Programování 2

6. cvičení, 25-3-2025

Farní oznamy

1. Tento text a kódy ke cvičení najdete v repozitáři cvičení na https://github.com/PKvasnick/Programovani-2.
2. Minulý týden jsme neměli ccvičení. Z domácích úkolů vidím, že podstatná část z vás si nastudovala materiál ke cvičení.
3. **Domácí úkoly**:
    - 2 úkoly na bubblesort
    - Přišlo méně řešení než obvykle, co připisuji zrušenému cvičení z minulého týdne.
    - K domácím úkolům se na chvilku vrátíme.

Dnešní program:

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

---

Na zahřátí

Čistý a srozumitelný kód je prostředek, ne cíl. Jasný kód signalizuje jasné myšlení a zvládnutí řemesla.

Vylepšete svůj kód, když přijdete na lepší nápad, jak to nebo ono udělat. Třeba dejte lepší jména proměnným. Pro vylepšení kódu poskytují moderní IDE řadu nástrojů, většinou je najdete pod položkou "Refactor".

Kód, který je špatně srozumitelný a křehký, se vylepšuje špatně.

Co dělá tento kód

```python
def check(s:str) -> tuple[bool, bool, bool]:
    return s.isnumeric(), s.isdigit(), s.isdecimal()

check("V")
check("5")
check("-5")
```

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

Rekurze

O tomto budete mluvit často, dokonce už na tomto cvičení, a tak si také dáme něco rekurzivního a budeme v následujících cvičeních (a domácích úkolech) 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:

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

Rekurzivní definice se implementuje přímočaře:

```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),      # přidání
        lev(s, t[1:]),      # vynechání
        lev(s[1:], t[1:])   # záměna
    )
```

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.

```
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:

```
# 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.

Dekorátor `counted` můžeme vylepšit pomocí dekorátoru `wraps` z modulu `functools`, aby byl použitelný pro funkce s různými signaturami:

```python
# Dekorátor, počítající počet volání funkce
from functools import wraps

def counted(f):
    @wraps(f)
    def inner(*args, **kwargs):
        inner.calls += 1 # inkrementujeme atribut
        return f(*args, **kwargs)
    inner.calls = 0 # zřizujeme atribut funkce inner
    return(inner)
```

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 wraps, cache

# Dekorátor, počítající počet volání funkce
def counted(f):
    @wraps(f)
    def inner(*args, **kwargs):
        inner.calls += 1 # inkrementujeme atribut
        return f(*args, **kwargs)
    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)
```

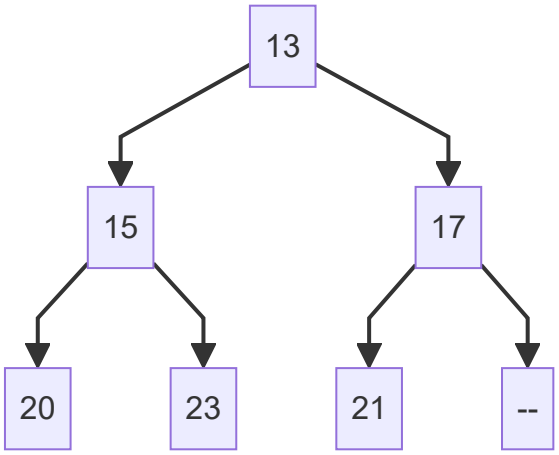
Počet volání je podstatně menší a teď už dokážeme spočítat lev(s, t) pro podstatně delší s, t.

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

```python
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** nebo **max-heap** je *kompletní* binární strom, u kterého je hodnota ve vrcholu menší než hodnota ve vrcholech potomků. (podobně můžeme sestrojit i max-heap; od jednoho ke druhému lehce přejdeme tak, že obrátíme znaménka hodnot ve vrcholech.



Implementace

Halda je abstraktní datová struktura. Při volbě implementace rozhodujeme, kde bude takováto struktura bydlet. Voleb je víc, například spojovaná dynamická struktura, hierarchie slovníků, matice atd.

Haldu můžeme lehko implementovat jako seznam, s následujícími pravidly:

- Vrchol stromu má index 0
- Pro hodnotu na indexu **k** jsou potomci na indexech **2k+1** a **2k+2**
- Pro hodnotu na indexu **k** je rodič na indexu **(k-1) // 2**

Pokud máme možnost, můžeme prvek s indexem 0 nechat prázdný, a začínat s indexem 1. Pak máme hezčí číslování:

- Vrchol stromu má index 1
- Pro hodnotu na indexu **k** jsou potomci na indexech **2k** a **2k+1**
- Pro hodnotu na indexu **k** je rodič na indexu **k // 2**

Hloubka stromu je $\log_2 k$, tedy strom je *velice mělký*.

Operace na min-haldě

get_min vrátí minimální prvek haldy, tedy kořen stromu. Složitost O(1)

pop_min odstraní z haldy minimální prvek, vrátí ho a přeorganizuje zbytek binárního stromu tak, aby zase byl haldou (**heapify**). Složitost $O(log_2 n$ (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) .

heapify je operace, která obnoví haldu po náhradě hodnoty v kořenu stromu. Hodnotu propagujeme směrem k listům stromu tak, že jí vyměňujeme za menší hodnotu z jejich potomků, až dokud nenajdeme uzel, kde jsou hodnoty u obou potomků větší anebo nedojdeme na kraj stromu (tedy k uzlu, který nemá dva potomky).

```
1  # simplistic heap implementation
2  from random import randint
3
4
5  def add(h:list[int], x:int) -> None:
```

```
    """Add x to the heap"""
    h.append(x)
    j = len(h)-1
    while j > 1 and h[j] < h[j//2]:
        h[j], h[j//2] = h[j//2], h[j]
        j //= 2


def pop_min(h: list[int]) -> int:
    """remove minimum element from the heap"""
    if len(h) == 1: # empty heap
        return None
    result = h[1]   # we have the value, but have to tidy up
    h[1] = h.pop()  # pop the last value and find a place for it
    j = 1
    while 2*j < len(h):
        n = 2 * j
        if n < len(h) - 1:
            if h[n + 1] < h[n]:
                n += 1
        if h[j] > h[n]:
            h[j], h[n] = h[n], h[j]
            j = n
        else:
            break
    return result


def main() -> None:
    heap = [None]  # no use for element 0
    for i in range(10):
        add(heap, randint(1, 100))
        print(heap)
    for i in range(len(heap)):
        print(pop_min(heap))
        print(heap)


if __name__ == '__main__':
    main()
```

Heapsort: setřídění seznamu na místě

Používáme *max-heap* a číslování od 0, tedy potomci uzlu k jsou 2k+1 a 2k+2, a předek uzlu k je (k-1) // 2. Max-heap proto, že chceme standardní způsob třídění.

1. Přidáme do heapu první prvek.
2. Přidáme následující prvek a podle potřeby ho propagujeme doleva.
3. Takto postupujeme, až je celý seznam přeorganizovaný na haldu.
4. Pak začneme haldu rozebírat: Odstraníme kořen, co je spolehlivě maximum, vyměníme ho za poslední prvek haldy, a odstraníme ho z haldy.

5. Haldu přeorganizujeme: nový prvek v 0 propagujeme nahoru na správné místo: zaměňujeme ho s potomkem, který má největší hodnotu. Nakonec máme opět na indexu 0 maximální prvek hlady.
6. Opakujeme od 4. kroku, až vyčerpáme celou haldu.

Udělali jsme O(n log n) operací a máme setříděný seznam.

```
# simplistic heap implementation
from random import randint


def print_heap(h: list[int], size: int) -> None:
    """Infix printout"""
    def to_string(h: list[int], index: int, size: int, level: int) ->
list[str]:
        rows = []
        if (child := 2*index + 1) < size:
            rows.extend(to_string(h, child, size, level + 1))
        rows.append(f"{' '*(level * 4)} -- {h[index]}")
        if (child := 2*index + 2) < size:
            rows.extend(to_string(h, child, size, level + 1))
        return rows

    print("\n".join(to_string(h, 0, size, 0)))


def heapify(h:list[int]) -> None:
    """Turn a list into a max-heap in-place"""
    for element in range(len(h)):
        p = element
        while (prev := (p-1) // 2) >= 0:
            if h[p] > h[prev]:
                h[p], h[prev] = h[prev], h[p]
                p = prev
            else:
                break
        print_heap(h, element+1)
        print(h[element+1:])


def heap_sort(h: list[int]) -> None:
    """Turne a heap into a sorted list"""
    for heap_size in reversed(range(1, len(h))):
        h[0], h[heap_size] = h[heap_size], h[0]
        p = 0
        while True:
            p_child = 2 * p + 1
            if p_child >= heap_size:
                break
            p_child2 = 2 * p + 2
            if p_child2 < heap_size and h[p_child2] > h[p_child]:
                p_child = p_child2
            if h[p] >= h[p_child]:
                break
            h[p], h[p_child] = h[p_child], h[p]
```

```
            p = p_child
        print_heap(h, heap_size)
        print(h[heap_size:])


def main() -> None:
    heap = [randint(1,100) for _ in range(10)]
    print(heap)
    heapify(heap)
    print(heap)
    heap_sort(heap)
    print(heap)


if __name__ == '__main__':
    main()
```

Výstup ilustruje budování haldy (heapify) a rozebírání hlady na setříděný seznam:

1. Heapify: seznam -> max- halda

```
[36, 83, 76, 71, 80, 9, 100, 5, 57, 80]
 -- 36
[83, 76, 71, 80, 9, 100, 5, 57, 80]
     -- 36
 -- 83
[76, 71, 80, 9, 100, 5, 57, 80]
     -- 36
 -- 83
     -- 76
[71, 80, 9, 100, 5, 57, 80]
         -- 36
     -- 71
 -- 83
     -- 76
[80, 9, 100, 5, 57, 80]
         -- 36
     -- 80
         -- 71
 -- 83
     -- 76
[9, 100, 5, 57, 80]
         -- 36
     -- 80
         -- 71
 -- 83
         -- 9
     -- 76
[100, 5, 57, 80]
         -- 36
     -- 80
         -- 71
 -- 100
         -- 9
     -- 83
```

```
          -- 76
[5, 57, 80]
              -- 5
          -- 36
     -- 80
          -- 71
 -- 100
          -- 9
     -- 83
          -- 76
[57, 80]
              -- 5
          -- 57
              -- 36
     -- 80
          -- 71
 -- 100
          -- 9
     -- 83
          -- 76
[80]
              -- 5
          -- 57
              -- 36
     -- 80
              -- 71
          -- 80
 -- 100
          -- 9
     -- 83
          -- 76
[]

```

2. max-halda -> setříděný seznam

```
[100, 80, 83, 57, 80, 9, 76, 5, 36, 71]
              -- 5
          -- 57
              -- 36
     -- 80
          -- 80
 -- 83
          -- 9
     -- 76
          -- 71
[100]
              -- 5
          -- 57
     -- 80
          -- 36
 -- 80
          -- 9
     -- 76
          -- 71
```

```
[83, 100]
          -- 5
     -- 57
          -- 36
 -- 80
          -- 9
     -- 76
          -- 71
[80, 83, 100]
          -- 5
     -- 57
          -- 36
 -- 76
          -- 9
     -- 71
[80, 80, 83, 100]
          -- 5
     -- 57
          -- 36
 -- 71
     -- 9
[76, 80, 80, 83, 100]
          -- 5
     -- 36
 -- 57
     -- 9
[71, 76, 80, 80, 83, 100]
     -- 5
 -- 36
     -- 9
[57, 71, 76, 80, 80, 83, 100]
     -- 5
 -- 9
[36, 57, 71, 76, 80, 80, 83, 100]
 -- 5
[9, 36, 57, 71, 76, 80, 80, 83, 100]
[5, 9, 36, 57, 71, 76, 80, 80, 83, 100]

```

Modul `heapq`

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

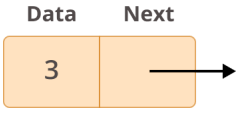
```python
# Python3 program to demonstrate working of heapq

from heapq import heapify, heappush, heappop

# Creating empty heap
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 = ' ')


```

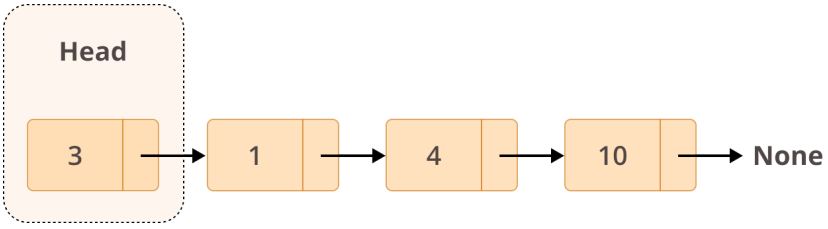
Aplikace: spojování setříděných seznamů (setříděný seznam je validní reprezentace hlady, tedy heapify na setříděném seznamu nic nestojí)

Lineární spojovaný seznam

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

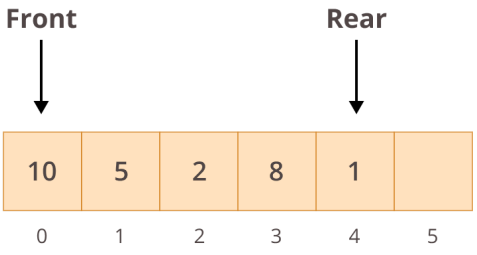


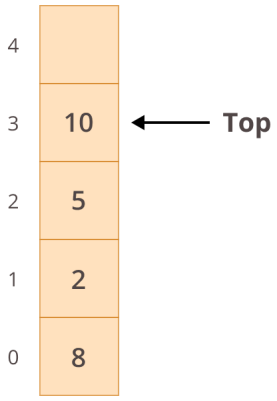
Takovéto jednotky pak umíme spojovat do větších struktur. LSS je nejjednodušší z nich.



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, implementovaný jako dynamická struktura (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
14 class LinkedList:
15     def __init__(self, values = None):
16         """Spojovany seznam volitelne inicializujeme seznamem hodnot"""
17         if values is None:
18             self.head = None
19             return
20         self.head = Node(values.pop(0)) # pop vrati a odstrani hodnotu z
   values
21         node = self.head
22         for value in values:
23             node.next = Node(value)
24             node = node.next
```

```
25
26      def __repr__(self):
27          """Reprezentace na Pythonovske konzoli:
28          Hodnoty spojene sipkami a na konci None"""
29          values = []
30          node = self.head
31          while node is not None:
32              values.append(str(node.value))
33              node = node.next
34          values.append("None")
35          return " -> ".join(values)
36
37      def __iter__(self):
38          """Iterator prochazejici _hodnotami_ seznamu,
39          napr. pro pouziti v cyklu for"""
40          node = self.head
41          while node is not None:
42              yield node.value
43              node = node.next
44
45      def add_first(self, node):
46          """Prida polozku na zacatek seznamu,
47          tedy na head."""
48          node.next = self.head
49          self.head = node
50
51      def add_last(self, node):
52          """Prida polozku na konec seznamu."""
53          p = self.head
54          prev = None
55          while p is not None:
56              prev, p = p, p.next
57          prev.next = node
58
59
```

Vkládání a odstraňování prvků

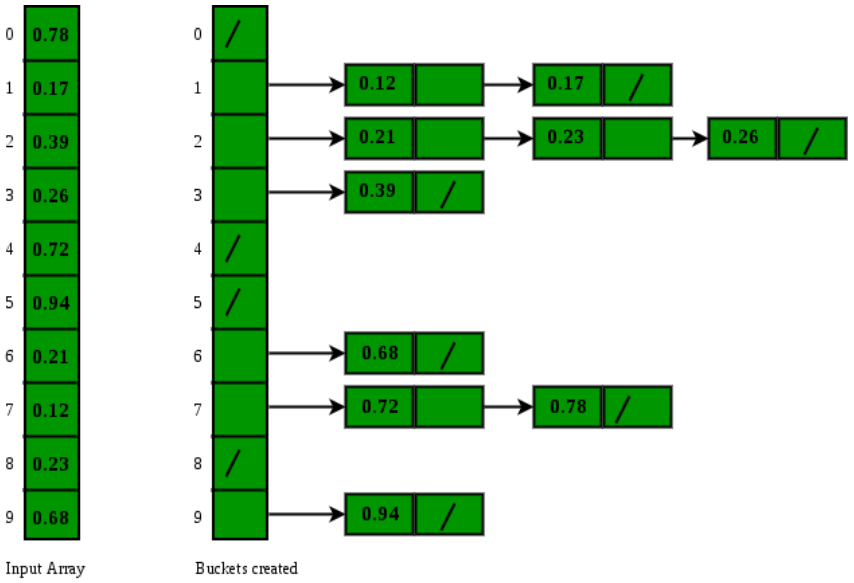
- `add_first`, `add_last`
- `add_before`, `add_after`
- `remove`

Třídění LSS

Utříděný seznam: `add` vloží prvek na správné místo

Jak utřídit již existující seznam?

Bucket sort vyžaduje složitou datovou strukturu

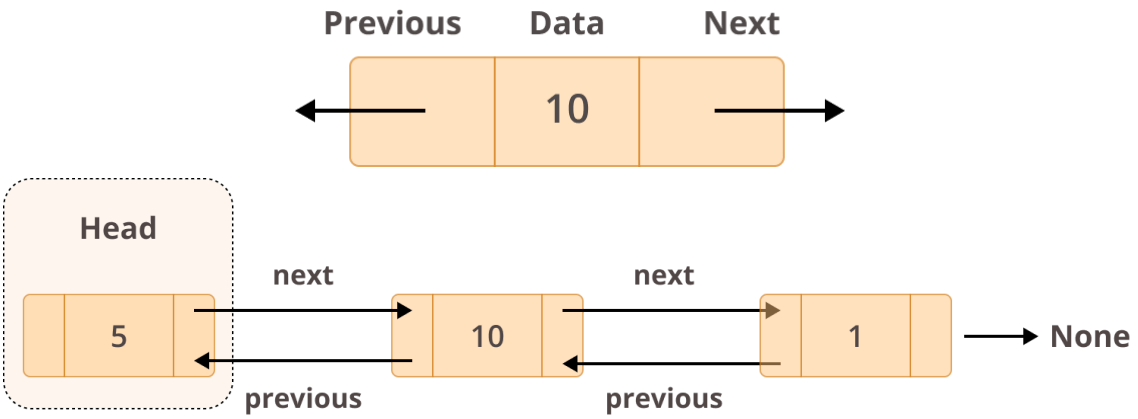


Heapsort potřebuje skákat z k na 2k (umíme, ale neradi děláme) a zpátky (neumíme, nebo jenom ztěžka)

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

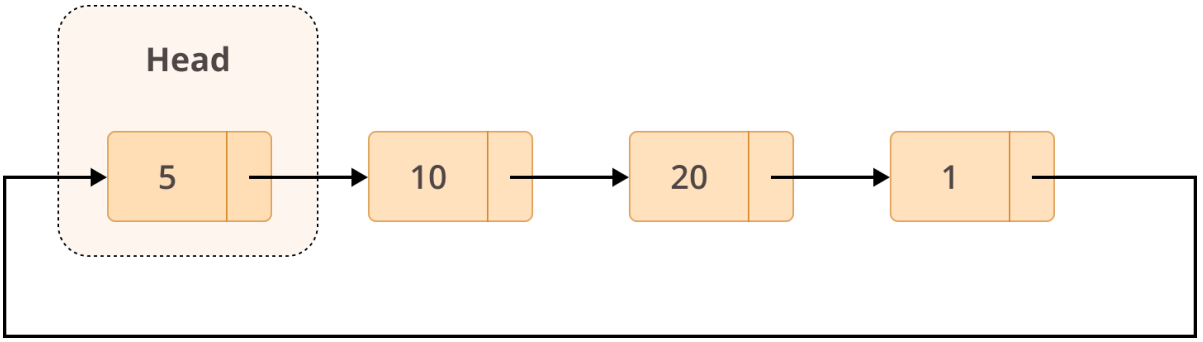
Varianty LSS

- **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:

```
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
```