

# Základy algoritmizace

## 6. Abstraktní datové typy

doc. Ing. Jiří Vokřínek, Ph.D.

Katedra počítačů

Fakulta elektrotechnická

České vysoké učení technické v Praze

# Základy algoritmizace

- Dnes:
  - Abstraktní datové typy
  - Zásobník
  - Fronta
  - Spojový seznam

# Abstraktní datový typ

# Abstraktní datový typ

- **Datová struktura** (typ) je množina dat a operací s těmito daty
- **Abstraktní datový typ** formálně definuje data a operace s nimi

*Nezávisle na konkrétní implementaci*

- **Příklady:**
  - Fronta (queue)
  - Zásobník (stack)
  - Pole (array)
  - Tabulka (table)
  - Seznam (list)
  - Strom (tree)
  - Množina (set)

# Abstraktní datový typ

- **Abstraktní datový typ** je množina dat (hodnot) a příslušných operací, které jsou přesně specifikovány a to **nezávisle na konkrétní implementaci**
  
- **Můžeme definovat:**
  - Matematicky – signatura a axiomy
  - Rozhraním (interface) a popisem operací, rozhraní poskytuje:
    - Konstruktor vracející odkaz (na objekt nebo strukturu)  
*Objektově orientovaný i procedurální přístup*
    - Operace, které akceptují odkaz na argument (data) jako argument a které mají přesně definovaný účinek na data
  
- **Příklad matematického popisu – datový typ Boolean**

# Matematický popis ADT - Boolean

- **Syntax** popisuje, jak správně vytvořit logický výraz:

1. `true` a `false` jsou logické výrazy
2. Jestliže  $x$  a  $y$  jsou logické výrazy, pak

- I.  $!(x)$  – negace
- II.  $(x \& y)$  – logický součin `and`
- III.  $(x | y)$  – logický součet `or`
- IV.  $(x == y)$ ,  $(x != y)$  – relační operátory

jsou logické výrazy

*Pokud se chceme vyhnout psát u každé operace závorky,  
musíme definovat priority operátorů*

*Konkrétní implementace se může syntakticky lišit – viz. `!` vs. `not`, atp.*

# Matematický popis ADT - Boolean

- **Sémantika** popisuje význam jednotlivých operací

- Můžeme definovat axiomy:

- `true == true : true`

- `false == false : true`

- `!(true) : false`

- `x & false : false`

- `x & y : y & x`

- `x | false : x`

- `true == false : false`

- `false == true : false`

- `!(false) : true`

- `x & true : x`

- `x | y : y | x`

- `x | true : true`

# Abstraktní datový typ - vlastnosti

- Počet datových položek může být
  - Neměnný – **statický datový typ** – počet položek je konstantní  
*Např. pole, řetězec, třída, ...*
  - Proměnný – **dynamický datový typ** – počet položek se mění v závislosti na provedené operaci  
*Např. vložení nebo odebrání určitého prvku*
- Typ položek (dat) může být
  - **Homogenní** – všechny položky jsou stejného typu
  - **Nehomogenní** – položky mohou být různých typů
- Existence bezprostředního následníka je
  - **Lineární** – existuje bezprostřední následník prvku, např. pole, fronta, seznam, ...
  - **Nelineární** – neexistuje přímý jednoznačný následník, např. strom



# Zásobník

# Zásobník

- Dynamická datová struktura umožňující vkládání a odebrání hodnot tak, že naposledy vložená hodnota se odebere jako první

*LIFO – Last In, First Out*

- Základní operace
  - Vložení hodnoty na vrchol zásobníku
  - Odebrání hodnoty z vrcholu zásobníku
  - Test prázdnosti zásobníku



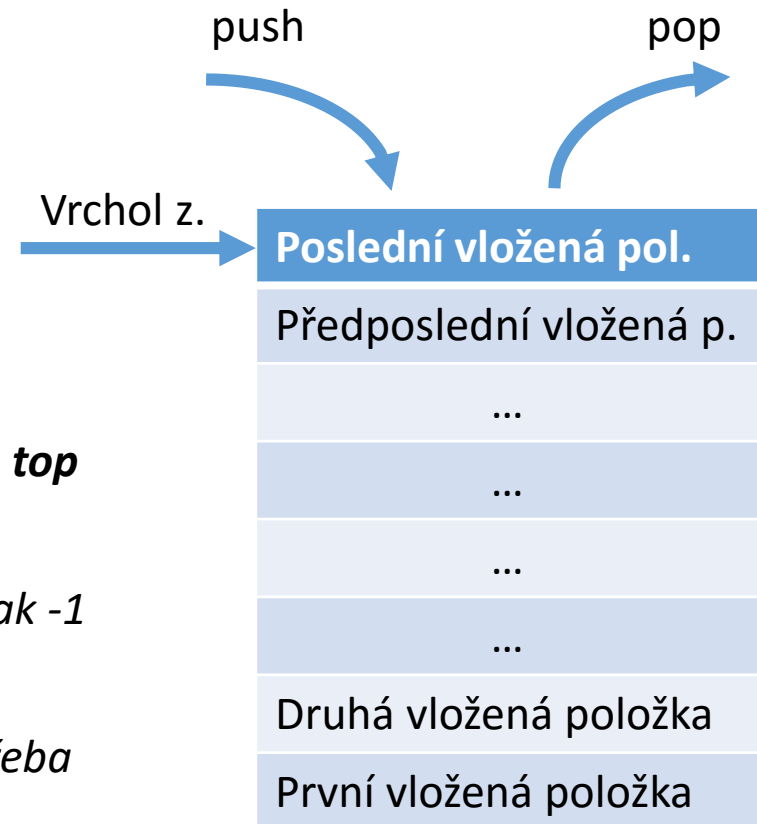
# Zásobník

## ■ Základní operace

- **push** – vložení hodnoty na vrchol zásobníku
- **pop** – odebrání hodnoty z vrcholu zásobníku
- **isEmpty** – test prázdnosti zásobníku

## ■ Další operace mohou být

- **peek** – čtení hodnoty z vrcholu  
*Alternativně také třeba top*
- **search** – vrátí pozici prvku  
*Pokud se nachází v zásobníku, jinak -1*
- **size** – aktuální počet prvků  
*Nebývá potřeba*



# Zásobník

- Příklad implementace pomocí List
  - push  $\approx$  append
  - pop  $\approx$  pop
  - isEmpty  $\approx$  not len(stack)

```
stack = []
stack.append("dela")
stack.append("to")
stack.append("co")
stack.append("vime")

while len(stack):
    print(stack.pop())
```

*Uměli byste implementovat zásobník pomocí pole?*

# Zásobník

- Příklad implementace s definicí rozhraní

```
class Stack:
```

```
    def __init__(self):  
        self.items = []
```

```
    def push(self, item):  
        self.items.append(item)
```

```
    def pop(self):  
        return self.items.pop()
```

```
    def isEmpty(self):  
        return (self.items == [])
```

```
s = Stack()  
s.push("dela")  
s.push("to")  
s.push("co")  
s.push("vime")
```

```
while not s.isEmpty():  
    print(s.pop())
```

# Zásobník

- Příklad použití – kontrola vyváženosti závorek

```
def parChecker(symbolString):  
    s = Stack()  
    for symbol in symbolString:  
        if symbol == "(":  
            s.push(symbol)  
        elif symbol == ")":  
            if s.isEmpty():  
                return False  
            else:  
                s.pop()  
    return s.isEmpty()
```

```
parChecker(' (3+ (2* (-2) + (3*5) -1) / (3-2) *2) ')  
parChecker(' ( (3+2) + (5*8) * (4) ')
```

# Fronta

# Fronta

- Dynamická datová struktura umožňující vkládání a odebírání hodnot v pořadí, v jakém byly vloženy

*FIFO – First In, First Out*

- Základní operace
  - Vložení hodnoty na konec fronty
  - Odebrání hodnoty z čela fronty
  - Test prázdnosti fronty





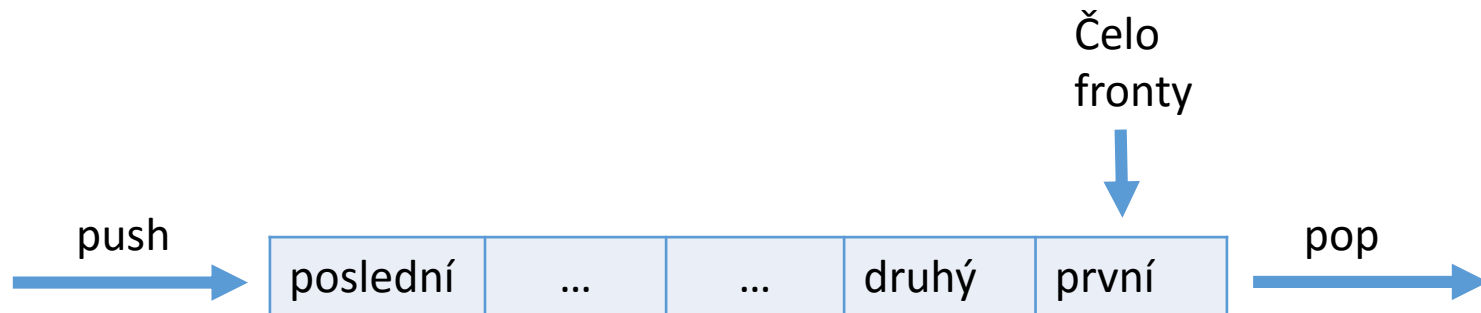
# Fronta

- Základní operace

- `push` – vložení hodnoty na konec fronty (nebo též `enqueue`)
- `pop` – odebrání hodnoty z čela fronty (nebo též `dequeue`)
- `isEmpty` – test prázdnosti fronty

- Další operace mohou být

- `peek` – čtení hodnoty z čela
- `back` – čtení hodnoty z konce



# Fronta

- Příklad implementace pomocí List
  - push  $\approx$  append
  - pop  $\approx$  pop(0)
  - isEmpty  $\approx$  not len(stack)

```
queue = []  
queue.append("dela")  
queue.append("to")  
queue.append("co")  
queue.append("vime")  
  
while len(queue):  
    print(queue.pop(0))
```

*Pozor, operace pop(0) je pomalá – víte proč?*

# Fronta

## ■ Příklad implementace s definicí rozhraní

*Jaký je rozdíl v použití seznamu oproti minulému příkladu?*

```
class Queue:
```

```
    def __init__(self):  
        self.items = []
```

```
    def isEmpty(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 = Queue()  
q.enqueue("dela")  
q.enqueue("to")  
q.enqueue("co")  
q.enqueue("vime")
```

```
while not q.isEmpty():  
    print(q.dequeue())
```

# Fronta

- Rozšíření – prioritní fronta

- Prvky jsou odebírány na základě priority (např. velikosti)

*Jak to lze naimplementovat?*

- Hledání největšího prvku při odebírání
- Seřazení po vložení
- Seřazení při vložení

*Určitě to jde lépe s pomocí složitějších struktur*

# Spojové struktury

# Spojové struktury

## ■ Seznam – List

- Základní datová struktura pro uchovávání množiny prvků
- Vložení/vyjmutí prvku může být velmi pomalé

## ■ Spojové seznamy

- Datová struktura realizující seznam dynamické délky
- Každý prvek obsahuje
  - Datovou část (hodnota proměnné, objekt, ...)
  - Referenci na další prvek

*None v případě posledního prvku seznamu*

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

*Realizováno jako referenční proměnná odkazující na první prvek seznamu*

# Spojové struktury

- Prvek seznamu – **Node**
- Základ i pro komplikovanější struktury

```
class Node:  
    def __init__(self, initdata):  
        self.data = initdata  
        self.next = None
```

- Spojový seznam – **LinkedList**

```
class LinkedList:  
    def __init__(self):  
        self.head = None  
        ...  
        ...
```

# Spojové struktury

- Test prázdnoti – isEmpty

```
def isEmpty(self):  
    return self.head == None
```

- Přidání prvku – push

```
def push(self, item):  
    node = Node(item)  
    node.next = self.head  
    self.head = node
```

*Přidání prvku není závislé na počtu prvků v seznamu*



# Spojové struktury

## ■ Odebrání prvního prvku – pop

```
def pop(self):  
    if self.head:  
        ret = self.head.data  
        self.head = self.head.next  
    return ret
```

*Odebrání prvního prvku není závislé na počtu prvků v seznamu*

## ■ Zjištění počtu prvků – size

```
def size(self):  
    current = self.head  
    count = 0  
    while current != None:  
        count += 1  
        current = current.next  
    return count
```

*Kompletní projití seznamu!*

# Spojové struktury

- Zrychlení operace *size*
  - Můžeme si pamatovat aktuální počet prvků *count*
  - Zvyšujeme a snižujeme při každé operaci vložení/vyjmutí
- Práce s posledním prvkem
  - Zavedeme si referenci na poslední prvek *end*
  - Pracujeme obdobně jako s *head*
    - V případě přidání prvku na začátek aktualizujeme pouze pokud byl seznam dosud prázdný
    - Aktualizujeme v případě přidání prvku na konec
    - Aktualizujeme při vyjmutí posledního prvku

# Spojové struktury

```
class LinkedList:
    def __init__(self):
        self.head = None
        self.count = 0

    def push(self, item):
        node = Node(item)
        node.next = self.head
        self.head = node
        self.count +=1

    def pop(self):
        if self.head:
            ret = self.head.data
            self.head = self.head.next
            self.count -=1
            return ret

    def size(self):
        return self.count
```

# Spojovací struktury

```
class LinkedList:
    def __init__(self):
        self.head = None
        self.end = None
        self.count = 0

    def push(self, item):
        node = Node(item)
        if not self.head:
            self.head = self.end = node
        else:
            node.next = self.head
            self.head = node
        self.count += 1

    def pop(self):
        if self.head:
            ret = self.head.data
            self.head = self.head.next
            if self.head == None:
                self.end = None
            self.count -= 1
        return ret
```

# Spojové struktury

- Vložení prvku na konec – `pushEnd`

```
def pushEnd(self, item):  
    node = Node(item)  
    if not self.end:  
        self.head = self.end = node  
    else:  
        self.end.next = node  
        self.end = node  
    self.count +=1
```

# Spojové struktury

- Odebrání posledního prvku – `popEnd`

```
def popEnd(self):  
    if self.head:  
        self.count -= 1  
        ret = self.end.data  
        if self.head == self.end:  
            self.head = self.end = None  
        else:  
            cur = self.head  
            while cur.next != self.end:  
                cur = cur.next  
            self.end = cur  
            self.end.next = None  
    return ret
```

# Spojové struktury

- Obecně vkládání do seznamu

- Na začátek – push
- Na konec – pushEnd
- Za daný prvek – potřebujeme referenci na prvek, za který chceme vkládat (node)

```
newNode = Node(item)
newNode.next = node.next
node.next = newNode
```

- Na určitou pozici – insertAt

- Nutno najít odpovídající prvek – průchod seznamu
- Vložit za daný prvek
- Ošetření mezních hodnot (velikost seznamu, head, end)

# Spojové struktury

- Odebírání prvku

- Ze začátku – pop
- Z konce – popEnd
- Daný prvek – potřebujeme referenci na prvek, který ho předchází (prevNode)

```
if prevNode.next == node:  
    prevNode.next = node.next  
    node.next = None
```

- Z určité pozice – removeAt

- Nutno najít odpovídající prvek – průchod seznamu
- Odstranit prvek
- Ošetření mezních hodnot (velikost seznamu, head, end)



# Spojové struktury

- Složitější operace / práce s prvky
  - Vkládání a odebírání
  - Vyhledávání
  - Výpis prvků, ...

*Příklad: search a remove*

- Rozšíření základního spojového seznamu
  - Čítač prvků *count*
  - Reference na poslední prvek *end*

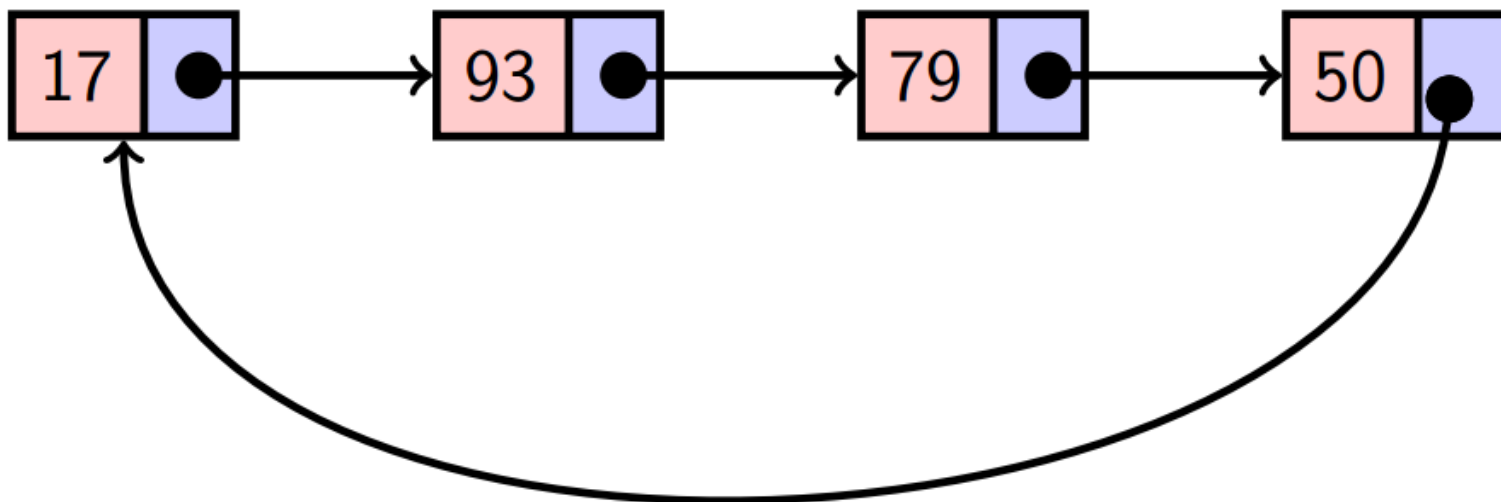
*Zjednodušuje některé operace – z lineární náročnosti na konstantní*

- Další možná rozšíření
  - Kruhový spojový seznam
  - Obousměrný spojový seznam

# Spojové struktury

- Kruhový spojový seznam

- Položka *next* posledního prvku odkazuje na první prvek
- *head* a *end* ztrácejí smysl, ale je třeba držet odkaz na některý prvek (první, poslední přidaný, poslední vyhledaný, atp.)

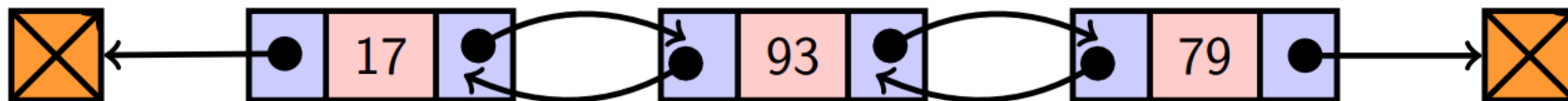


# Spojové struktury

## ■ Obousměrný spojový seznam

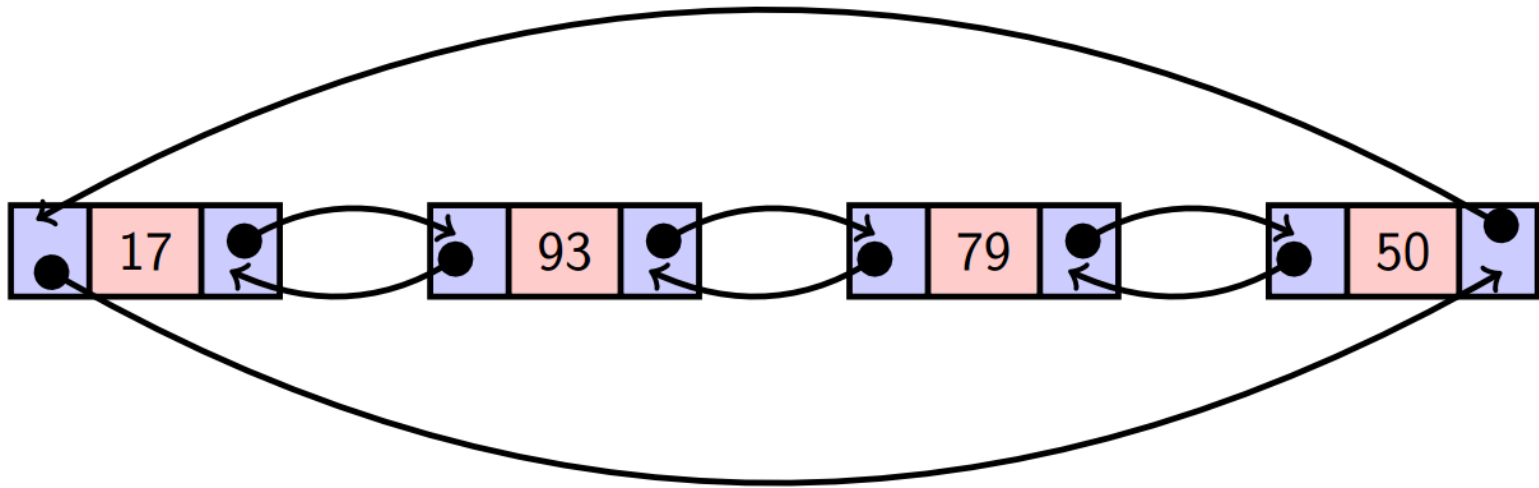
- Každý prvek obsahuje odkaz na následující i předchozí položku v seznamu (*prev* a *next*)
- První prvek má nastavenou položku *prev* na *None*
- Poslední prvek má nastavenou položku *next* na *None*
- Snadný „pohyb“ seznamem vpřed i vzad

*Srovnejte popEnd u jednosměrného spojového seznamu*



# Spojové struktury

- Kruhový obousměrný spojový seznam
  - Kombinace předchozích



# Základy algoritmizace

- Dnes:
  - Abstraktní datové typy
  - Zásobník
  - Fronta
  - Spojový seznam lineární, kruhový, obousměrný
- Zamyslete se
  - Jak udělat zásobník (frontu) pomocí spojového seznamu s konstantní složitostí operací (bez nutnosti procházet seznam)
  - Jak implementovat obousměrný spojový seznam
  - Jak udržovat spojový seznam setříděný
  - Jak implementovat prioritní frontu spojovým seznamem

**Příště** vyhledávání a řazení – grafy a stromy

# Základy algoritmizace

- Dodatečné zdroje:

- Implementing a Stack in Python

<http://interactivepython.org/runestone/static/pythonds/BasicDS/ImplementingaStackinPython.html>

- Implementing a Queue in Python

<http://interactivepython.org/runestone/static/pythonds/BasicDS/ImplementingaQueueinPython.html>

- Linked Lists in Python

<http://interactivepython.org/runestone/static/pythonds/BasicDS/ImplementinganUnorderedListLinkedLists.html>

- Implementing a singly linked list in Python

<https://www.codefellows.org/blog/implementing-a-singly-linked-list-in-python>

- Linked lists

<http://www.cs.dartmouth.edu/~cs1/chapter19/index.html>