

# 9. Spojový seznam

## BAB37ZPR – Základy programování

Stanislav Vítek

Katedra radioelektroniky  
Fakulta elektrotechnická  
České vysoké učení v Praze

# Přehled témat

---

- Část 1 – Rekurze
  
- Část 2 – Spojový seznam

Spojový seznam

Implementace v Pythonu

Část I

Rekurze

# Rekurze

---

- **Rekurze** = odkaz na sama sebe, definice za pomoci sebe sama
- **Rekurzivní funkce** = funkce volá sama sebe (i nepřímo)
- Je to hlavně způsob přemýšlení o řešení problémů
  - Řešení problému za pomoci řešení jednodušší varianty téhož problému
  - O menší instance se nemusíme příliš starat, musíme ale dodržovat **pravidla**
- Obvykle elegantní × je potřeba hlídat efektivitu

Rekurze bývá paměťově náročná.

## Pravidla pro správné použití rekurze

1. Dostatečně jednoduché vstupy musíme vyřešit přímo.
2. Rekurzivní volání musí být v nějakém smyslu jednodušší než je aktuální problém.

# Příklad: Umocňování

---

## Přímá definice

$$x^n = \prod_{i=1}^n x = \underbrace{x \cdot x \cdots x}_{n\text{-krát}}$$

## Rekurzivní definice

$$\begin{aligned} x^0 &= 1 \\ x^{n+1} &= x x^n \quad \text{for } n > 0 \end{aligned}$$

## Anatomie rekurze

- Základní / bázový případ (*base case*)
- Převod problému na jednodušší
- Odkaz na sebe / Rekurzivní volání

- iterativně

```
def power_iterative(x,n):  
    prod=1.0  
    for i in range(n): prod*=x  
    return prod
```

```
>>> power_iterative(2.0,10)  
1024.0
```

- rekurzivně

```
def power_recursive(x,n):  
    if n<=0: return 1.0  
    return x*power_recursive(x,n-1)
```

```
>>> power_recursive(2.0,10)  
1024.0
```

## Příklad: Umocňování – implementace (2/2)

---

- přímočará verze

```
def power_recursive(x,n):  
    if n<=0:  
        return 1.0  
    return x*power_recursive(x,n-1)
```

- stručnější verze

```
def power_recursive2(x,n):  
    return x*power_recursive2(x,n-1) if n>0 else 1.0
```

Volte verzi, která je pro vás čitelnější.

# Odbočka: Volání funkcí

---

Co už víme o tom, jak probíhá volání funkcí?

- Zásobník volání (call stack)
  - hodnoty lokálních proměnných a parametrů
  - návratová adresa (kam se máme vrátit)

Vztah rekurze a iterace

- každý rekurzivní program je možno přepsat jako iterativní za pomoci zásobníku
- (explicitní) zásobník zde simuluje zásobník volání funkcí
  - hodnoty lokálních proměnných
  - kde jsme byli (v původní funkci)
- často, ale ne vždy, se dá zjednoduši



## Příklad: součet pole

---

```
def sum_array(a):  
    s=0  
    for x in a:  
        s+=x  
    return s
```

```
>>> a=[68, 0, 61, 34, 2, 51, 29, 10, 5, 45]  
>>> sum_array(a)  
305
```

Šlo by to napsat bez cyklu?

## Příklad: součet pole – rekurzivně

---

```
def sum_array_recursive(a):  
    if len(a)==0:  
        return 0  
    return a[0]+sum_array_recursive(a[1:])
```

```
>>> a=[68, 0, 61, 34, 2, 51, 29, 10, 5, 45]  
>>> sum_array_recursive(a)  
305
```

# Nepřímá rekurze

---

- rekurzivní funkce se nemusí volat přímo, ale i skrz jinou funkci

```
def even(num):  
    print("even", num)  
    odd(num-1)
```

```
def odd(num):  
    print("odd", num)  
    if num>1:  
        even(num-1)
```

```
>>> even (3)  
even 3  
odd 2  
even 1  
odd 0
```

- Iterativní verze

```
def count_to(n):  
    for i in range(1,n+1): print(i, end=" ")  
  
count_to(5)
```

```
1 2 3 4 5
```

- Rekurzivní verze

```
def count_to_recursive(n):  
    count_to_recursive_inner(n,1)  
  
def count_to_recursive_inner(n,i):  
    if i<=n: print(i, end=" ")  
    count_to_recursive_inner(n,i+1)  
  
count_to_recursive(5)
```

```
1 2 3 4 5
```

- rekurzivně s vnitřní funkcí

```
def count_to_recursive2(n):  
    def count_to_recursive_inner(i):  
        if i<=n:  
            print(i, end=" ")  
            count_to_recursive_inner(i+1)  
    count_to_recursive_inner(1)  
count_to_recursive2(5)
```

```
1 2 3 4 5
```

# Vnitřní funkce

---

```
def count_to_recursive2(n):  
    def count_to_recursive_inner(i):  
        if i<=n:  
            print(i)  
            count_to_recursive_inner(i+1)  
    count_to_recursive_inner(1)
```

- + Skrytí soukromých funkcí
- + Sdílení proměnných vnější funkce
- Nemožnost znovupoužití
- Nemožnost samostatného odladění

# Porušení pravidel

---

- Co se stane?

```
def wrong1(num):  
    return num*wrong1(num-1)
```

```
#
```

```
def wrong2(num):  
    if num<=1:  
        return num  
  
    return 1+wrong1(num)
```

```
#
```

```
def wrong3(num):  
    if num > 1000:  
        return 1000  
  
    return 1+wrong3(num)
```

# Recursion Error

---

- Velikost zásobníku volání je typicky nějak omezena
- V Pythonu implicitně na 1000 volání, dá se změnit
- Překročení limitu zásobníku volání je chyba
- V Pythonu `Recursion Error`

jinde též `stack overflow`

- znamená to, že nemáme při programování používat rekurzi?
- **NE!** znamená to, že ji máme používat rozumně
- nezapomeňte, že rekurze je i způsob přemýšlení



- Iterativně

```
def reverse_iterative(s):  
    r="" # result  
    for i in range(len(s)-1,-1,-1):  
        r+=s[i]  
    return r  
  
print(reverse_iterative("dobry vecer"))
```

```
recev yrbdod
```

- Rekurzivně

```
def rev_rec1(s):  
    if len(s)==0:  
        return ""  
    return rev_rec1(s[1:])+s[0]  
print(rev_rec1("dobry vecer"))
```

- Kratší verze

```
def rev_rec2(s):  
    return "" if s=="" else rev_rec2(s[1:])+s[0]  
print(rev_rec2("dobry vecer"))
```

- Pythonská verze

```
print("dobry vecer"[::-1])
```

Převeď číslo  $n$  v soustavě se základem  $b$  na řetězec.

- $5_{10} = 101_2$  protože  $1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5$ .
- Pokud  $b > 10$  používáme A = 10, B = 11, ...
- Např.  $2016_{10} = 7E0_{16}$  protože  $7 \cdot 16^2 + 14 \cdot 16^1 + 0 \cdot 16^0 = 2016$

## Myšlenka řešení

- Pokud  $n < b$  pak vrať číslici odpovídající  $n$ .
- Jinak
  - Najdi  $n // b$  v soustavě  $b$ .
  - Přidej nakonec číslici odpovídající  $n \% b$

- vrať `n` jako řetězec v číselné soustavě se základem `base`.

```
def to_str(n, base):  
    assert(n>=0)  
    cislice = "0123456789ABCDEF"  
    if n < base:  
        return cislice[n]  
    return to_str(n // base, base) + cislice[n % base]  
print(to_str(5,2))
```

101

```
print(to_str(2016,16))
```

7E0

- Nerekurzivní řešení
  - Každé rekurzivní řešení je možné napsat bez rekurze
    - Ale může to být těžké.
  - Nutnost zapamatovat si lokální proměnné v jednotlivých voláních
    - Například v zásobníku (stack).

```
def to_str_nonrecursive(n,base):
    cislice = "0123456789ABCDEF"
    stack=[n] # hodnoty n
    n//=base
    while n>0:
        stack+=[n]
        n//=base
    result=""
    for m in stack[::-1]:
        result+=cislice[m % base]
    return result
```

- Nerekurzivní řešení bez zásobníku

```
def to_str_nonrecursive2(n,base):  
    assert(n>=0)  
    cislice="0123456789ABCDEF"  
    result=""  
    while True:  
        result=cislice[n % base]+result  
        n//=base  
        if n==0: break  
    return result
```

- Přidávání na začátek řetězce je pomalé (lineární).
- Skrytě kvadratický algoritmus.

# Permutace

---

## Problém

Vytiskni všechny permutace prvků dané množiny  $M$ .

## Myšlenka řešení

- Vezmi každý prvek  $m_i$  z  $M$ .
  - Najdi všechny permutace prvků  $M \setminus \{m_i\}$
  - Ke každé na začátek přidej  $m_i$

## Permutace – implementace

---

```
def tisk_permutaci(m):  
    """ Vytiskne vsechny permutace prvku v 'm' """  
    tisk_permutaci_acc(m, "")  
  
# acc - retezec pridavany na zacatek  
def tisk_permutaci_acc(m, acc):  
    if len(m)==0:  
        print(acc, end=" ", "  
    else:  
        for i in range(len(m)):  
            tisk_permutaci_acc(m[:i]+m[i+1:], acc+m[i]+" "  
  
tisk_permutaci(["a", "b", "c", "d"])
```

```
a b c d , a b d c , a c b d , a c d b , a d b c , a d c b ,  
b a c d , b a d c , b c a d , b c d a , b d a c , b d c a ,  
c a b d , c a d b , c b a d , c b d a , c d a b , c d b a ,  
d a b c , d a c b , d b a c , d b c a , d c a b , d c b a ,
```



# Příklad: Mince

---

## Problém

Vypiš všechny způsoby, jak zaplatit  $x$  Kč mincemi.

Hodnoty mincí  $h = \{50, 20, 10, 5, 2, 1\}$  Kč.

## Myšlenka řešení

- Vyber největší minci  $h_i \leq x$ . Pak jsou dvě možnosti:
  - Použij  $h_i$  — zaplať  $x - h_i$  pomocí  $h_i, h_{i+1}, \dots, h_n$  a přidej jednu  $h_i$ .
  - Nepoužij  $h_i$  — zaplať  $x$  pomocí  $h_{i+1}, \dots, h_n$

- Vytiskni všechny možné způsoby, jak zaplatit x Kč.

```
def zaplat(x):
    h=[50,20,10,5,2,1] # hodnoty minci sestupne
    def doplat(x,m,i):
        """ m - kolik zaplaceno v poctech minci
            i - kterou minci zacit """
        if x==0:
            vytiskni_platbu(m,h)
        else:
            if x>=h[i]: # zaplať minci h[i]
                doplat(x-h[i],m[:i]+[m[i]+1]+m[i+1:],i)
            if i<len(h)-1: # zaplať menšími, lze-li
                doplat(x,m,i+1)
    doplat(x,len(h)*[0],0) # zacatek fce zaplat
```

```
def vytiskni_platbu(m,h):  
    """ m - pocty minci, h - hodnoty """  
    for j in range(len(h)):  
        if m[j]>0:  
            print("%3d*%3dKc" % (m[j],h[j]), end="")  
    print("")
```

```
>>> zaplat(12)
1* 10Kč 1* 2Kč
1* 10Kč 2* 1Kč
2* 5Kč 1* 2Kč
2* 5Kč 2* 1Kč
1* 5Kč 3* 2Kč 1* 1Kč
1* 5Kč 2* 2Kč 3* 1Kč
1* 5Kč 1* 2Kč 5* 1Kč
1* 5Kč 7* 1Kč
6* 2Kč
5* 2Kč 2* 1Kč
4* 2Kč 4* 1Kč
3* 2Kč 6* 1Kč
2* 2Kč 8* 1Kč
1* 2Kč 10* 1Kč
12* 1Kč
```

```
def zaplat2(x):
    h=[50,20,10,5,2,1] # hodnoty minci sestupne
    def doplat(x,m,i):
        """ m - kolik zaplaceno v poctech minci
            i - kterou minci zacit """
        if x==0:
            vytiskni_platbu(m,h)
        else:
            if x>=h[i]: # zaplat minci h[i]
                m[i]+=1
                doplat(x-h[i],m,i)
                m[i]-=1 # úklid
            if i<len(h)-1: # zaplat mensimi
                doplat(x,m,i+1)
    doplat(x,len(h)*[0],0)
```

```
>>> zaplat2(12)
1* 10Kč  1*  2Kč
1* 10Kč  2*  1Kč
2*  5Kč  1*  2Kč
2*  5Kč  2*  1Kč
1*  5Kč  3*  2Kč  1*  1Kč
1*  5Kč  2*  2Kč  3*  1Kč
1*  5Kč  1*  2Kč  5*  1Kč
1*  5Kč  7*  1Kč
6*  2Kč
5*  2Kč  2*  1Kč
4*  2Kč  4*  1Kč
3*  2Kč  6*  1Kč
2*  2Kč  8*  1Kč
1*  2Kč 10*  1Kč
12*  1Kč
```

## Část II

### Spojový seznam

## II. Spojový seznam

---

Spojový seznam

Implementace v Pythonu



# Spojový seznam

---

- V programech je velmi běžný požadavek na uchování seznamu (množiny) prvků (proměnných/struktur)
- Základní kolekce je **pole**
  - Jedná se o kolekci položek (proměnných) stejného typu
  - + Umožňuje jednoduchý přístup k položkám indexací prvku
    - Položky jsou stejného typu (velikosti).**
  - Velikost pole je určena při vytvoření pole
    - Velikost (maximální velikost) musí být známa v době vytváření
    - Změna velikost v podstatě není přímo možná
      - Nutné nové vytvoření (alokace paměti), resp. realloc.**
    - Využití pouze malé části pole je mrháním paměti
  - V případě řazení pole přesouváme položky
    - Vložení prvku a vyjmutí prvku vyžaduje kopírování

# Odbočka: Pole v Pythonu?

---

- V Pythonu obvykle pracujeme se seznamem...
- Přesto i zde existuje datový typ pole

O datovém typu pole budeme ještě mluvit v souvislosti s knihovnou numpy.

```
>>> import array from array
>>> a = array('l', [1, 2, 3, 4])
>>> type(a)
<class 'array.array'>
>>> a[1]
2
>>> for i in range(len(a)):
        print(a[i], end=" ")

1 2 3 4
```

<https://docs.python.org/3/library/array.html>

# 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

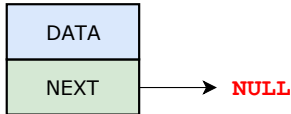
Nová alokace a kopírování.

- Spojové seznamy

# Spojové seznamy

---

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

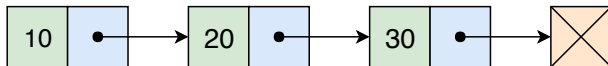
Obousměrný spojový seznam.

- Odebrání prvku
  - Předchozí prvek aktualizuje hodnotu odkazu na následující prvek
  - Předchozí prvek tak nově odkazuje na následující hodnotu, na kterou odkazoval odebíraný prvek
- Základní implementací spojového seznamu je tzv.

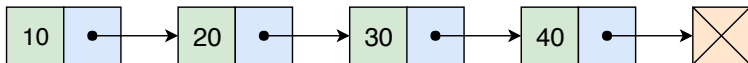
**jednosměrný spojový seznam**

# Jednosměrný spojový seznam

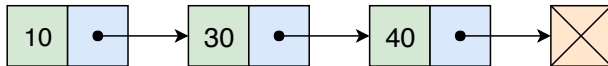
- Příklad spojového seznamu pro uložení číselných hodnot



- 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.

## II. Spojový seznam

---

Spojový seznam

Implementace v Pythonu

# Reprezentace uzlu

---

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



# Spojový seznam jako zásobník

---

```
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
```

## Spojový seznam jako zásobník – příklad

---

```
from linkedliststack import ListStack
s = ListStack()
s.push(1)
s.push(2)
print(s.pop(), s.pop(), end=" ")
```

```
2 1
```

```
s.push(10)
print(s.pop())
```

```
10
```

```
print(s.is_empty()) # True
```

## Spojový seznam jako fronta

---

```
from linkedliststack import Node, ListStack
class ListQueue(ListStack):    # zdědíme 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()
```

## Spojový seznam jako fronta (2)

---

```
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
```

## Spojový seznam jako fronta – příklad

---

```
from linkedlistqueue import ListQueue
q=ListQueue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
print(q.dequeue(), q.dequeue(), end = " ")
```

```
1 2
```

```
q.enqueue(10)
print(q.dequeue())
```

```
3
```

```
print(q.is_empty()) # False
```

# Procházení prvků pole

---

- Doplníme do třídy `ListQueue` metody:

```
def iter(self,f):
    """ execute f(x) for all 'x' in the queue """
    node=self.head
    while node is not None:
        f(node.data)
        node=node.next

def reduce(self,f,acc):
    """ execute acc=f(x,acc) for all 'x' in the queue
    """
    node=self.head
    while node is not None:
        acc=f(node.data,acc)
        node=node.next
    return acc
```

# Konverze na pole a z pole

---

- Doplníme do třídy `ListQueue` metodu:

```
def to_array(self):  
    a=[]  
    self.iter(lambda x: a.append(x))  
    return a
```

- Funkce pro vytvoření seznamu z pole

```
def array_to_queue(a):  
    q=ListQueue()  
    for x in a:  
        q.enqueue(x)  
    return q
```

linkedlistqueue.py

## Příklady

---

```
from linkedlistqueue import ListQueue,array_to_queue
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

---

- Metoda třídy `ListQueue`, složitost  $O(n)$

```
def contains(self,x):  
    """ returns True if the list contains element x """  
    node=self.head  
    while node is not None:  
        if node.data==x:  
            return True  
        node=node.next  
    return False
```

```
q=array_to_queue([3,52,69,17,19])
```

```
print(q.contains(17))
```

True

```
print(q.contains(20))
```

False

# Mazání v seznamu

---

- Metoda třídy `ListQueue`:

```
def remove(self,x):
    """ removes an element x, if present """
    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
```

# Mazání v seznamu

---

- Příklad

```
from linkedlistqueue import ListQueue,array_to_queue
q=array_to_queue([3,2,5,8,11])
q.remove(5)
print(q.to_array())
```

```
[3, 2, 8, 11]
```

```
q.remove(3)
print(q.to_array())
```

```
[2, 8, 11]
```

```
q.remove(11)
print(q.to_array())
```

```
[8, 11]
```

## 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):  
    """ sorts array a inplace in ascending order """  
    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
```

insertion\_sort\_linkedlist.py

# Spojování seznamů v konstantním čase

---

- Doplníme do třídy `ListQueue` metodu

```
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
```

## Spojování seznamů – příklad

---

```
from linkedlistqueue import ListQueue, array_to_queue
q=array_to_queue([1,2,3])
r=array_to_queue([4,5])
q.concatenate(r)
print(q.to_array())
```

[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			$O(1)$	$O(n)$ <sup>‡</sup>	$O(1)$
iterace vpřed				ano	ano
iterace vzad					ano

*začátek* = self.head

*konec* = self.last

<sup>‡</sup>  $O(1)$ , 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).