Programování 2

5. cvičení, 19-3-2024

Farní oznamy

1. Tento text a kódy ke cvičení najdete v repozitáři cvičení na https://github.com/PKvasnick/Programovani-2.
2. **Domácí úkoly**: 3 nové úkoly od minulého cvičení.
    1. Odstranění duplikátů ze seznamu
    2. Bucketsort
    3. Utřídění seznamu slov pomocí bucket sort

Dnešní program:

- Kvíz
- Python: jak rychlý je můj kód
- Domácí úkoly
- Opakování: Třídění
- Lineární spojovaný seznam

Na zahřátí

> *"Before software can be reusable it first has to be usable." – Ralph Johnson*

Opakovaná použitelnost kódu se přeceňuje. Největší využití mívají krátké kousky kódu. Velké knihovny sami po sobě dědíme zřídka.

Co dělá tento kód

```
ll = [1, 2, 3, 4]
for k in ll:
    k = k+1
ll
```

Pokud chcete změnit seznam, iterujte přes indexy a ne přes položky. Viz neměnné typy a pointry. Jaký je správný kód pro přičtení 1 ke všem položkám?

Jak rychlý je můj kód?

Jak můžeme měřit rychlost kódu?

Příklad: ciferný součet

```
def cif_soucet_1(cislo: int) -> int:
```

```
    """mod 10 dává číslici, div 10 zbytek"""
    soucet = 0
    while cislo:
        soucet += cislo % 10
        cislo //= 10
    return cislo

def cif_soucet_2(cislo: int) -> int:
    return sum(map(int, str(cislo)))

def cif_soucet_3(cislo: int) -> int:
    """mod 10 dává číslici, div 10 zbytek"""
    soucet = 0
    while cislo:
        cislo, zbytek = divmod(cislo, 10)
        soucet += zbytek
    return soucet
```

Který kód je rychlejší?

```
from timeit import Timer

def cif_soucet_1(cislo: int) -> int:
    """mod 10 dává číslici, div 10 zbytek"""
    soucet = 0
    while cislo:
        soucet += cislo % 10
        cislo //= 10
    return soucet

def cif_soucet_2(cislo: int) -> int:
    return sum(map(int, str(cislo)))

def cif_soucet_3(cislo: int) -> int:
    """mod 10 dává číslici, div 10 zbytek"""
    soucet = 0
    while cislo:
        cislo, zbytek = divmod(cislo, 10)
        soucet += zbytek
    return soucet

t1 = Timer("cif_soucet_1(123456789)", "from speedy import cif_soucet_1")
print(1, t1.timeit(number = 1000000))

t2 = Timer("cif_soucet_2(123456789)", "from speedy import cif_soucet_2")
print(2, t2.timeit(number = 1000000))

t3 = Timer("cif_soucet_3(123456789)", "from speedy import cif_soucet_3")
print(3, t3.timeit(number = 1000000))

------
1 1.4341183999786153
2 1.8329799000639468
3 1.358796100015752
```

```
1 1.184459500014782
2 1.6710229000309482
3 1.2787032999331132
```

Toto je dost nepohodlné, pohodlnější je použít *magic* `%%timeit` v IPythonu, tedy např.

- v Pythonské konzoli PyCharmu
- v jakémkoli typu Jupyter notebooku - Google Colab, JupyterLab a pod.

Čtení z konzole

Jedna varianta, kterou jsme dosud neměli:

```
from contextlib import suppress

with suppress(EOFError):
    while p := input():
        print(int(p))
```

Třídění

Z minula: Partial sort

No já úplně nechápu, co tam nechodilo...

```
from random import randint

data = [randint(1,10) for _ in range(5)]


def partial_sort(a: list[int], start: int = None, end: int = None) ->
tuple[list[int], int, int]:
    if start is None:
        start = 0
    if end is None:
        end = len(a)
    pivot = a[start]
    l_pivots = start
    r_pivots = start + 1
    p = r_pivots
    while p < end:
        if a[p] < pivot:
            hold = a[p]
            for i in range(p, l_pivots, -1):
                a[i] = a[i-1]
            a[l_pivots] = hold
            l_pivots += 1
            r_pivots += 1
        elif a[p] == pivot:
            hold = a[p]
```

```
            for i in range(p, r_pivots, -1):
                a[i] = a[i-1]
            a[r_pivots] = hold
            r_pivots += 1
        p += 1
    return a, l_pivots, r_pivots

print(partial_sort(data))
```

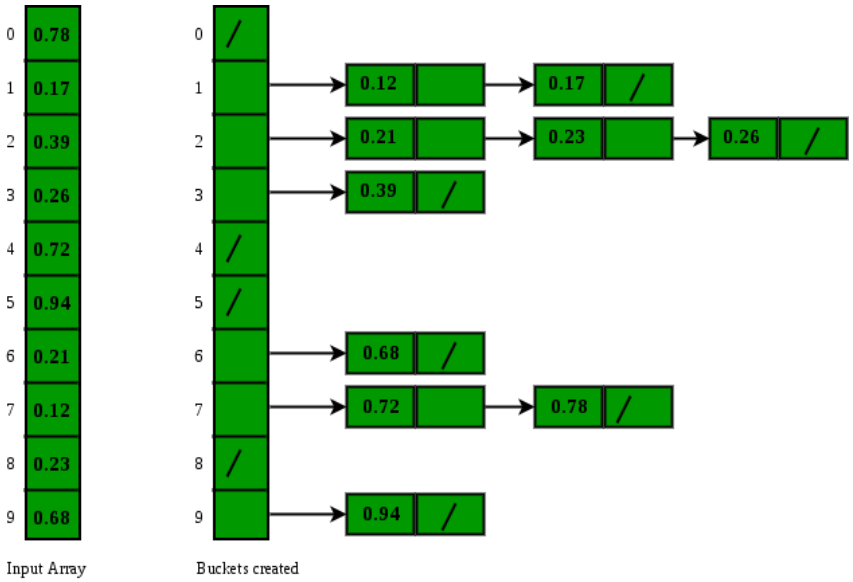
Mohli bychom něco vylepšit ?

Přerovnávání: bolestivé a pomalé.

Můžeme použít counting sort.

`Bucket sort`

- Nahrubo si setřídíme čísla do příhrádek
- Setřídíme obsah přihrádek
- Spojíme do výsledného seznamu



- Jednoduchý algoritmus
- Občas musíme popřemýšlet, jak vytvořit přihrádky (viz domácí úkol)
- Funguje krásně mysticky.

Varianta: deque

Namísto toho, abychom pokaždé sesypali kyblíky do pole, prostě průběžně dělíme věci do z kyblíků do jiných kyblíků. Potřebujeme ale zarážky, abychom věděli, co přišlo nově.

```
from collections import deque

words: list[str] = []

while "-end-" not in (line := input()):
    words.append(line.strip())
```

```
def get_bucket(word: str, order: int) -> int:
    if order < len(word):
        return ord(word[order]) - ord("a") + 1
    return 0


def queue_bucketsort(words: list[str]) -> list[str]:
    buckets = [deque([""]) for _ in range(ord("z") - ord("a") + 1)]
    # 1. Fill buckets
    buckets[0].extendleft(words)
    max_length = len(max(words, key=len))
    for order in reversed(range(max_length)):
        for bucket in buckets:
            while (w := bucket.popleft()) != "":
                buckets[get_bucket(w, order)].append(w)
        for bucket in buckets:
            bucket.append("")
        print(order)
        for i, bucket in enumerate(buckets):
            print(chr(i + ord("a")-1), ": ", *bucket)

    return [w for bucket in buckets for w in bucket if w != ""]


words = queue_bucketsort(words)
print(*words, sep="\n")
print("-end-")
```

Varianta: Counting sort

Quick sort

Souvisí s domácím úkolem o mediánu.

```
from random import randint


def quick_sort(b):
    if len(b) < 2:
        return b
    if len(b) == 2:
        return [min(b), max(b)]
    pivot = b[randint(0, len(b)-1)]
    lows = [x for x in b if x < pivot]
    pivots = [x for x in b if x == pivot]
    highs = [x for x in b if x > pivot]
    return quick_sort(lows) + pivots + quick_sort(highs)


data = [randint(1,100) for _ in range(10)]

print(data)
print(quick_sort(data))
```

Lineární spojovaný seznam

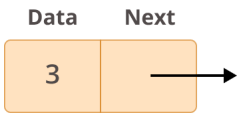
Proč se zabýváme datovými strukturami, když vše máme v Pythonu hotové?

Trénink mozku:

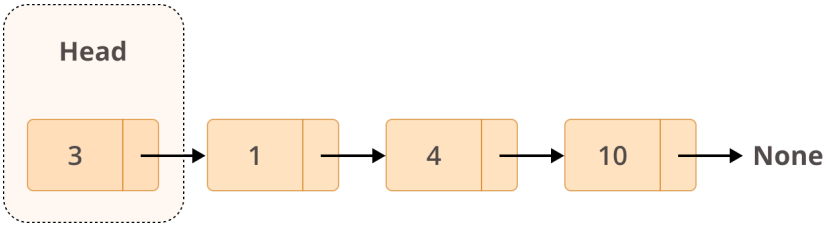
- Musíme umět rozebrat algoritmy na kolečka a šroubky
- Chceme identifikovat společné vzory v algoritmech
- Základní struktury umíme podrobně analyzovat.

Proč LSS?

"Převratný vynález": **spojení dat a strukturní informace**:

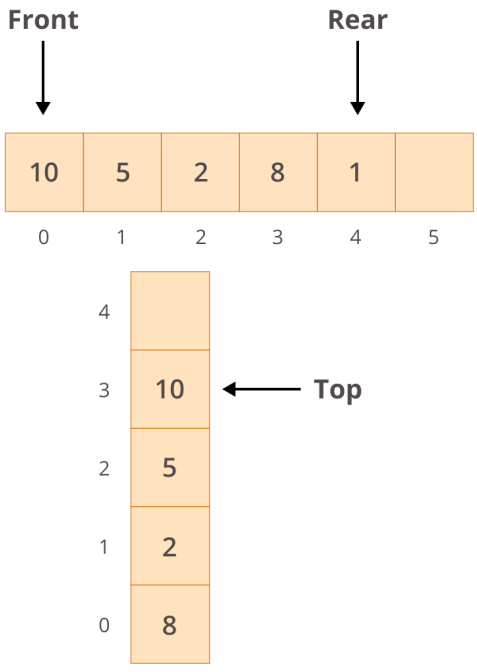


Takovéto jednotky pak umíme spojovat do větších struktur. LSS je nejjednodušší z nich.

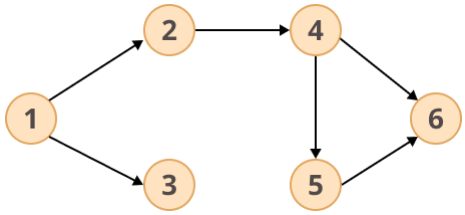


Aplikace:

- Fronty a zásobníky



- Grafy



Spojované seznamy v Pythonu

`list` v Pythonu je dynamické pole.

- Zanedbatelný rozdíl ve využití paměti
- Důležitá je časová efektivnost

Interface:

- přidávání prvků: `insert` a `append`
- odebírání prvků: `pop` a `remove`

Náročnost:

- na konci: O(1)
- kolem začátku: musíme nejdřív nalézt správné místo, O/(n)

Přístup k prvkům: `list` O(1), LSS O(n)

`collections.deque`

Pythonovská implementace nejbližší - co do API - k LSS.

- Rychlé přidávání/odebírání prvků na začátku/konci
- Přístup k prvkům přes začátek / konec

```
from collections import deque
llist = deque("abcde")
llist
deque(['a', 'b', 'c', 'd', 'e'])

llist.append("f")
llist
deque(['a', 'b', 'c', 'd', 'e', 'f'])

llist.appendleft("-")
llist
deque(['-', 'a', 'b', 'c', 'd', 'e', 'f'])

llist.pop()
'f'

llist.popleft()
'-'
llist
deque(['a', 'b', 'c', 'd', 'e'])

# off-brand API
llist[1]
```

```
'b'
```

`deque` podporuje také interface pole a umíme přistupovat k prvkům přes index. Tento přístup je rychlý.

Implementujeme spojovaný seznam

Existuje víc možností implementace - přes seznam, slovník nebo class.

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

Samotný LSS má jenom hlavu:

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

Přidáme metody, které nám seznam vytlačí na konzoli a při tom se naučíme seznamem procházet:

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

    def __str__(self):
        return self.data

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

    def __str__(self):
        node = self.head
        nodes = []
        while node is not None:
            nodes.append(node.value)
            node = node.next
        nodes.append("None")
        return " -> ".join(nodes)
```

Nyní můžeme nějaký seznam opravdu vytvořit:

```
# Importujeme kód do IPythonové konzole jako modul:
>>> from linked_list1 import * # Obecně ne-ne, ale pro náš malý kód OK.

>>> llist = LinkedList()
>>> str(llist)
None

>>> first_node = Node("a")
```

```
>>> llist.head = first_node
>>> str(llist)
a -> None

>>> second_node = Node("b")
>>> third_node = Node("c")
>>> first_node.next = second_node
>>> second_node.next = third_node
>>> str(llist)
a -> b -> c -> None
```

Vylepšíme `__init__`, abychom mohli vytvářet seznam pohodlněji:

```
def __init__(self, values=None):
    self.head = None
    if values is not None:
        node = Node(value=nodes.pop(0))
        self.head = node
        for elem in values:
            node.next = Node(value=elem)
            node = node.next
```

Procházení seznamem

```
def __iter__(self):
    node = self.head
    while node is not None:
        yield node
        node = node.next
```

a vyzkoušíme:

```
>>> llist = LinkedList(["a", "b", "c", "d", "e"])
>>> str(llist)
a -> b -> c -> d -> e -> None

>>> for node in llist:
...     print(node)
a
b
c
d
e
```

Vkládání prvků do seznamu

- add_first, add_last

```
def add_first(self, node):
    node.next = self.head
    self.head = node
```

```
def add_last(self, node):
    if self.head is None:
        self.head = node
        return
    for current_node in self:
        pass
    current_node.next = node
```

a vyzkoušíme:

```
>>> llist = LinkedList()
>>> str(llist)
None

>>> llist.add_first(Node("b"))
>>> str(llist)
b -> None

>>> llist.add_first(Node("a"))
>>> str(llist)
a -> b -> None
```

```
>>> llist = LinkedList(["a", "b", "c", "d"])
>>> str(llist)
a -> b -> c -> d -> None

>>> llist.add_last(Node("e"))
>>> str(llist)
a -> b -> c -> d -> e -> None

>>> llist.add_last(Node("f"))
>>> str(llist)
a -> b -> c -> d -> e -> f -> None
```

- add_after, add_before

  Musíme nejdřív nalézt, kam prvek vložit, a přitom uvážit, že umíme seznamem procházet pouze jedním směrem.

```
def add_after(self, target_node_value, new_node):
    if self.head is None:
        raise Exception("List is empty")

    for node in self:
        if node.value == target_node_value:
            new_node.next = node.next
            node.next = new_node
            return

    raise Exception("Node with value '%s' not found" % target_node_value)
```

```
>>> llist = LinkedList()
>>> llist.add_after("a", Node("b"))
```

```
Exception: List is empty

>>> llist = LinkedList(["a", "b", "c", "d"])
>>> str(llist)
a -> b -> c -> d -> None

>>> llist.add_after("c", Node("cc"))
>>> str((llist)
a -> b -> c -> cc -> d -> None

>>> llist.add_after("f", Node("g"))
Exception: Node with value 'f' not found
```

```
def add_before(self, target_node_value, new_node):
    if self.head is None:
        raise Exception("List is empty")

    if self.head.value == target_node_value:
        return self.add_first(new_node)

    prev_node = self.head
    for node in self:
        if node.value == target_node_value:
            prev_node.next = new_node
            new_node.next = node
            return
        prev_node = node

    raise Exception("Node with value '%s' not found" % target_node_value)
```

```
llist = LinkedList()
>>> llist.add_before("a", Node("a"))
Exception: List is empty

>>> llist = LinkedList(["b", "c"])
>>> str(llist)
b -> c -> None

>>> llist.add_before("b", Node("a"))
>>> str(llist)
a -> b -> c -> None

>>> llist.add_before("b", Node("aa"))
>>> llist.add_before("c", Node("bb"))
>>> str(llist)
a -> aa -> b -> bb -> c -> None

>>> llist.add_before("n", Node("m"))
Exception: Node with value 'n' not found
```

Odstraňujeme prvky

```
def remove_node(self, target_node_value):
    if self.head is None:
        raise Exception("List is empty")

    if self.head.value == target_node_value:
        self.head = self.head.next
        return

    previous_node = self.head
    for node in self:
        if node.value == target_node_value:
            previous_node.next = node.next
            return
        previous_node = node

    raise Exception("Node with value '%s' not found" % target_node_value)
```