ Mírně větší “režijní” náročnost.
- ▶ Použití
 - ▶ Řazení externích souborů (na disku, na pásce).
 - ▶ Řazení dat větších než operační paměť.

Merge sort — implementace

```
def merge_sort_queue(q):  
    """ Setřídí frontu 'q' pomocí merge sort """  
    if q.size() <= 1:    # triviální případ  
        return q  
    m = q.size() // 2    # počet prvků po rozdělení  
    left = Queue()  
    for i in range(m):  
        left.enqueue(q.dequeue())  
    left = merge_sort_queue(left)  
    right = merge_sort_queue(q)  
    return join_queues(left, right)
```

Merge sort — implementace (2)

```
def join_queues(left, right):  
    """ Dvě vzestupně seříděné fronty spojí do jedné """  
    result=Queue() # výsledek  
  
    while not left.is_empty() and not right.is_empty():  
        if left.peek()<right.peek():  
            result.enqueue(left.dequeue())  
        else:  
            result.enqueue(right.dequeue())  
  
    while not left.is_empty(): # doplnit zbytky  
        result.enqueue(left.dequeue())  
    while not right.is_empty():  
        result.enqueue(right.dequeue())  
  
    return result
```

Merge sort — implementace (3)

```
def merge_sort(a):  
    """ Setřídí pole 'a' pomocí merge sort """  
    q=Queue()  
    for x in a:  
        q.enqueue(x)  
    q=merge_sort_queue(q)  
    for i in range(q.size()):  
        a[i]=q.dequeue()  
    return a
```

Merge sort — příklad

```
a=[random.randrange(100) for i in range(10)]  
print(a)
```

```
[36, 39, 3, 20, 68, 68, 78, 99, 10, 99]
```

```
print(merge_sort(a))
```

```
[3, 10, 20, 36, 39, 68, 68, 78, 99, 99]
```

Námět: Implementujte *radix sort* pomocí front.

Lineární datové struktury

(Linear data structures)

Lineární datová struktura

- ▶ strukturovaný/složený datový typ
- ▶ obsahuje předem neznámé množství položek, typicky stejného typu
- ▶ pozice položky mezi ostatními je při vložení a poté se nemění

Příklady

- ▶ Pole (*array*)
- ▶ Zásobník (*stack*)
- ▶ Fronta (*queue*)
- ▶ Oboustranná fronta (*double ended queue, dequeue*)
- ▶ Spojový seznam (*linked list*)

Záznam

Zásobník

Fronta