Programování 2

6. cvičení, 23-3-2023

tags: `Programovani 2`, `čtvrtek 1`, `čtvrtek 2`

Farní oznamy

1. Tento text a kódy ke cvičení najdete v repozitáří cvičení na https://github.com/PKvasnick/Programovani-2.
2. **Domácí úkoly**:
   - 1 lehká, dvě těžší.
   - Omlouvám se za špatně zadaný domácí úkol, který jsem musel nahradit jiným.

Dnešní program:

- Kvíz a jazykové okénko
- Třídění (pořád): Halda neboli heap
- Pokračování: Lineární spojovaný seznam
- Varianty LSS: zásobník, fronta, cyklický zásobník, dvojitě spojovný seznam

---

Na zahřátí

> Make it work, then make it better.

Představte si, že svůj kód tesáte do kamene. Každé písmenko musíte pracně vyrazit do kamene, takže každé zbytečné písmenko je zbytečná práce navíc. Co pak ale vytesáte, prožije staletí.

Co dělá tento kód

```
1 def check(s:str) -> tuple[bool, bool, bool]:
2     return s.isnumeric(), s.isdigit(), s.isdecimal()
3 
4 check("V")
5 check("5")
6 check("-5")
```

Těmto metodám je lepší se vyhýbat, protože často dávají neintuitivní výsledky.

Rekurze

O tomto mluvíte na přednáškách, a tak si také dáme něco rekurzivního a budeme v následujících cvičeních přidávat.

Levenshteinova vzdálenost

Mějme dva znakové řetězce **a** a **b**. Počet záměn, přidání a vynechání jednotlivých znaků z **b**, abychom dostali **a** se nazývá *Levenshteinova vzdálenost* řeťězců **a** a **b**. Např. lev("čtvrtek", "pátek") je 4 (přidat čt, zaměnit "pá" za "vr"). Definice funkce je rekurzivní, takže implementace rekurzivní verze je triviální:

$$\operatorname{lev}(a,b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ \operatorname{lev}\left(\operatorname{tail}(a), \operatorname{tail}(b)\right) & \text{if } a[0] = b[0], \\ 1 + \min \begin{cases} \operatorname{lev}\left(\operatorname{tail}(a), b\right) \\ \operatorname{lev}\left(a, \operatorname{tail}(b)\right) \\ \operatorname{lev}\left(\operatorname{tail}(a), \operatorname{tail}(b)\right) \end{cases} & \text{otherwise,} \end{cases}$$

```python
def lev(s:str, t:str) -> int:
    """Calculate Levenshtein (edit) distance between strings s and t."""
    if (not s) or (not t):
        return len(s) + len(t)
    if s[0] == t[0]:
        return lev(s[1:], t[1:])
    return 1 + min(
        lev(s[1:], t),
        lev(s, t[1:]),
        lev(s[1:], t[1:])
    )
```

Problém s touto implementací je zjevný: každé volání může potenciálně vyvolat tři další. Pro delší řetězce to znamená, že takovýto výpočet je nepoužitelný. Začneme tím, že si to vyzkoušíme, a pak vyzkoušíme dva způsoby nápravy.

Budeme především potřebovat dva dostatečně dlouhé řetězce, např.

```python
s = "Démon kýs' škaredý, chvost vlečúc po zemi"
t = "Ko mne sa priplazil, do ucha šepce mi:"

k = 10

print(lev(s[:k],t[:k]))
```

Pro k>12 už výpočet trvá neúnosně dlouho. Pojďme se podívat na počet volání funkce. Pro tento účel použijeme *dekorátor* - tedy funkci, které pošleme naši funkci jako argument a ona vrátí modifikovanou funkci:

```python
# Dekorátor, počítající počet volání funkce
# Toto není úplně dokonalá implementace, protože nepřenáší signaturu funkce f.
def counted(f):
    def inner(s, t):
        inner.calls += 1 # inkrementujeme atribut
        return f(s, t)
    inner.calls = 0 # zřizujeme atribut funkce inner
    return(inner)


@counted
def lev(s:str, t:str) -> int:
    """Finds Levenshtein (edit) distance between two strings"""
    if (not s) or (not t):
        return len(s) + len(t)
    if s[0] == t[0]:
        return lev(s[1:], t[1:])
    return 1 + min(
        lev(s[1:], t),
        lev(s, t[1:]),
        lev(s[1:], t[1:])
    )


s = "Démon kýs' škaredý, chvost vlečúc po zemi"
t = "Ko mne sa priplazil, do ucha šepce mi:"

k = 10

print(lev(s[:k],t[:k]), lev.calls)
```

Vidíme, že počet volání funkce lev roste velice rychle.

Jeden způsob řešení je memoizace, o které jste mluvili na přednášce: zapamatujeme si hodnoty funkce, které jsme už počítali, a u těchto hodnot namísto volání funkce použijeme uloženou hodnotu. V Pythonu nemusíme psát vlastní memoizační funkci, stačí použít dekorátor:

```python
from functools import cache

# Dekorátor, počítající počet volání funkce
# Toto není úplně dokonalá implementace, protože nepřenáší signaturu funkce f.
def counted(f):
    def inner(s, t):
        inner.calls += 1 # inkrementujeme atribut
        return f(s, t)
    inner.calls = 0 # zřizujeme atribut funkce inner
    return(inner)


@counted
```

```
@cache
def lev(s:str, t:str) -> int:
    """Finds Levenshtein (edit) distance between two strings"""
    if (not s) or (not t):
        return len(s) + len(t)
    if s[0] == t[0]:
        return lev(s[1:], t[1:])
    return 1 + min(
        lev(s[1:], t),
        lev(s, t[1:]),
        lev(s[1:], t[1:])
    )


s = "Démon kýs' škaredý, chvost vlečúc po zemi"
t = "Ko mne sa priplazil, do ucha šepce mi:"

k = 10

print(lev(s[:k],t[:k]), lev.calls)
```

Počet volání je podstatně menší a teď už dokážeme spočítat lev(s, t) pro podstatně delší s, t.

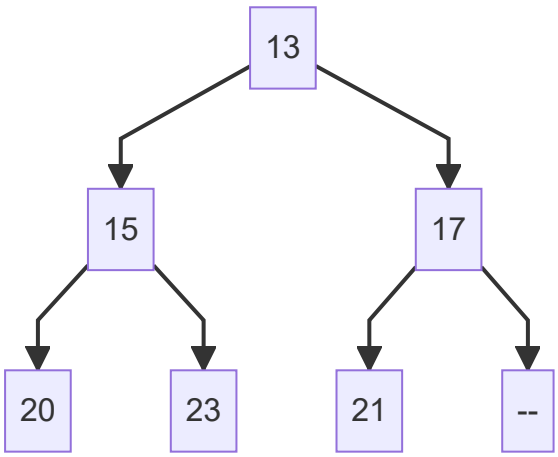
Problém u memoizace je, že nám keš může nekontrolovatelně růst. V praxi se ukazuje, že zpravidla můžeme výrazně omezit velikost keše beze stráty efektivnosti:

```
from functools import lru_cache

# Dekorátor, počítající počet volání funkce
# Toto není úplně dokonalá implementace, protože nepřenáší signaturu funkce f.
def counted(f):
    def inner(s, t):
        inner.calls += 1 # inkrementujeme atribut
        return f(s, t)
    inner.calls = 0 # zřizujeme atribut funkce inner
    return(inner)


@counted
@lru_cache(maxsize=1000)
def lev(s:str, t:str) -> int:
    """Finds Levenshtein (edit) distance between two strings"""
    if (not s) or (not t):
        return len(s) + len(t)
    if s[0] == t[0]:
        return lev(s[1:], t[1:])
    return 1 + min(
        lev(s[1:], t),
        lev(s, t[1:]),
        lev(s[1:], t[1:])
```

```
    )


s = "Démon kýs' škaredý, chvost vlečúc po zemi"
t = "Ko mne sa priplazil, do ucha šepce mi:"

k = 10

print(lev(s[:k],t[:k]), lev.calls)
```

Halda a heap sort

Halda, **min-heap** je *kompletní* binární strom, u kterého je hodnota ve vrcholu menší než hodnota ve vrcholech potomků. (podobně můžeme sestrojit i max-heap; rozdíl je tak triviální, že budeme mluvit o min-heap)



Implementace

Haldu můžeme lehko implementovat jako seznam, s následujícími pravidly:

- Hodnotu s indexem 0 nepoužijeme (získáme drobné zjednodušení kódu)
- Pro hodnotu na indexu **k** jsou potomci na indexech **2k** a **2k+1**
- Pro hodnotu na indexu **k** je rodič na indexu **k // 2**

Operace na min-haldě

get_min vrátí minimální prvek haldy. Složitost O(1)

pop_min odstraní z haldy minimální prvek, vrátí ho a přeorganizuje zbytek binárního stromu tak, aby zase byl haldou (**heapify**). Složitost O(n) (O(1) pro získání minimálního prvku, O(log n)) pro heapify).

add vloží do haldy novou hodnotu. Hodnotu přidáváme na konec a voláme **heapify**. Složitost O(log n)
.

```
# simplistic heap implementation
from random import randint

```

```
4
5  def add(h:list[int], x:int) -> None:
6      """Add x to the heap"""
7      h.append(x)
8      j = len(h)-1
9      while j > 1 and h[j] < h[j//2]:
10         h[j], h[j//2] = h[j//2], h[j]
11         j //= 2
12
13
14 def pop_min(h: list[int]) -> int:
15     """remove minimum element from the heap"""
16     if len(h) == 1: # empty heap
17         return None
18     result = h[1]   # we have the value, but have to tidy up
19     h[1] = h.pop()  # pop the last value and find a place for it
20     j = 1
21     while 2*j < len(h):
22         n = 2 * j
23         if n < len(h) - 1:
24             if h[n + 1] < h[n]:
25                 n += 1
26         if h[j] > h[n]:
27             h[j], h[n] = h[n], h[j]
28             j = n
29         else:
30             break
31     return result
32
33
34 def main() -> None:
35     heap = [None]  # no use for element 0
36     for i in range(10):
37         add(heap, randint(1, 100))
38         print(heap)
39     for i in range(len(heap)):
40         print(pop_min(heap))
41         print(heap)
42
43
44 if __name__ == '__main__':
45     main()
```

V Pythonu máme k dispozici modul *heapq*, který obslouží haldu za nás.

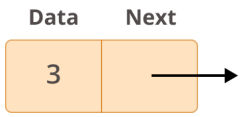
```
1  # Python3 program to demonstrate working of heapq
2
3  from heapq import heapify, heappush, heappop
4
5  # Creating empty heap
6  heap = []
```

```
heapify(heap)

# Adding items to the heap using heappush function
heappush(heap, 10)
heappush(heap, 30)
heappush(heap, 20)
heappush(heap, 400)

# printing the value of minimum element
print("Head value of heap : "+str(heap[0]))

# printing the elements of the heap
print("The heap elements : ")
for i in heap:
    print(i, end = ' ')
print("\n")

element = heappop(heap)

# printing the elements of the heap
print("The heap elements : ")
for i in heap:
    print(i, end = ' ')


```

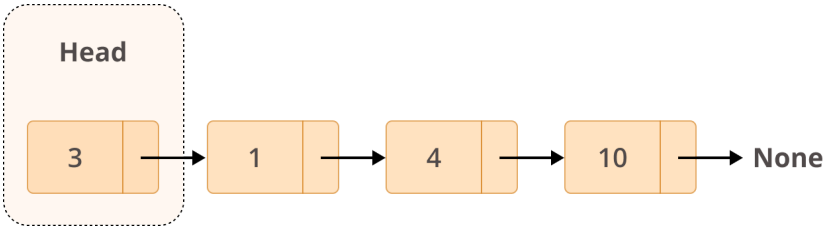
Aplikace: spojování setříděných seznamů.

Lineární spojovaný seznam

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

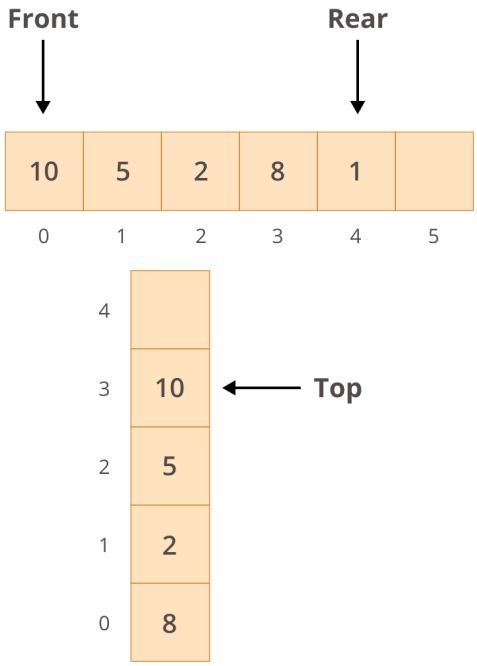


Takovéto jednotky pak umíme spojovat do větších struktur. LSS je nejjednodušší z nich.



Aplikace:

- Fronty a zásobníky



Spojované seznamy v Pythonu

`list` v Pythonu je dynamické pole

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

`collections.deque` je implementace fronty se dvěma konci.

- `append` / `appendleft`
- `pop` / `popleft`

Implementujeme spojovaný seznam

Spojovaný seznam s hlavou (kód v repozitáři, `code/Ex6/simply_linked_list1.py`)

```
1  # Simple linked list
2
3  class Node:
4      def __init__(self, value):
5          """Polozku inicializujeme hodnotou value"""
6          self.value = value
7          self.next = None
8
9      def __repr__(self):
10         """Reprezentace objektu na Pythonovske konzoli"""
11         return str(self.value)
12
13
```

```
class LinkedList:
    def __init__(self, values = None):
        """Spojovany seznam volitelne inicializujeme seznamem hodnot"""
        if values is None:
            self.head = None
            return
        self.head = Node(values.pop(0)) # pop vrati a odstrani hodnotu z values
        node = self.head
        for value in values:
            node.next = Node(value)
            node = node.next

    def __repr__(self):
        """Reprezentace na Pythonovske konzoli:
        Hodnoty spojene sipkami a na konci None"""
        values = []
        node = self.head
        while node is not None:
            values.append(str(node.value))
            node = node.next
        values.append("None")
        return " -> ".join(values)

    def __iter__(self):
        """Iterator prochazejici _hodnotami_ seznamu,
        napr. pro pouziti v cyklu for"""
        node = self.head
        while node is not None:
            yield node.value
            node = node.next

    def add_first(self, node):
        """Prida polozku na zacatek seznamu,
        tedy na head."""
        node.next = self.head
        self.head = node

    def add_last(self, node):
        """Prida polozku na konec seznamu."""
        p = self.head
        prev = None
        while p is not None:
            prev, p = p, p.next
        prev.next = node


```

Vkládání a odstraňování prvků

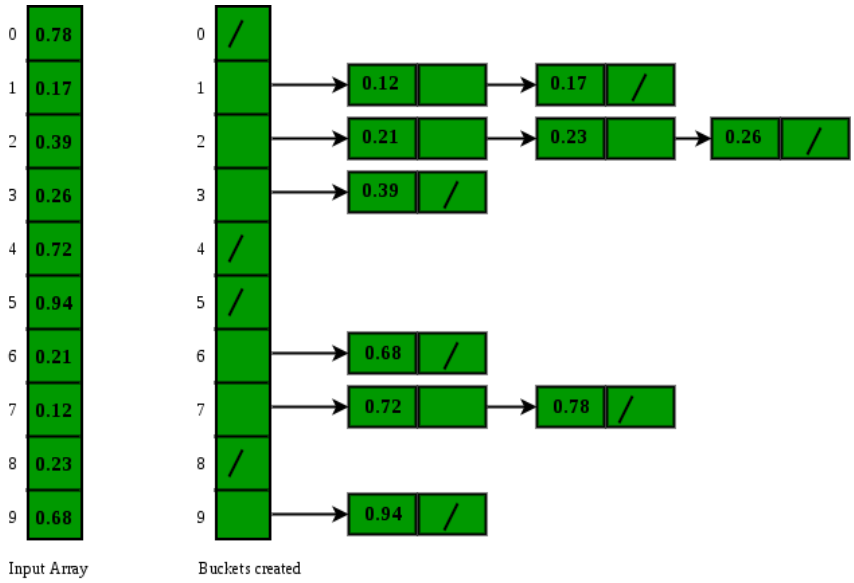
- `add_first`, `add_last`
- `add_before`, `add_after`
- `remove`

Třídění LSS

Utříděný seznam: `add` vloží prvek na správné místo

Jak utřídit již existující seznam?

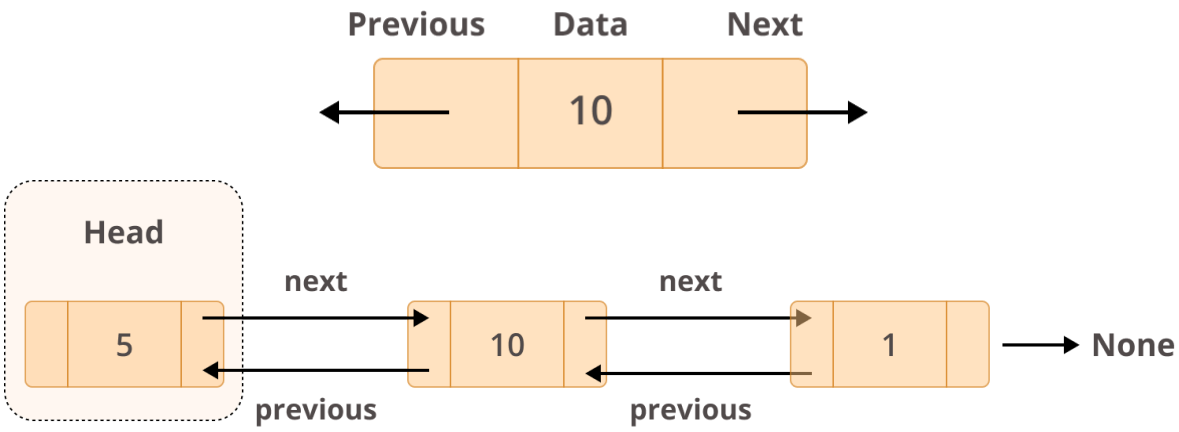
Bucket sort vyžaduje složitou datovou strukturu



Máme algoritmus, který by vystačil s průchody v jednom směru? Umíte ho implementovat?

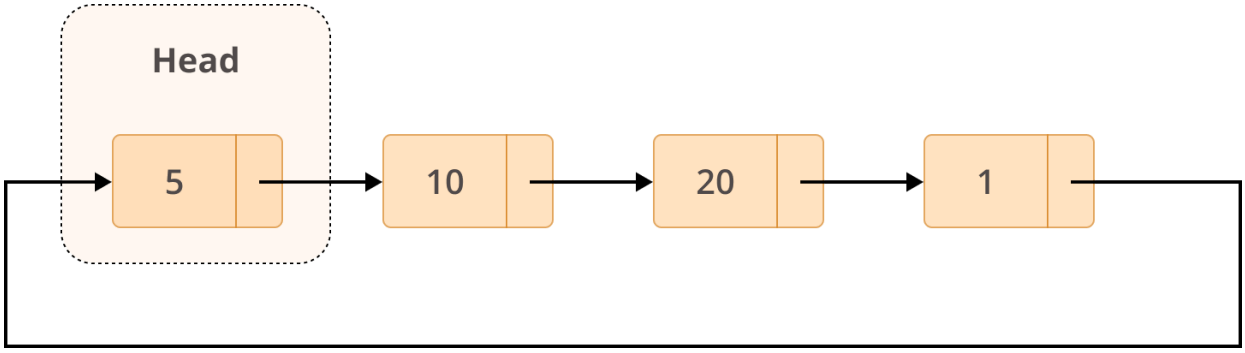
Varianty LSS

- **Dvojitě spojovaný seznam** - pro `deque`



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

- **Cyklický seznam**



Cyklickým seznamem můžeme procházet počínaje libovolným prvkem:

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

    def traverse(self, starting_point=None):
        if starting_point is None:
            starting_point = self.head
        node = starting_point
        while node is not None and (node.next != starting_point):
            yield node
            node = node.next
        yield node

    def print_list(self, starting_point=None):
        nodes = []
        for node in self.traverse(starting_point):
            nodes.append(str(node))
        print(" -> ".join(nodes))
```

Jak to funguje:

```
>>> circular_llist = CircularLinkedList()
>>> circular_llist.print_list()
None

>>> a = Node("a")
>>> b = Node("b")
>>> c = Node("c")
>>> d = Node("d")
>>> a.next = b
>>> b.next = c
```

```
>>> c.next = d
>>> d.next = a
>>> circular_llist.head = a
>>> circular_llist.print_list()
a -> b -> c -> d

>>> circular_llist.print_list(b)
b -> c -> d -> a

>>> circular_llist.print_list(d)
d -> a -> b -> c
```