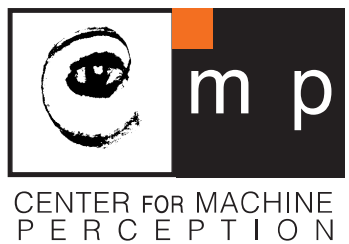
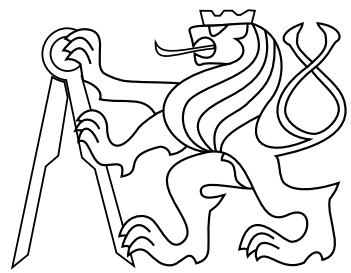
Spojový seznam

Jan Kybic

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

2016–2017





Složitost operací u lineárních datových struktur

				v Pythonu	
operace	zásob.	fronta	pole	pole	řetězce
přidej na začátek	$O(1)$			$O(n)$	
přidej na konec		$O(1)$		$O(1)$	$O(1)$
vlož doprostřed				$O(n)$	
odeber ze začátku	$O(1)$	$O(1)$		$O(n)$	
odeber kdekoliv				$O(n)$	
odeber z konce				$O(1)$	
indexový přístup			$O(1)$	$O(1)$	$O(1)$
spojení				$O(n)$	$O(n)$
iterace vpřed			ano	ano	ano
iterace vzad			ano	ano	ano

Přímo nepodporované operace lze implementovat v čase $O(n)$.
“Konec” a “začátek” jsou zaměnitelné.

Složitost operací u lineárních datových struktur

				v Pythonu		spoj.sez.	
operace	zásob.	fronta	pole	pole	řetězce	jedn.	dvoj.
přidej na začátek	$O(1)$			$O(n)$		$O(1)$	$O(1)$
přidej na konec		$O(1)$		$O(1)$	$O(1)$	$O(1)^*$	$O(1)$
vlož doprostřed				$O(n)$		$O(1)^*$	$O(1)^*$
odeber ze začátku	$O(1)$	$O(1)$		$O(n)$		$O(1)$	$O(1)$
odeber kdekoliv				$O(n)$		$\sharp$	$O(1)$
odeber z konce				$O(1)$		$\sharp$	$O(1)$
indexový přístup			$O(1)$	$O(1)$	$O(1)$		
spojení				$O(n)$	$O(n)$	$O(1)$	$O(1)$
iterace vpřed			ano	ano	ano	ano	ano
iterace vzad			ano	ano	ano		ano

Přímo nepodporované operace lze implementovat v čase $O(n)$.
"Konec" a "začátek" jsou zaměnitelné.

Jednoduše/dvojitě zřetězený spojový seznam *(singly/doubly linked list)*.

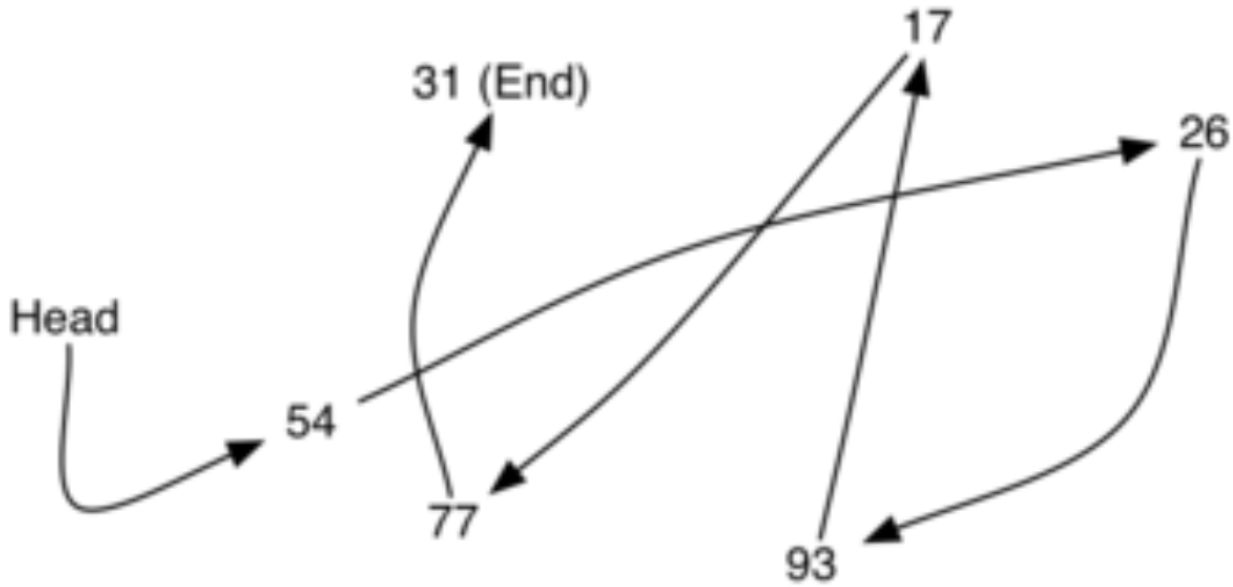
* pokud máme odkaz na příslušný uzel.

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

Spojový seznam

Základní myšlenka:

- Data jsou součástí *uzlů*.
- *Uzly* = data + odkaz na následníka
  - Uzly *dvojitě zřetězeného* seznamu obsahují odkaz i na předchůdce.
- Seznam = odkaz na první uzel.
  - Pokud chceme přidávat na konec, pak potřebujeme odkaz i na poslední uzel.



Images courtesy of Brad Miller, David Ranum.

Spojový seznam

Základní myšlenka:

- Data jsou součástí *uzlů*.
- *Uzly* = data + odkaz na následníka
  - Uzly *dvojitě zřetězeného* seznamu obsahují odkaz i na předchůdce.
- Seznam = odkaz na první uzel.
  - Pokud chceme přidávat na konec, pak potřebujeme odkaz i na poslední uzel.

Images courtesy of Brad Miller, David Ranum.

Reprezentace uzlu

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

Soubor `linkedliststack.py`.

Spojový seznam jako zásobník — příklad

```
s=ListStack()
s.push(1)
s.push(2)
s.push(3)
print(s.pop())
print(s.pop())
```

```
3
2
```

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

```
10
```

```
print(s.is_empty())
```

```
False
```

```
print(s.pop())
```

```
1
```

```
print(s.is_empty())
```

```
True
```

Soubor `linkedlist_examples.py`

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
```

Soubor `linkedlistqueue.py`

Spojový seznam jako fronta — příklad

```
q=ListQueue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
print(q.dequeue())
print(q.dequeue())
```

```
1
2
```

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

```
3
```

```
print(q.is_empty())
```

```
False
```

```
print(q.dequeue())
```

```
10
```

```
print(q.is_empty())
```

```
True
```

Soubor `linkedlist_examples.py`

Procházení prvků pole
linked list traversal

Doplníme do třídy `ListQueue` metody

```
def iter(self,f):
  """ execute f(x) for all elements '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
```

Soubor `linkedlistqueue.py`

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
```

Soubor `linkedlistqueue.py`

Příklady

```
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
```

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
```

Soubor `linkedlistqueue.py`

Mazání v seznamu — příklad

```
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())
```

```
[2, 8]
```

Soubor `linkedlist_examples.py`

Uspořádaný spojový seznam — řazení
(Sorted/ordered linked list)

- 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):
    """ inserts item x in a descending order list"""
    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
```

Soubor `insertion_sort_linkedlist.py`

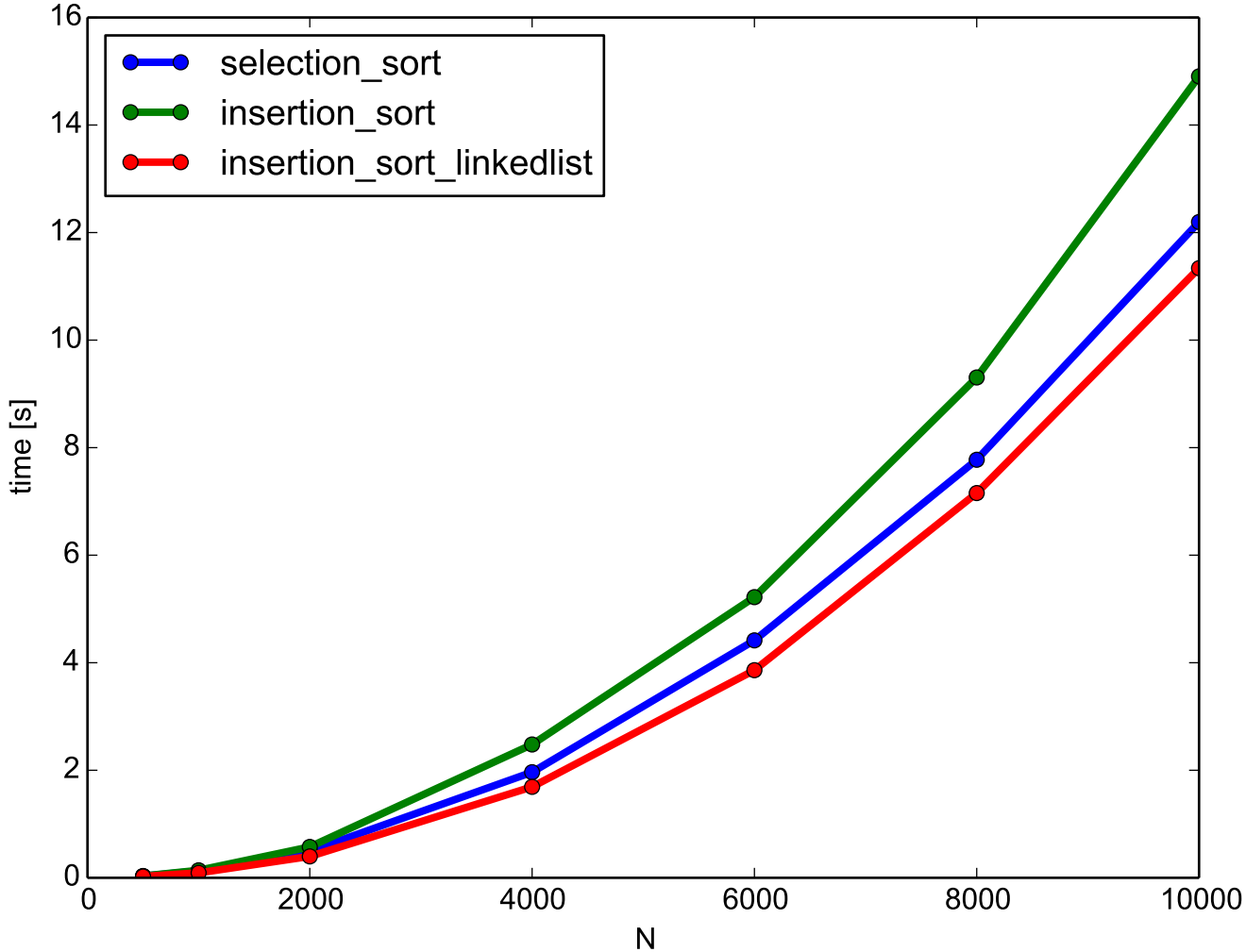
Řazení vkládáním a spojové seznamy

Insertion sort

```
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
```

Soubor `insertion_sort_linkedlist.py`

Řazení vkládáním — měření rychlosti



Spojování seznamů
v konstantním čase

Doplníme do třídy `ListQueue` metodu

```
def concatenate(self,l):
  """ destruktivně přidej 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
```

Soubor `linkedlistqueue.py`.

Spojování seznamů — příklad

```
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]
```

Soubor `linkedlist_examples.py`

Radix sort a spojové seznamy

```
def radix_sort_queue_integers(q):
  """ setřídí frontu ListQueue přirozených čísel """
  max_digits=num_digits(q.reduce(max,0))
  for i in range(max_digits):
    q=sort_queue_by_digit(q,i)
  return q

def sort_queue_by_digit(q,i):
  prihradka=[ ListQueue() for j in range(10) ]
  while not q.is_empty():
    x=q.dequeue()
    c=digit(x,i) # číslice pro třídění
    prihradka[c].enqueue(x)
  r=ListQueue()
  for p in prihradka:
    r.concatenate(p)
  return r
```

Soubor `radix_sort_linkedlist.py`

Radix sort a spojové seznamy

```
def radix_sort_queue_integers(q):
  """ setřídí frontu ListQueue přirozených čísel """
  max_digits=num_digits(q.reduce(max,0))
  for i in range(max_digits):
    q=sort_queue_by_digit(q,i)
  return q

def sort_queue_by_digit(q,i):
  prihradka=[ ListQueue() for j in range(10) ]
  while not q.is_empty():
    x=q.dequeue()
    c=digit(x,i) # číslice pro třídění
    prihradka[c].enqueue(x)
  r=ListQueue()
  for p in prihradka:
    r.concatenate(p)
  return r
```

V Pythonu se bohužel nevyplatí, režie je příliš velká.

Soubor `radix_sort_linkedlist.py`

Oboustranná fronta
(double ended queue)

Lineární datová struktura kombinující frontu a zásobník.

operace	zásob.	fronta	oboustranná fronta	spoj.sez. jedn.	spoj.sez. 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			$O(1)$	$O(n)^\sharp$	$O(1)$
iterace vpřed				ano	ano
iterace vzad					ano

začátek = `self.head`
konec = `self.last`
$^\sharp$ $O(1)$, 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).