

```

270     childpos = rightpos
271     # Move the smaller child up.
272     heap[pos] = heap[childpos]
273     pos = childpos
274     childpos = 2*pos + 1
275     # The leaf at pos is empty now. Put newitem there, and bubble it up

```

# Algoritmy a programování

Abstraktní datové struktury: fronta, zásobník

```

284     # newitem fits.
285     while pos > startpos:
286         parentpos = (pos - 1) // 2
287         parent = heap[parentpos]
288         if parent < newitem:
289             heap[pos] = parent
290             pos = parentpos
291             continue
292         break
293     heap[pos] = newitem
294
295     def _siftup_max(heap, pos):
296         'Maxheap variant of _siftup'
297         endpos = len(heap)
298         startpos = pos
299         newitem = heap[pos]
300         # Bubble up the larger child until hitting a leaf.
301         childpos = 2*pos + 1    # leftmost child position
302         while childpos < endpos:

```

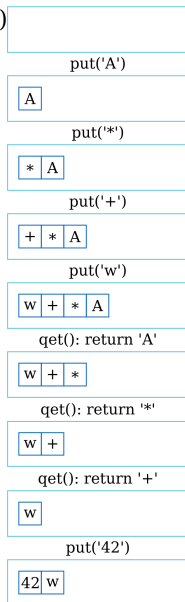
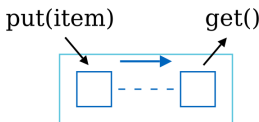
**Vojtěch Vonásek**

Department of Cybernetics

Faculty of Electrical Engineering

Czech Technical University in Prague

- Složená datová struktura
- Je to kolekce položek (typicky stejného datového typu)
- Počet položek není dopředu znám
- Požadované operace:
  - přidání položky (push, add, put, enqueue)
  - odebrání položky (pop, get, dequeue)
  - test na prázdnotu (isEmpty)
- FIFO — First In First Out
- Položky jsou odebírány **v tom pořadí, v jakém byly vloženy**
- Typická implementace používá pole, kde prvky jsou vkládány na jeden konec, a odebírány z druhého konce
- Další užitečné operace
  - čtení z konce
  - počet položek



- Vytvoříme třídu pro reprezentaci fronty
- Vnitřně budou data organizována v poli
- `put(item)`: přidá prvek do fronty (vnitřně na 0. pozici pole)
- `get()`: odebere prvek z fronty (vnitřně z poslední pozice pole)

```
1 class ALP_Queue:
2     def __init__(self):
3         self.items = []
4
5     def put(self, item):
6         self.items.insert(0, item)
7
8     def get(self):
9         return self.items.pop()
10
11    def isEmpty(self):
12        return self.items == []
13
14    def size(self):
15        return len(self.items)
```

- Tato fronta implementována jako pole, tj. lze do ní vkládat prvky různých datových typů

```
1 from alpQueue import ALP_Queue
2
3 q = ALP_Queue()
4 q.put("first")
5 q.put("2")
6 q.put(3.0)
7
8 while not q.isEmpty():
9     print(q.get())
```

```
first
2
3.0
```

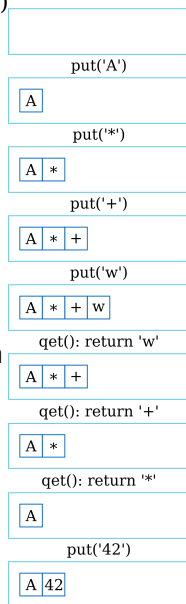
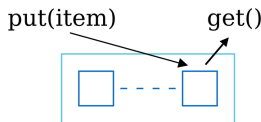
- Standardní knihovna Pythonu nabízí modul `queue`
- I tato fronta umožňuje pracovat s prvky různých datových typů

```
1 from queue import Queue
2 q = Queue()
3 q.put(1)
4 q.put("ahoj")
5 q.put(3/4)
6 while not q.empty():
7     print(q.get())
```

```
1
ahoj
0.75
```

- Obecně problémy, kde se vyžaduje data zpracovávat ve FIFO režimu
- Tisková fronta (prvně zaslané soubory jsou vtištěny jako první)
- Zpracování událostí GUI (pořadí kliknutí na různá okna)
- Zpracování dat z periférií (klávesnice, myš)
- Grafové algoritmy (prohledávání do šířky)
- Prohledávání stavového prostoru

- Složená datová struktura
- Je to kolekce položek
- Počet položek není dopředu znám
- Požadované operace:
  - přidání položky (push, add, put)
  - odebrání položky (pop, get)
  - test na prázdnotu (isEmpty)
- LIFO — Last In First Out
- Položky jsou odebírány **v opačném pořadí než v jakém byly vloženy**
- Položky mohou být organizovány v poli: přidáváme na konec, odebíráme z konce
- Další užitečné operace
  - čtení z konce
  - počet položek



- Zásobník lze jednoduše implementovat v klasickém Python poli
- `put(item)`: je realizována jako `append(item)`
- `get()`: je realizována jako `pop()`

```
1 stack = []  
2  
3 stack.append('first')  
4 stack.append(2)  
5 stack.append('last')  
6  
7 while len(stack) > 0:  
8     print( stack.pop() )
```

```
last  
2  
first
```

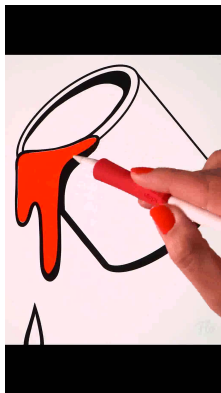


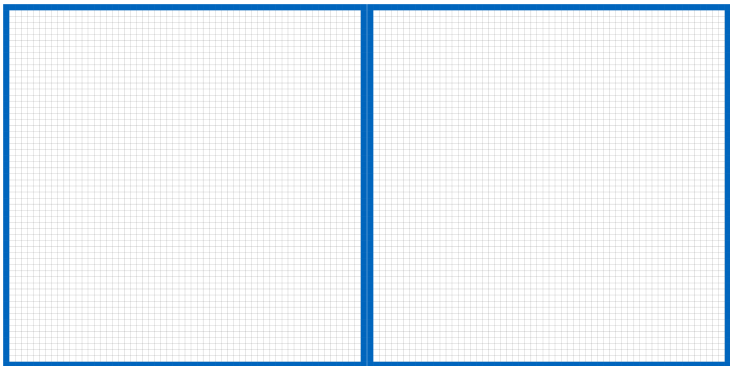
## Úloha

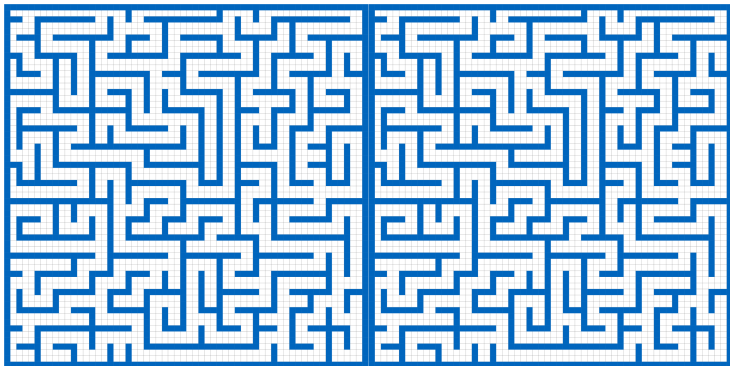
- Vstup je 2D pole, obsahuje 0 a 1 (1=zeď, 0=volný prostor) a souřadnice  $(x_0, y_0)$
- Cílem je vyplnit všechny prázdné buňky dosažitelné ze startu

## Floodfill

- Vlož  $(x_0, y_0)$  na zásobník
- Dokud není zásobník prázdný:
  - Vezmi prvek  $(x_i, y_i)$  ze zásobníku (pop), označ jeho pozici za obsazenou
  - Vlož do zásobníku všechny sousedy  $(x_i, y_i)$ , kteří jsou v poli a mají hodnotu 0
- Sousedi jsou buď prvky v 4-okolí nebo 8-okolí
- Úlohu lze řešit i s frontou
- Implementace na cvičeních







- Rekurzivní řešení lze počítat bez rekurze s využitím zásobníku
- Místo rekurzivního volání se ukládají na zásobník příslušné argumenty

```
1 def sumRecursively(x): #x is list
2     if len(x) == 0:
3         return 0
4     if len(x) == 1: #basic case
5         return x[0]
6     return sumRecursively(x[:-1]) + x[-1]
7
8 a = [2,4,6]
9 print( sumRecursively(a) )
```

12

- Sestavte pole, kdy tento způsob selže

- Rekurzivní řešení lze počítat bez rekurze s využitím zásobníku
- Místo rekurzivního volání se ukládají na zásobník příslušné argumenty

```
1 def sumRecursively(x): #x is list
2     if len(x) == 0:
3         return 0
4     if len(x) == 1: #basic case
5         return x[0]
6     return sumRecursively(x[:-1]) + x[-1]
7
8 a = [i for i in range(1100) ]
9 print( sumRecursively(a) )
```

```
    return sumRecursively(x[:-1]) + x[-1]
[Previous line repeated 995 more times]
File "../alpStack//sumRecursivelyLarge.py", line 2, in
    sumRecursively
    if len(x) == 0:
RecursionError: maximum recursion depth exceeded while calling a
    Python object
```

- Rekurzivní řešení lze počítat bez rekurze s využitím zásobníku
- Místo rekurzivního volání se ukládají na zásobník příslušné argumenty

```
1 def sumRecursivelyWithStack(x): #x is list
2     result = 0
3     stack = [ x ]
4     while len(stack) > 0:
5         a = stack.pop() #a is array
6         if len(a) == 0:
7             break
8         result += a.pop() #a.pop = last item of a
9         stack.append(a)
10    return result
11
12 a = [1,2]
13 print( sumRecursivelyWithStack(a) )
14
15 a = [i for i in range(1100) ]
16 print( sumRecursivelyWithStack(a) )
```

```
3
604450
```

- Program zjistí, jestli je výraz uzávorkován správně
- $a*(a+b)*(a-b) \rightarrow$  správně       $a[ \text{len}(b) ] \rightarrow$  nesprávně
- Postup: pokud přijde levá závorka, dáme ji na zásobník.
- Pokud přijde pravá závorka, vyjmemme ze zásobníku a zkontrolujeme, že párují
- Levých a pravých závorek musí být stejný počet, tj. na konci musí být zásobník prázdný

```
1 def checkParsSimple(x):
2     stack = []
3     for c in x:
4         if c=="(":
5             stack.append(c)
6         elif c == ")":
7             if len(stack) == 0:
8                 return False
9             leftPar = stack.pop()
10            if leftPar != "(":
11                return False
12    return len(stack) == 0
```

```
1 def checkParsSimple(x):
2     stack = []
3     for c in x:
4         if c=="(":
5             stack.append(c)
6         elif c == ")":
7             if len(stack) == 0:
8                 return False
9             leftPar = stack.pop()
10            if leftPar != "(":
11                return False
12    return len(stack) == 0
13
14 print( checkParsSimple("␣a*(a+b)␣" ) )
15 print( checkParsSimple("␣(1+(a+b))+(2-3)␣" ) )
16 print( checkParsSimple("␣1*(a+b␣" ) )
17 print( checkParsSimple("␣)a+b(␣" ) )
```

```
True
True
False
False
```



- Rozšíření předchozího programu o všechny typy závorek

```
1 def checkPars(x):
2     left = "({["
3     right = ")}]"
4     stack = []
5     for c in x:
6         if c in left:
7             stack.append(c)
8         else:
9             for i in range(len(right)):
10                if c == right[i]:
11                    #we expect left[i] in stack
12                    if len(stack)==0:
13                        return False
14                    lastLeft = stack.pop()
15                    if lastLeft != left[i]:
16                        return False
17    return len(stack) == 0
```

```
1 from checkPars import *
2
3 print(checkPars("_ (a+b) _") )
4 print(checkPars("_ (a+b)*[a-b]+{ } _") )
5 print(checkPars("_ (_[_]_) _") )
6 print(checkPars("_ () [_] _") )
7 print(checkPars("_ { _a*[1-2]+(3/4) _ _ } _") )
```

True  
True  
False  
False  
True

## Infixová notace:

- Operátor je mezi operandy
- Pořadí operací určují závorky (a priorita operátorů)
- $a*(b+c)$

## Postfixová notace:

- Operátor je po operandech
- Není třeba závorkování
- $a*(b+c) \rightarrow a \ b \ c \ + \ *$

## Prefixová notace:

- Operátor je před operandy
- Není třeba závorkování
- Též tzv. polská notace (autor J. Łukasiewicz)
- $a*(b+c) \rightarrow * \ a \ + \ b \ c$

Infix	Postfix
$12/4$	$12 \ 4 \ /$
$3 * 4 + 2$	$3 \ 4 \ * \ 2 \ +$
$3 * (4 - 2)$	$3 \ 4 \ 2 \ - \ *$
$(a - b) * c / d$	$a \ b \ - \ c \ * \ d \ /$

# Vyhodnocení výrazů

- Vyhodnocení postfixového výrazu s využitím zásobníku
- Procházíme řetězec (výraz) zleva doprava
- Pokud přijde číslo (operand), vložíme ho do zásobníku
- Pokud přijde operátor (uvažujeme  $+-*/$ ), vezmeme ze zásobníku dva poslední operandy, spočítáme výsledek a uložíme na zásobník

```

1 def evalPostfix(x):
2     """ x is correct postfix notation """
3     stack = []
4     for arg in x.split():
5         if arg == "+": #last+prev
6             stack.append( stack.pop() + stack.pop() )
7         elif arg == "-": #prev-last
8             stack.append( -stack.pop() + stack.pop() )
9         elif arg == "*": #prev*last
10            stack.append( stack.pop() * stack.pop() )
11        elif arg == "/": #prev/last
12            second = stack.pop()
13            first = stack.pop()
14            stack.append( first / second )
15        else:
16            stack.append(float(arg))
17    return stack.pop()

```

```
1 from evalPostfix import *
2
3 print( evalPostfix("3_4_") )
4 print( evalPostfix("3_4_+") )
5 print( evalPostfix("3_4_-") )
6 print( evalPostfix("3_4_/") )
7 print( evalPostfix("3_4_*_2_-_") ) #3*4 -2
8 print( evalPostfix("3_4_2_-_*_") ) #3*(4-2)
```

12.0

7.0

-1.0

0.75

10.0

6.0

- Chceme převést  $(a + b) * (c - d)$  na  $a b + c d - *$
- Procházíme zleva doprava, čísla předáváme na výstup
- Operátor se přidá na zásobník
- Je-li v zásobníku operátor s vyšší precedencí, přesuň tento nejdřív na výstup
- Otevírací závorka se přidá do zásobníku
- Pokud přijde uzavírací závorka, přesouvej ze zásobníku dokud se nenarazí na otevírací závorku (závorky na výstup nejdou)
- Přesuň ze zásobníku zbývající operátory
- `infixToPrefix.py`

```
1 from stack import Stack
2 from evalPostfix import *
3 from infixToPostfix import *
4
5 print(infixToPostfix("32+4"))
6 print(infixToPostfix("3*4-2"))
7 print(infixToPostfix("3*(4-2)"))
8 print(infixToPostfix("(62-32)*5/9"))
```

```
32 4 +
3 4 * 2 -
3 4 2 - *
62 32 - 5 * 9 /
```

- Provedeme kontrolu uzávorkování
- Pokud je v pořádku, převedeme infix na postfix
- Vyhodnotíme postfix

```
1 # Jan Kybic
2 from stack import Stack
3 from evalPostfix import *
4 from infixToPostfix import *
5
6 def evalInfix(s):
7     return evalPostfix(infixToPostfix(s))
8
9 print(evalInfix("32+4"))
10 print(evalInfix("3*4-2"))
11 print(evalInfix("3*(4-2)"))
12 print(evalInfix("(62-32)*5/9"))
```

```
36.0
10.0
6.0
16.666666666666668
```



- Vyhodnocení (Pythonovských) výrazů

```
1 print( eval("3*(5-1)" ) )
2 b = 10
3 print( eval("2**b" ) )
4 arr = [1,2,3]
5 print( eval("arr[0]*b" ) )
```

```
12
1024
10
```

- Při vyhodnocení se využívají globální proměnné a funkce

```
1 def add(a,b):
2     return a+b
3
4 x = 10
5 print( eval("add(x,1)" ) )
```

```
11
```

- Vstup do funkce eval() musí být syntakticky validní
- Nesmí nastat chyba běhu (runtime error)

```
1 a = 10
2 print( eval("a*b") )
```

```
    print( eval("a*b") )
File "<string>", line 1, in <module>
NameError: name 'b' is not defined
```