

Programování 2

6. cvičení, 23-3-2023

tags: Programování 2, čtvrtek 1, čtvrtek 2

Farní oznamy

1. Tento text a kódy ke cvičení najdete v repozitáři 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, dvojité spojový 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* řetězců **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:

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

```
1 def lev(s:str, t:str) -> int:
2     """Calculate Levenshtein (edit) distance between strings s and t."""
3     if (not s) or (not t):
4         return len(s) + len(t)
5     if s[0] == t[0]:
6         return lev(s[1:], t[1:])
7     return 1 + min(
8         lev(s[1:], t),
9         lev(s, t[1:]),
10        lev(s[1:], t[1:]))
11
```

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.

```
1 s = "Démon kýs' škaredý, chvost vlečúc po zemi"
2 t = "Ko mne sa priplazil, do ucha šepce mi:"
3
4 k = 10
5
6 print(lev(s[:k], t[:k]))
```

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

```

1  # Dekorátor, počítající počet volání funkce
2  # Toto není úplně dokonalá implementace, protože nepřenáší signaturu funkce f.
3  def counted(f):
4      def inner(s, t):
5          inner.calls += 1 # inkrementujeme atribut
6          return f(s, t)
7      inner.calls = 0 # zřizujeme atribut funkce inner
8      return(inner)
9
10
11 @counted
12 def lev(s:str, t:str) -> int:
13     """Finds Levenshtein (edit) distance between two strings"""
14     if (not s) or (not t):
15         return len(s) + len(t)
16     if s[0] == t[0]:
17         return lev(s[1:], t[1:])
18     return 1 + min(
19         lev(s[1:], t),
20         lev(s, t[1:]),
21         lev(s[1:], t[1:])
22     )
23
24
25 s = "Démon kýs' škaredý, chvost vlečúc po zemi"
26 t = "ko mne sa priplazil, do ucha šepce mi:"
27
28 k = 10
29
30 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:

```

1  from functools import cache
2
3  # Dekorátor, počítající počet volání funkce
4  # Toto není úplně dokonalá implementace, protože nepřenáší signaturu funkce f.
5  def counted(f):
6      def inner(s, t):
7          inner.calls += 1 # inkrementujeme atribut
8          return f(s, t)
9      inner.calls = 0 # zřizujeme atribut funkce inner
10     return(inner)
11
12
13 @counted

```

```

14 @cache
15 def lev(s:str, t:str) -> int:
16     """Finds Levenshtein (edit) distance between two strings"""
17     if (not s) or (not t):
18         return len(s) + len(t)
19     if s[0] == t[0]:
20         return lev(s[1:], t[1:])
21     return 1 + min(
22         lev(s[1:], t),
23         lev(s, t[1:]),
24         lev(s[1:], t[1:])
25     )
26
27
28 s = "Démon kýs' škaredý, chvost vlečúc po zemi"
29 t = "Ko mne sa priplazil, do ucha šepce mi:"
30
31 k = 10
32
33 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 efektivity:

```

1  from functools import lru_cache
2
3  # Dekorátor, počítající počet volání funkce
4  # Toto není úplně dokonalá implementace, protože nepřenáší signaturu funkce f.
5  def counted(f):
6      def inner(s, t):
7          inner.calls += 1 # inkrementujeme atribut
8          return f(s, t)
9      inner.calls = 0 # zřizujeme atribut funkce inner
10     return(inner)
11
12
13 @counted
14 @lru_cache(maxsize=1000)
15 def lev(s:str, t:str) -> int:
16     """Finds Levenshtein (edit) distance between two strings"""
17     if (not s) or (not t):
18         return len(s) + len(t)
19     if s[0] == t[0]:
20         return lev(s[1:], t[1:])
21     return 1 + min(
22         lev(s[1:], t),
23         lev(s, t[1:]),
24         lev(s[1:], t[1:])

```

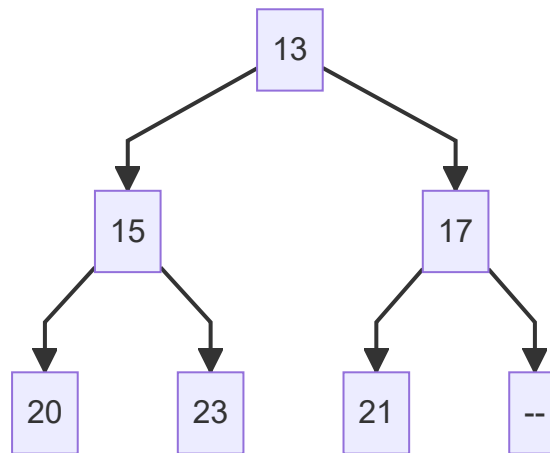
```

25     )
26
27
28     s = "Démon kýs' škaredý, chvost vlečúc po zemi"
29     t = "ko mne sa priplazil, do ucha šepce mi:"
30
31     k = 10
32
33     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 sestavit 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 $O(\log n)$

```

1     # simplistic heap implementation
2     from random import randint
3

```

```

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 = []

```

```

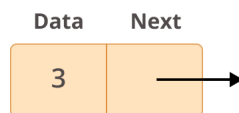
7  heapify(heap)
8
9  # Adding items to the heap using heappush function
10 heappush(heap, 10)
11 heappush(heap, 30)
12 heappush(heap, 20)
13 heappush(heap, 400)
14
15 # printing the value of minimum element
16 print("Head value of heap : "+str(heap[0]))
17
18 # printing the elements of the heap
19 print("The heap elements : ")
20 for i in heap:
21     print(i, end = ' ')
22 print("\n")
23
24 element = heappop(heap)
25
26 # printing the elements of the heap
27 print("The heap elements : ")
28 for i in heap:
29     print(i, end = ' ')
30
31

```

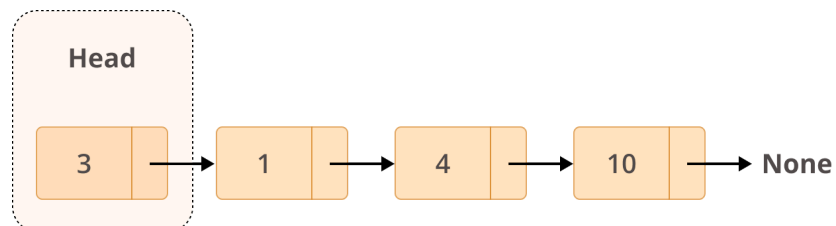
Aplikace: spojování setříděných seznamů.

Lineární spojovaný seznam

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

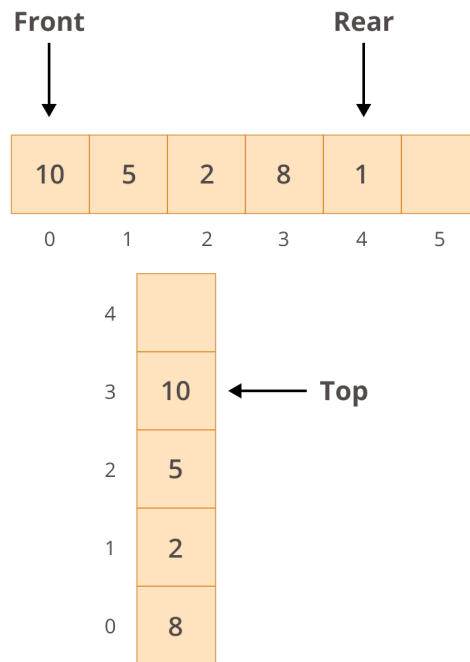


Takovéto jednotky pak umíme spojovat do větších struktur. LSS je nejjednodušší z nich.



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
```



```

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 spojeny 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

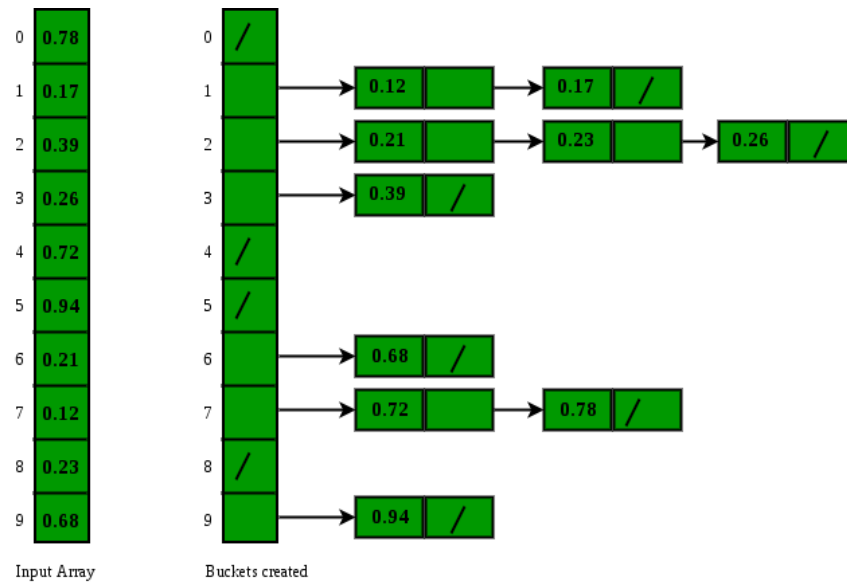
- `add_first`, `add_last`
- `add_before`, `add_after`
- `remove`

Třídění LSS

Utříděný seznam: `add` vloží prvek na správné místo

Jak utřídít již existující seznam?

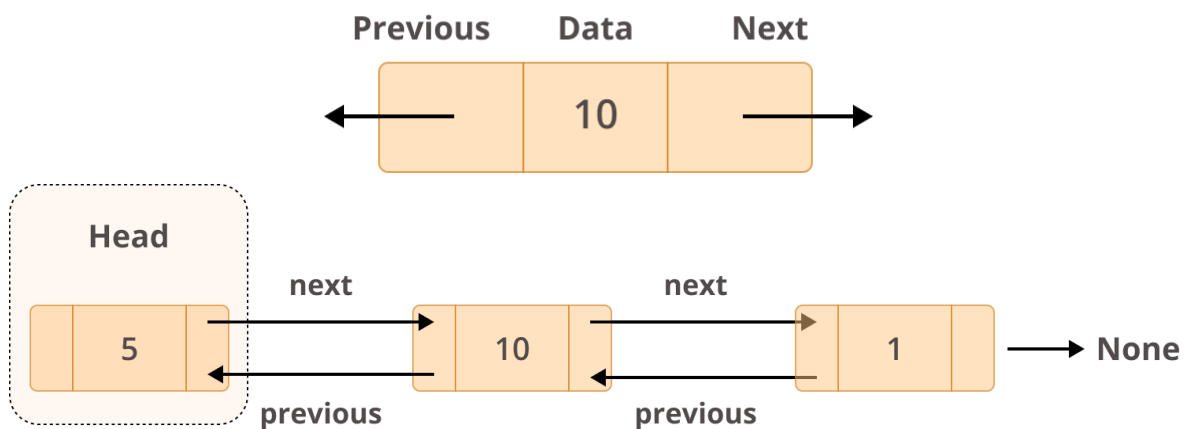
Bucket sort vyžaduje složitou datovou strukturu



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

Varianty LSS

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

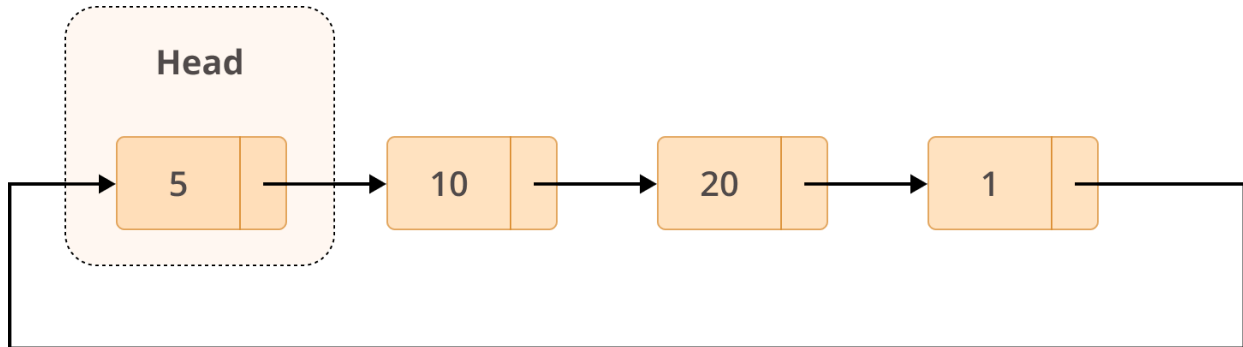


```

1 class Node:
2     def __init__(self, data):
3         self.data = data
4         self.next = None
5         self.previous = None

```

- Cyklický seznam



Cyklickým seznamem můžeme procházet počínaje libovolným prvkem:

```

1 class CircularLinkedList:
2     def __init__(self):
3         self.head = None
4
5     def traverse(self, starting_point=None):
6         if starting_point is None:
7             starting_point = self.head
8         node = starting_point
9         while node is not None and (node.next != starting_point):
10             yield node
11             node = node.next
12         yield node
13
14     def print_list(self, starting_point=None):
15         nodes = []
16         for node in self.traverse(starting_point):
17             nodes.append(str(node))
18         print(" -> ".join(nodes))

```

Jak to funguje:

```

1 >>> circular_llist = CircularLinkedList()
2 >>> circular_llist.print_list()
3 None
4
5 >>> a = Node("a")
6 >>> b = Node("b")
7 >>> c = Node("c")
8 >>> d = Node("d")
9 >>> a.next = b
10 >>> b.next = c

```

```
11 >>> c.next = d
12 >>> d.next = a
13 >>> circular_llist.head = a
14 >>> circular_llist.print_list()
15 a -> b -> c -> d
16
17 >>> circular_llist.print_list(b)
18 b -> c -> d -> a
19
20 >>> circular_llist.print_list(d)
21 d -> a -> b -> c
```