Datové struktury, základy složitosti

IB111 Základy programování

Tomáš Vojnar (na základě slajdů Nikoly Beneše)

17. října 2025

Efektivita algoritmů

Jak obrátit seznam?

`reverse.py`

- První myšlenka:
  - vytvoříme si nový prázdný seznam,
  - budeme postupně brát čísla z původního seznamu a budeme je vkládat na začátek nového seznamu.
- Jak vložíme číslo na *začátek* seznamu?
  - Zatím máme k dispozici jen indexování, `.append`, `.pop`, `len`.

Jak obrátit seznam?

reverse.py

- První myšlenka:
  - vytvoříme si nový prázdný seznam,
  - budeme postupně brát čísla z původního seznamu a budeme je vkládat na začátek nového seznamu.
- Jak vložíme číslo na *začátek* seznamu?
  - Zatím máme k dispozici jen indexování, `.append`, `.pop`, `len`.

```
def add_to_start(a_list: list[int],
                 value: int) -> None:
    for i, orig in enumerate(a_list):
        a_list[i] = value
        value = orig

    a_list.append(value)
```

Jak obrátit seznam?

reverse.py

```
def reverse1(my_list: list[int]) -> list[int]:
    result: list[int] = []
    for elem in my_list:
        add_to_start(result, elem)
    return result
```

- Jiná možnost?

Jak obrátit seznam?

reverse.py

```
def reverse1(my_list: list[int]) -> list[int]:
    result: list[int] = []
    for elem in my_list:
        add_to_start(result, elem)
    return result
```

- Jiná možnost?
  - Projdeme seznam pozpátku, přidáváme na *konec* nového.

```
def reverse2(my_list: list[int]) -> list[int]:
    result = []
    for i in range(len(my_list) - 1, -1, -1):
        result.append(my_list[i])
    return result
```

- Je nějaký rozdíl v rychlosti těchto dvou algoritmů?

Jak obrátit seznam?

reverse.py

```
def reverse1(my_list: list[int]) -> list[int]:
    result: list[int] = []
    for elem in my_list:
        add_to_start(result, elem)
    return result
```

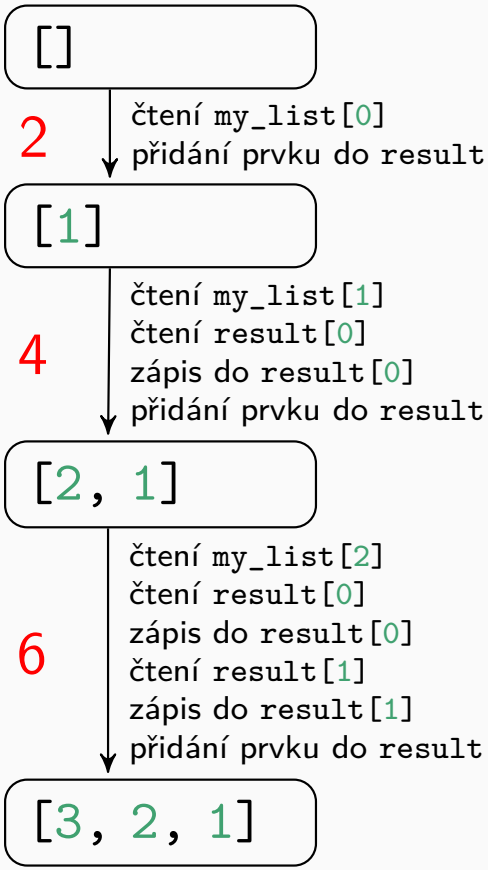
- Jiná možnost?
  - Projdeme seznam pozpátku, přidáváme na *konec* nového.

```
def reverse2(my_list: list[int]) -> list[int]:
    result = []
    for i in range(len(my_list) - 1, -1, -1):
        result.append(my_list[i])
    return result
```

- Je nějaký rozdíl v rychlosti těchto dvou algoritmů? ANO
  - První je *kvadratický*, druhý je *lineární*.

Počet kroků při obracení seznamu (zhruba)

`reverse1([1, 2, 3])`

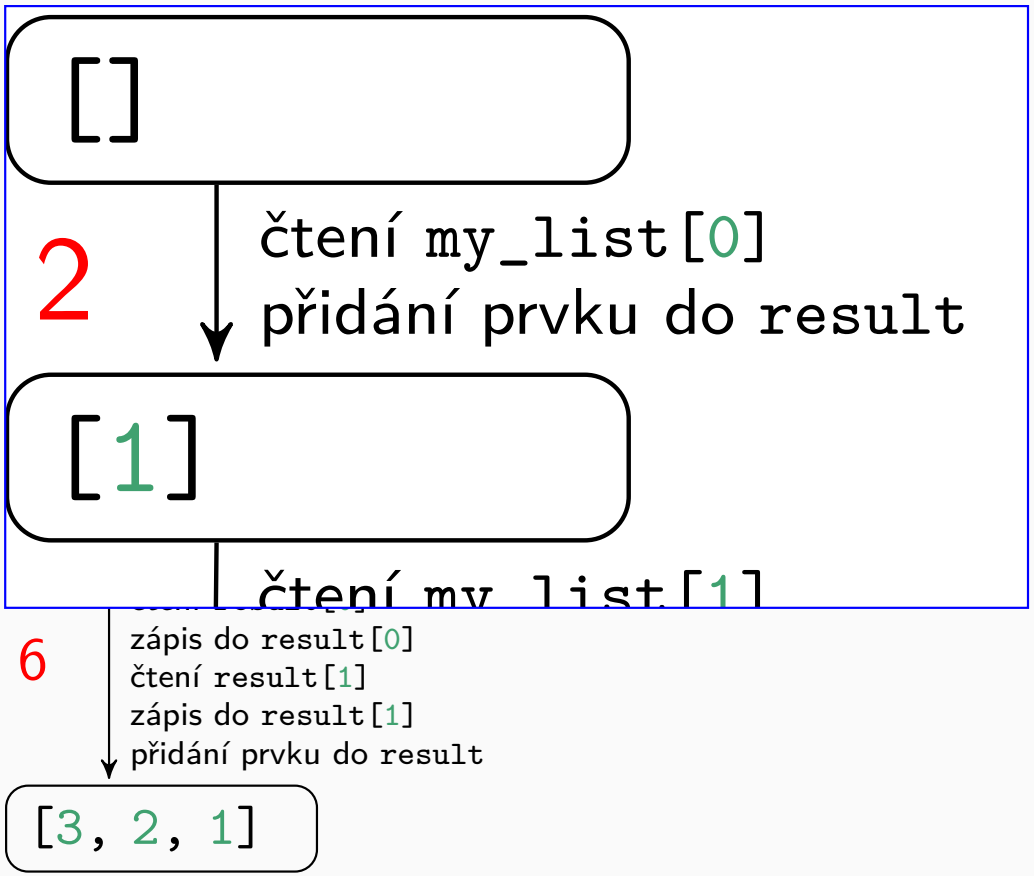


`reverse2([1, 2, 3])`

celkem $2 + 4 + 6 = 12$ kroků

Počet kroků při obracení seznamu (zhruba)

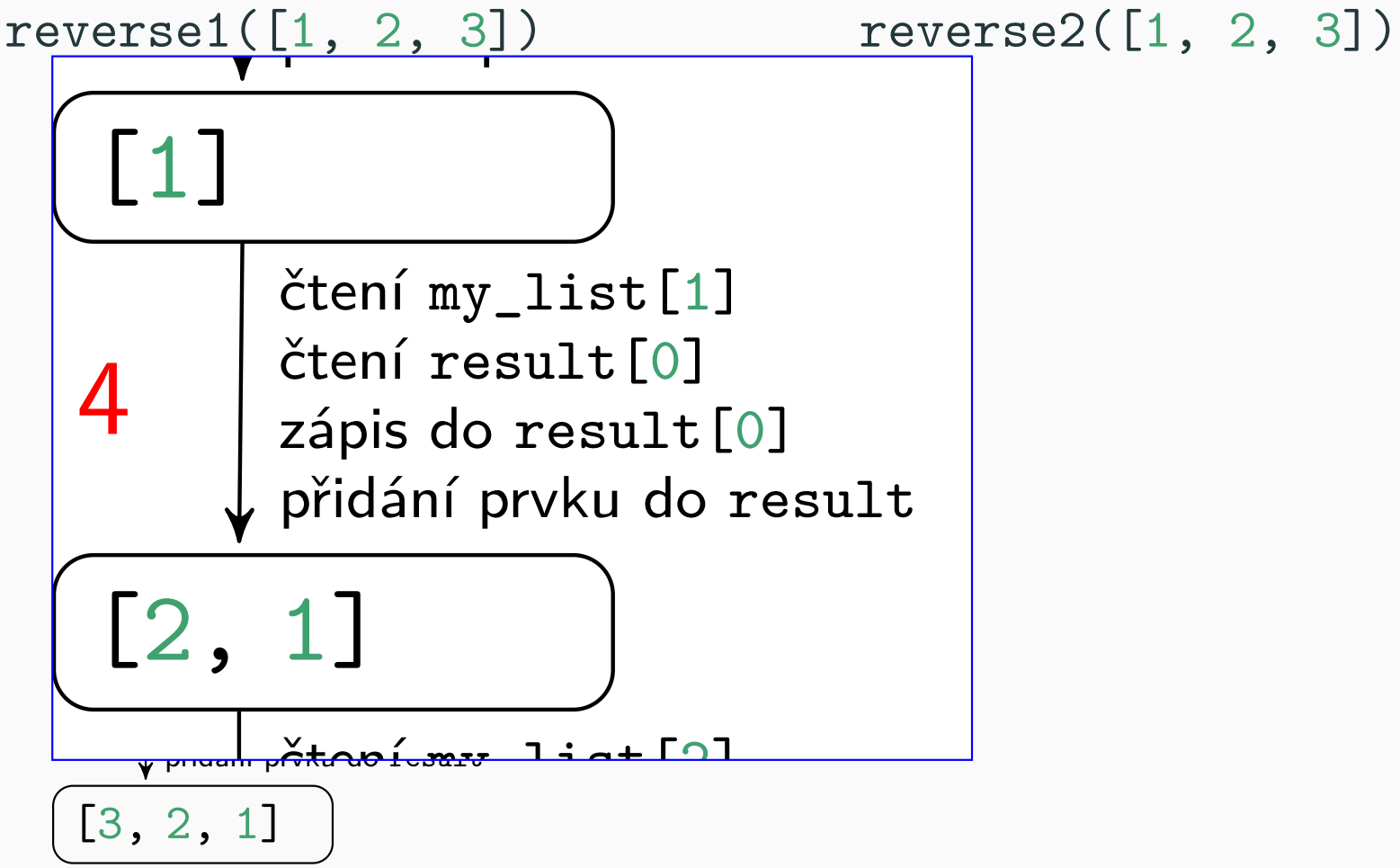
`reverse1([1, 2, 3])`



`reverse2([1, 2, 3])`

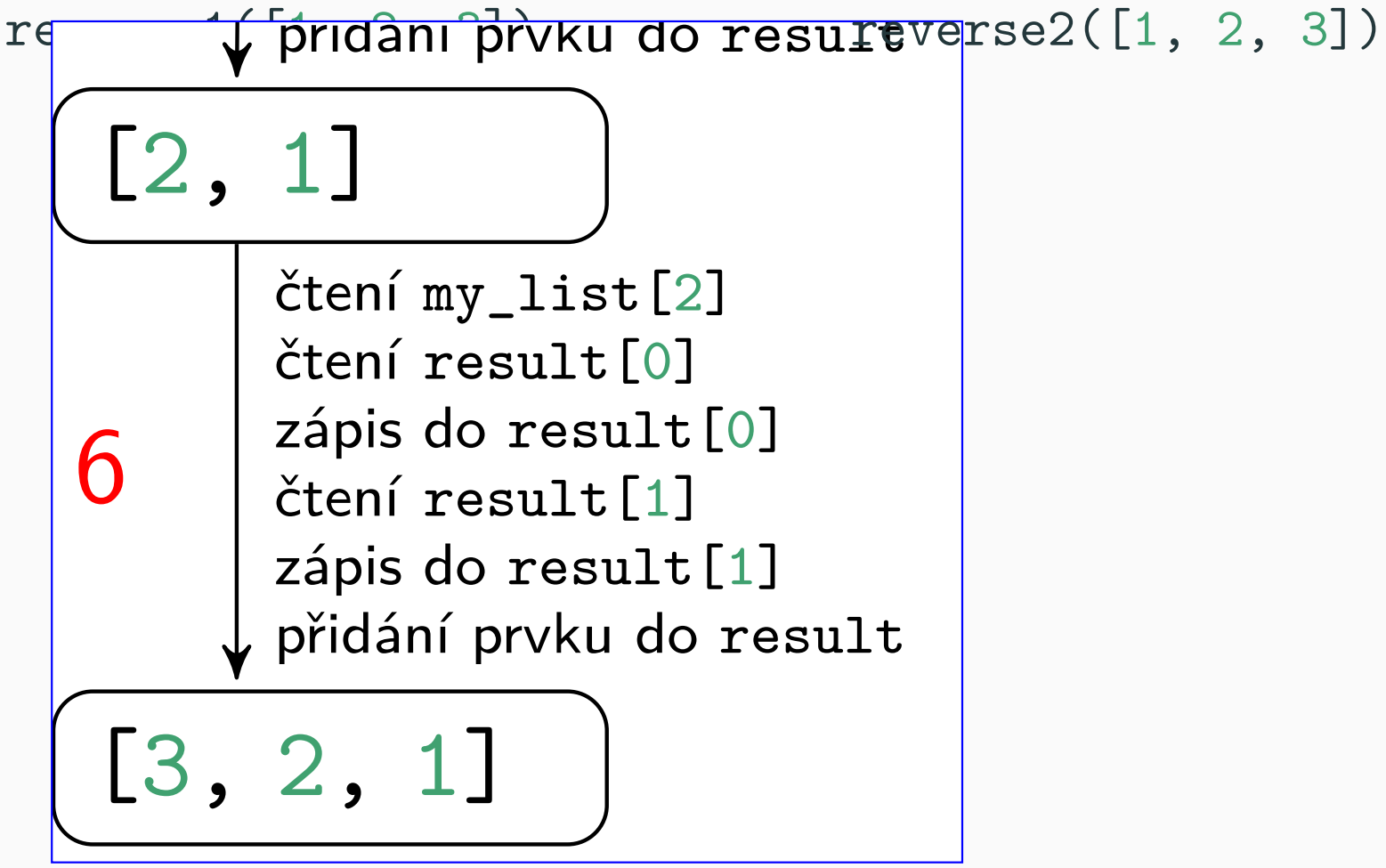
celkem $2 + 4 + 6 = 12$ kroků

Počet kroků při obracení seznamu (zhruba)



celkem $2 + 4 + 6 = 12$ kroků

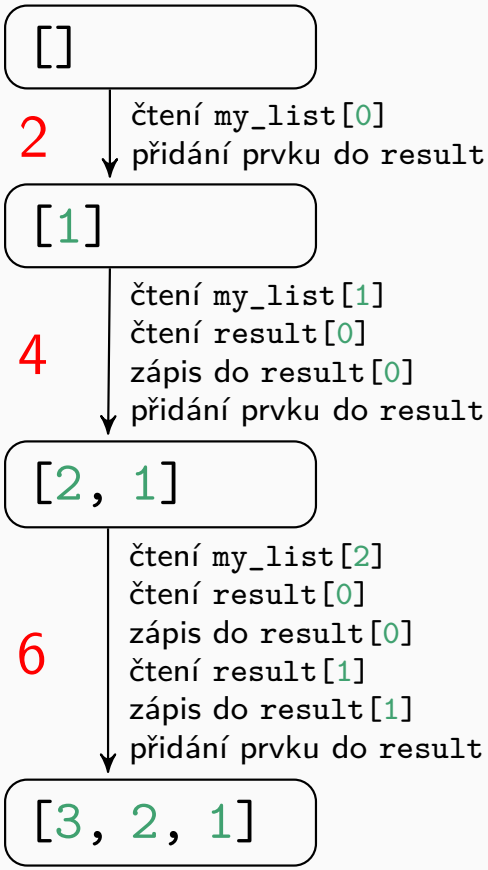
Počet kroků při obracení seznamu (zhruba)



celkem $2 + 4 + 6 = 12$ kroků

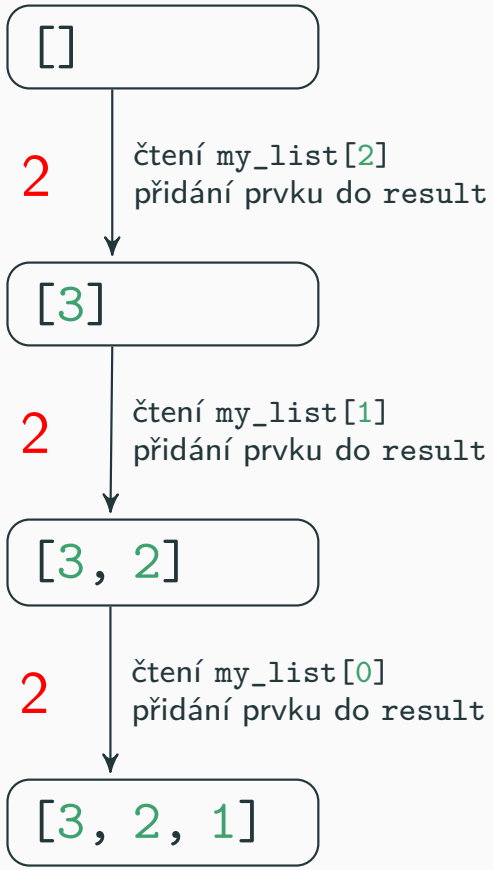
Počet kroků při obracení seznamu (zhruba)

`reverse1([1, 2, 3])`



celkem $2 + 4 + 6 = 12$ kroků

`reverse2([1, 2, 3])`



celkem $2 + 2 + 2 = 6$ kroků

Počet kroků při obracení seznamu (zhruba)

‘reverse1‘ – bereme prvky postupně, vkládáme je na začátek:

- pro seznam se třemi prvky: 12 kroků (seznamových operací),
- pro seznam se čtyřmi prvky: 20 kroků (seznamových operací),
- pro seznam s n prvky: $2 + 4 + \ldots + 2n = n^2 + n$,
- počet kroků každé iterace závisí (lineárně) na délce `result`.

‘reverse2‘ (bereme prvky odzadu, přidáváme je na konec):

- pro seznam se třemi prvky: 6 kroků (seznamových operací),
- pro seznam se čtyřmi prvky: 8 kroků (seznamových operací),
- pro seznam s n prvky: $2 + 2 + \ldots + 2 = 2n$,
- počet kroků každé iterace je nezávislý na délce `result`.

Složitost algoritmů se seznamy (zjednodušeně)

Časová složitost algoritmů se seznamy:

- funkce, která *počtu prvků vstupního seznamu* přiřazuje *počet kroků algoritmu*,
  - základní kroky se seznamy: indexování, `.append`, `.pop`, `len`,
  - měříme nejhorší případ (obvykle – lze i průměrný, nejlepší, …),
  - zajímá nás pouze rychlost růstu funkce.

Složitost algoritmů se seznamy (zjednodušeně)

Časová složitost algoritmů se seznamy:

- funkce, která *počtu prvků vstupního seznamu* přiřazuje *počet kroků algoritmu*,
  - základní kroky se seznamy: indexování, `.append`, `.pop`, `len`,
  - měříme nejhorší případ (obvykle – lze i průměrný, nejlepší, …),
  - zajímá nás pouze rychlost růstu funkce.

- Konstantní složitost:
  - počet kroků nezávislý na délce seznamu.
- Lineární složitost:
  - počet kroků lineárně závislý na délce seznamu,
  - tj. konstantní počet kroků *pro každý prvek seznamu*,
  - se stejnou konstantou pro každý prvek.
- Kvadratická složitost:
  - počet kroků kvadraticky závislý na délce seznamu,
  - tj. lineární počet kroků *pro každý prvek seznamu*.

Další jednoduché seznamové operace

`list_ops.py`

Obrácení seznamu „v místě“ (*inplace*, *in situ*)

- `reverse_inplace`,
- modifikace původního seznamu,
- časová složitost:

Další jednoduché seznamové operace

`list_ops.py`

Obrácení seznamu „v místě“ (*inplace*, *in situ*)

- `reverse_inplace`,
- modifikace původního seznamu,
- časová složitost: *lineární*.

Další jednoduché seznamové operace

`list_ops.py`

Obrácení seznamu „v místě“ (*inplace*, *in situ*)

- `reverse_inplace`,
- modifikace původního seznamu,
- časová složitost: *lineární*.

Vyhledání prvku v (neseřazeném) seznamu

- `contains`.
- vstup: seznam a hledané číslo,
- výstup: `True`/`False`, jestli je číslo v seznamu,
- časová složitost:

Další jednoduché seznamové operace

`list_ops.py`

Obrácení seznamu „v místě“ (*inplace*, *in situ*)

- `reverse_inplace`,
- modifikace původního seznamu,
- časová složitost: *lineární*.

Vyhledání prvku v (neseřazeném) seznamu

- `contains`.
- vstup: seznam a hledané číslo,
- výstup: `True`/`False`, jestli je číslo v seznamu,
- časová složitost: ,
  - závisí na tom, jestli je číslo v seznamu a kde,
  - nejhorší případ:

Další jednoduché seznamové operace

`list_ops.py`

Obrácení seznamu „v místě“ (*inplace*, *in situ*)

- `reverse_inplace`,
- modifikace původního seznamu,
- časová složitost: *lineární*.

Vyhledání prvku v (neseřazeném) seznamu

- `contains`.
- vstup: seznam a hledané číslo,
- výstup: `True`/`False`, jestli je číslo v seznamu,
- časová složitost: *lineární*,
  - závisí na tom, jestli je číslo v seznamu a kde,
  - nejhorší případ: číslo v seznamu není.

Další jednoduché seznamové operace

`list_ops.py`

Spojení dvou seznamů do jednoho (nového)

- `join`,
- vstupem dva seznamy, výstupem nový seznam,
  - prvky levého seznamu, následované prvky pravého seznamu,
- časová složitost:

Další jednoduché seznamové operace

`list_ops.py`

Spojení dvou seznamů do jednoho (nového)

- `join`,
- vstupem dva seznamy, výstupem nový seznam,
  - prvky levého seznamu, následované prvky pravého seznamu,
- časová složitost: *lineární*,
  - podrobněji:

Další jednoduché seznamové operace

`list_ops.py`

Spojení dvou seznamů do jednoho (nového)

- `join`,
- vstupem dva seznamy, výstupem nový seznam,
  - prvky levého seznamu, následované prvky pravého seznamu,
- časová složitost: *lineární*,
  - podrobněji: *lineární k součtu délek obou seznamů*,
  - levý seznam délky n, pravý délky m, složitost cca $n + m$.

Další jednoduché seznamové operace

`list_ops.py`

Spojení dvou seznamů do jednoho (nového)

- `join`,
- vstupem dva seznamy, výstupem nový seznam,
  - prvky levého seznamu, následované prvky pravého seznamu,
- časová složitost: *lineární*,
  - podrobněji: *lineární k součtu délek obou seznamů*,
  - levý seznam délky n, pravý délky m, složitost cca $n + m$.

Spojení dvou seznamů do jednoho – s modifikací levého

- `extend`,
- přidáme do levého seznamu všechny prvky z pravého,
- časová složitost:

Další jednoduché seznamové operace

`list_ops.py`

Spojení dvou seznamů do jednoho (nového)

- `join`,
- vstupem dva seznamy, výstupem nový seznam,
  - prvky levého seznamu, následované prvky pravého seznamu,
- časová složitost: *lineární*,
  - podrobněji: *lineární k součtu délek obou seznamů*,
  - levý seznam délky n, pravý délky m, složitost cca $n + m$.

Spojení dvou seznamů do jednoho – s modifikací levého

- `extend`,
- přidáme do levého seznamu všechny prvky z pravého,
- časová složitost: *lineární*,
  - podrobněji:

Další jednoduché seznamové operace

`list_ops.py`

Spojení dvou seznamů do jednoho (nového)

- `join`,
- vstupem dva seznamy, výstupem nový seznam,
  - prvky levého seznamu, následované prvky pravého seznamu,
- časová složitost: *lineární*,
  - podrobněji: *lineární k součtu délek obou seznamů*,
  - levý seznam délky n, pravý délky m, složitost cca $n + m$.

Spojení dvou seznamů do jednoho – s modifikací levého

- `extend`,
- přidáme do levého seznamu všechny prvky z pravého,
- časová složitost: *lineární*,
  - podrobněji: *lineární k délce pravého seznamu*,
  - levý seznam délky n, pravý délky m, složitost cca m,
  - na délce levého seznamu *nezávisí*.

Složitost algoritmů (zjednodušeně)

Časová složitost (obecně)

- funkce, která *velikosti vstupu* přiřazuje *počet kroků*.
- Typicky měříme nejhorší případ
  - (existují i jiné možnosti, např. průměrný případ).
- Je třeba si ujasnit:
  - co jsou základní kroky, které počítáme,
  - co je *velikost vstupu*.

- Při počítání se seznamy (pro účely tohoto předmětu):
  - základní kroky jsou operace s jednotlivými prvky,
  - velikost vstupu je délka seznamu.

Asymptotická časová složitost

- Zajímá nás pouze rychlost růstu funkcí.
- Zanedbáváme nedůležité části složitostní funkce:
  - násobení konstantou, přičítání konstanty, nevedoucí členy polynomů apod.
  - (V praxi ale někdy i tyto složky mohou být důležité!)

Asymptotická časová složitost

- Zajímá nás pouze rychlost růstu funkcí.
- Zanedbáváme nedůležité části složitostní funkce:
  - násobení konstantou, přičítání konstanty, nevedoucí členy polynomů apod.
  - (V praxi ale někdy i tyto složky mohou být důležité!)

- *Konstantní složitost* – shora omezená konstantou,
  - tj. nezávislá na velikosti vstupu.
- *Lineární složitost* – shora omezená lineární funkcí,
  - tj. polynomem stupně 1, např. n, $3n + 7$, $1000n + 9$, ...
- *Kvadratická složitost* – shora omezená kvadratickou funkcí,
  - tj. polynomem stupně 2, např. n^2, $6n^2 + 11$, $n^2 - 7n + 3$, ...
- *Exponenciální složitost* – shora omezená exponenciální funkcí,
  - např. 2^n, $3^n + 10$, $1000^n - 1$, ...
- A jiné...

Složitost algoritmů – pro představu

- Mějme tři algoritmy A, B, C pro zadaný problém:
  - A má složitost $100 \cdot n$ operací,
  - B má složitost $4 \cdot n^2$ operací,
  - C má složitost $2^n - 1$ operací.
  - Řekněme, že (pro účely tohoto slajdu) jedna operace trvá 1 ns.

n	A	B	C
1	100 ns	4 ns	1 ns
10	1 µs	400 ns	1 µs
50	5 µs	10 µs	13 *dní*
90	9 µs	32,4 µs	39 **miliard let**[1]
1000	100 µs	4 ms	
300000	30 ms	6 minut	

- I vyšší řád složitosti může být někdy výhodnější!
  - Např. malé vstupy: viz 1. řádek (lze zvýraznit volbou konstant).

[1] Odhadované stáří vesmíru je cca 14 miliard let.

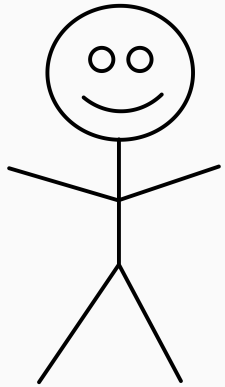
Datové struktury

Práce s daty

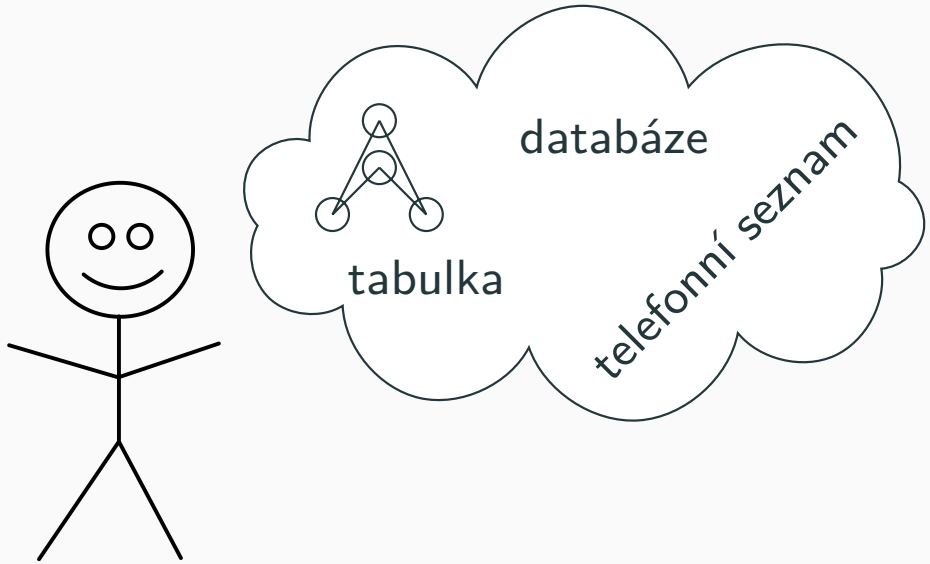
- Jaká data budu zpracovávat/potřebovat k řešení problému?
- Jaké operace s daty budu chtít provádět?

- Jak rychle budu chtít, aby operace s daty fungovaly?
- Jak úsporně budu chtít, aby data byla uložena?

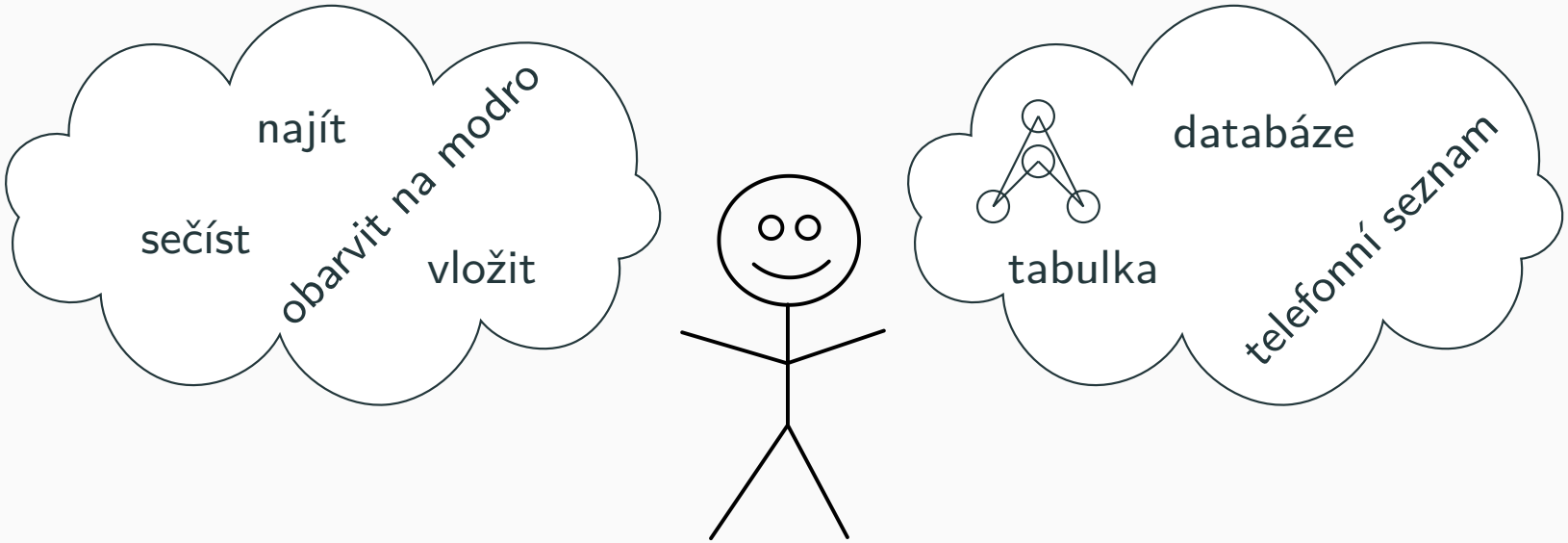
Pohled na data – dva světy



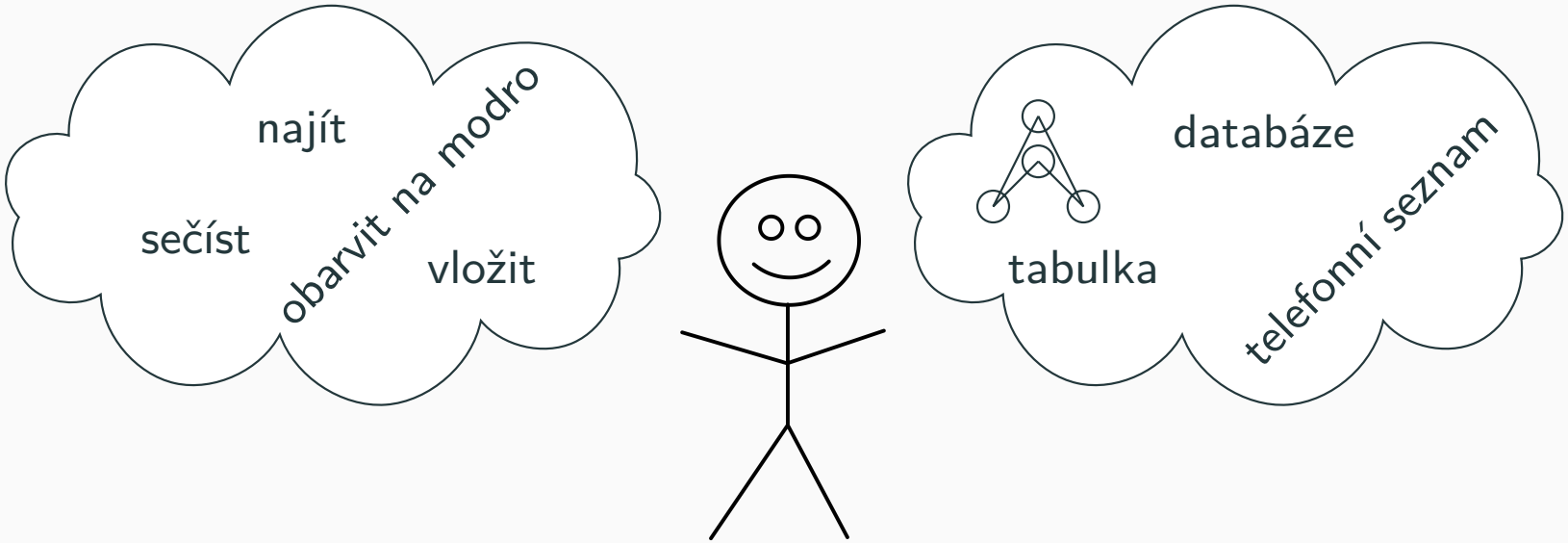
Pohled na data – dva světy

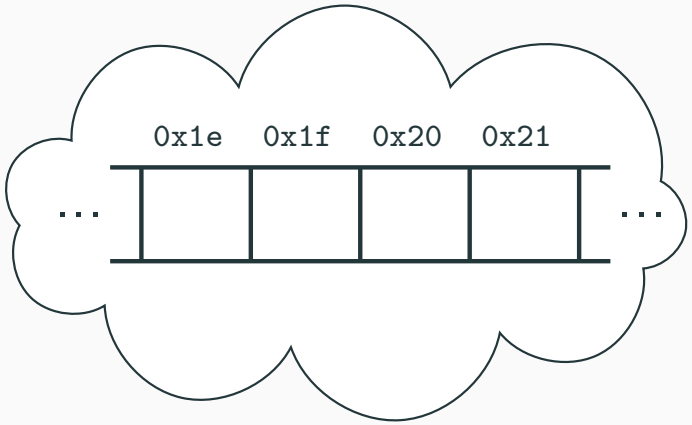


Pohled na data – dva světy

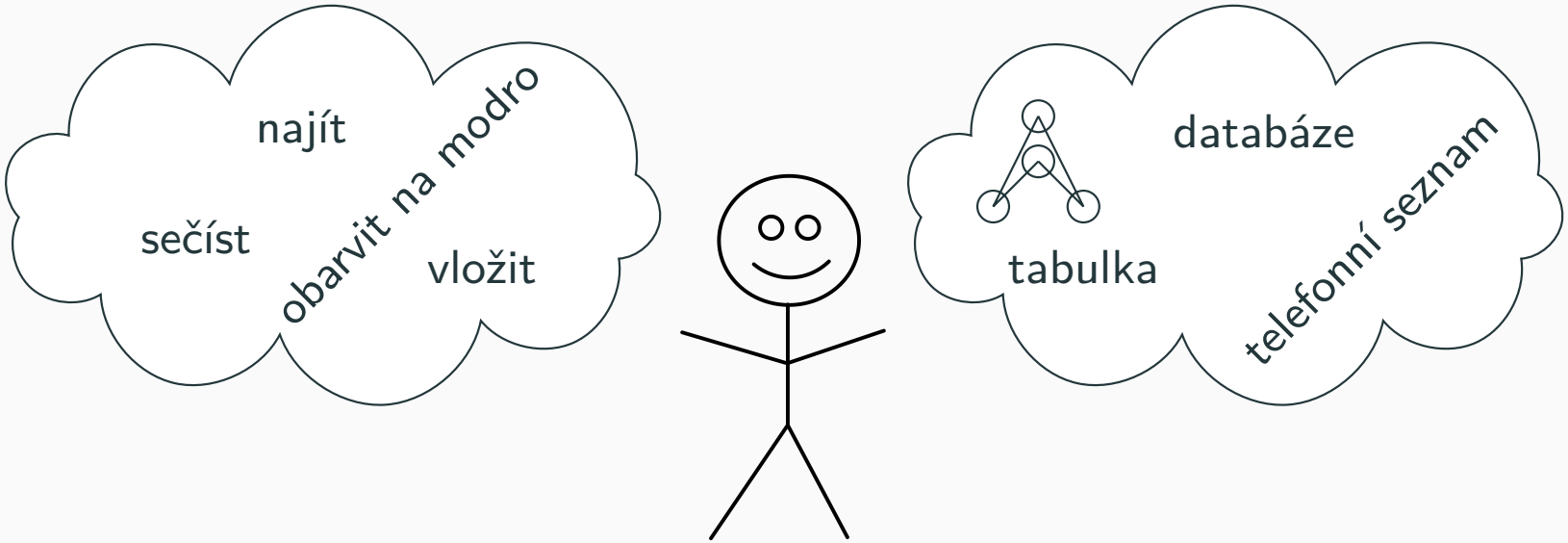


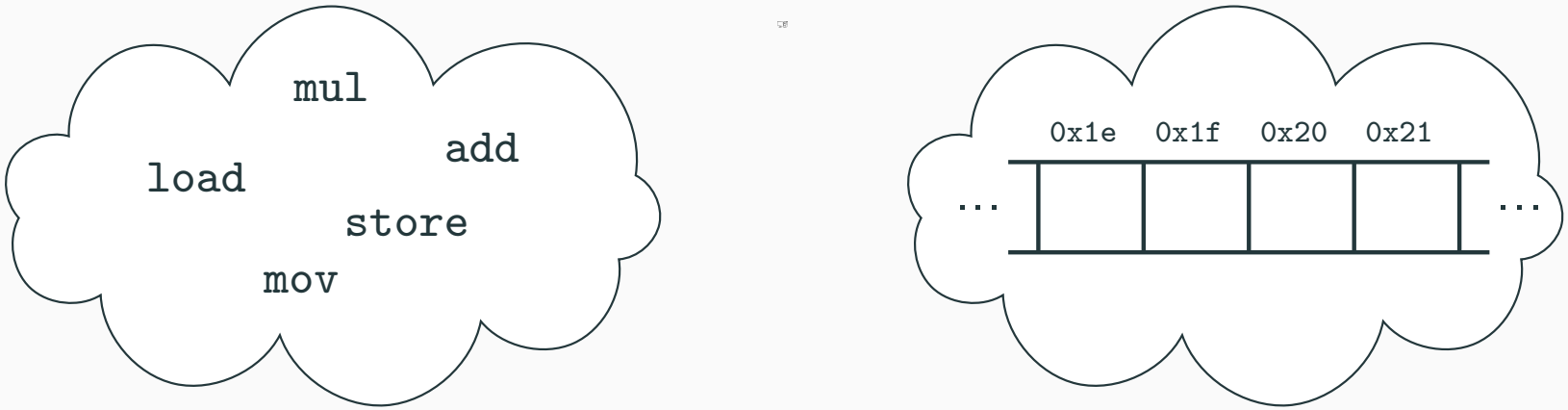
Pohled na data – dva světy



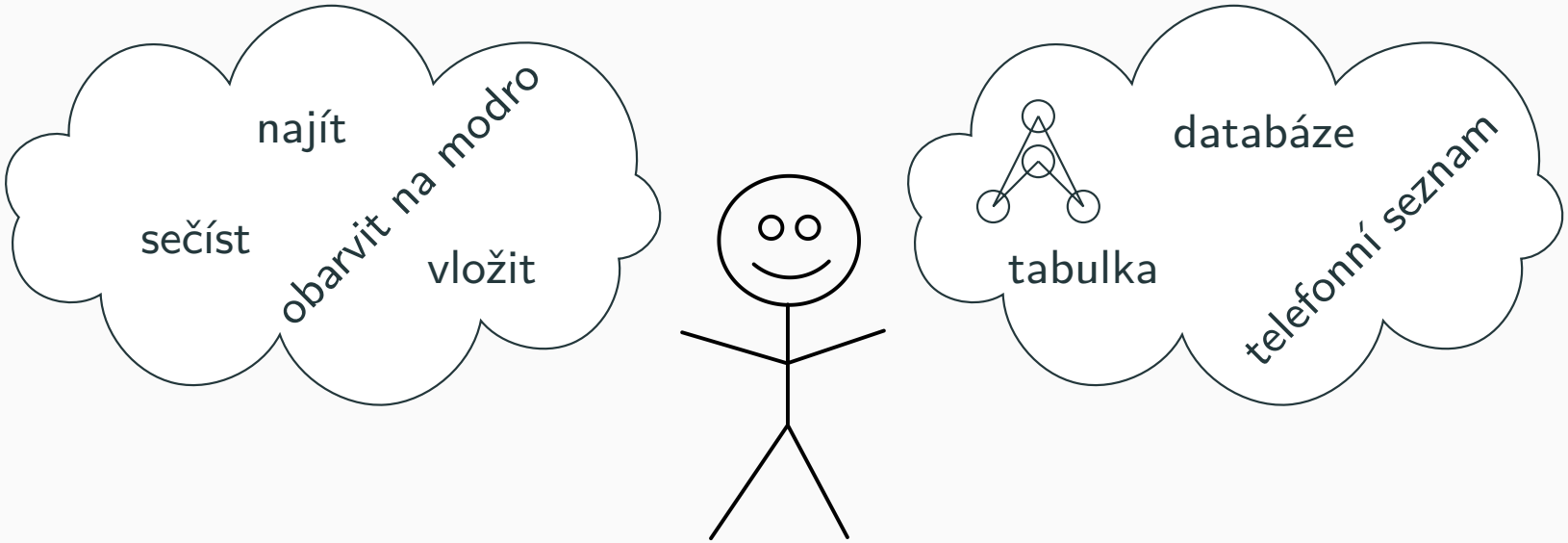


Pohled na data – dva světy

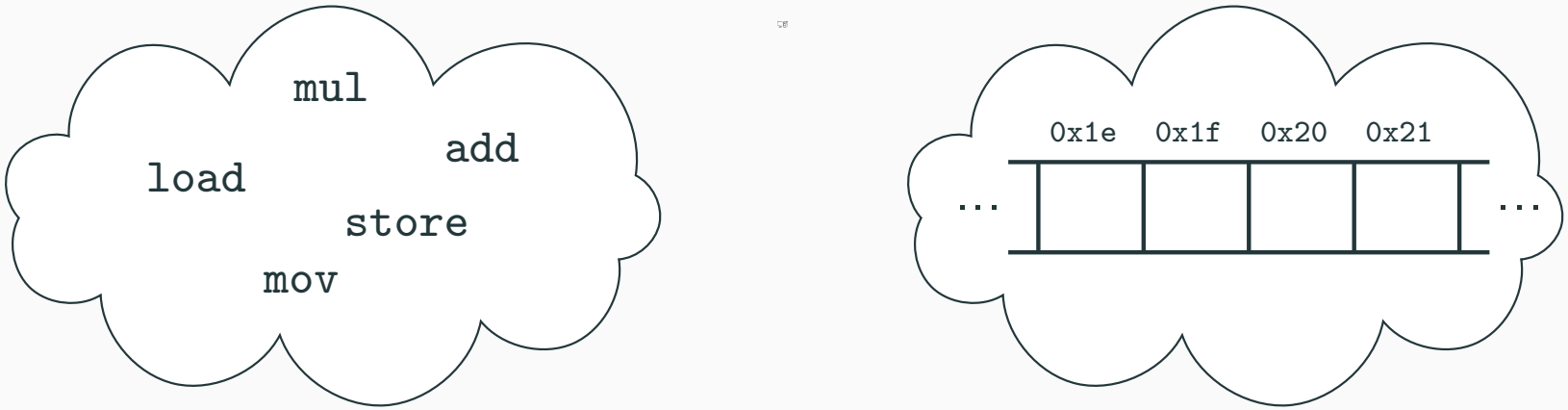




Pohled na data – dva světy




```
cops = set()
cops.add("Jake Peralta")
cops.add("Rosa Diaz")
```



Datové typy

Datový typ

- Množina hodnot, které patří do daného typu.
- Operace, které je možno s těmito hodnotami provádět.

Abstraktní datový typ (uživatelský pohled na data)

- rozhraní,
- popis ukládaných hodnot a operací, které chceme provádět (případně i složitost).

Konkrétní datová struktura (implementační pohled na data)

- implementace,
- popis uložení dat v paměti (registrech, ...),
- definice funkcí pro práci s těmito daty.

Poznámka: hranice mezi abstraktním datovým typem a konkrétní datovou strukturou není vždy úplně ostrá.

Abstraktní datové typy

Nejznámější ADT:

- seznam, pole, ntice,
- zásobník,
- fronta, prioritní fronta,
- množina,
- slovník (asociativní pole).

Za ADT můžeme ovšem taky považovat:

- textový řetězec,
- celé číslo.

Abstraktní datové typy

Nejznámější ADT:

- seznam, pole, ntice,
- zásobník,
- fronta, prioritní fronta,
- množina,
- slovník (asociativní pole).

Za ADT můžeme ovšem taky považovat:

- textový řetězec,
- celé číslo.

Abstraktní pohled na data

- Výhody: jednodušší přemýšlení, snazší vývoj.
- Riziko: svádí k ignorování efektivity.

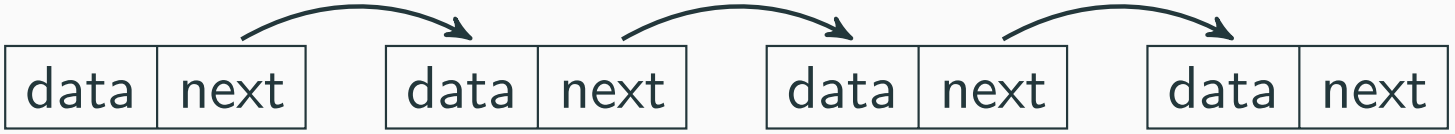
Abstraktní datový typ seznam

Seznam (různé varianty)

- obsahuje posloupnost prvků:
  - stejného typu/různého typu;
- přidání prvku:
  - na začátek,
  - na konec,
  - na určené místo;
- odebrání prvku:
  - ze začátku,
  - z konce,
  - konkrétní prvek;
- test prázdnosti, dotaz na délku
- a další operace…
  - např. přístup pomocí indexu.

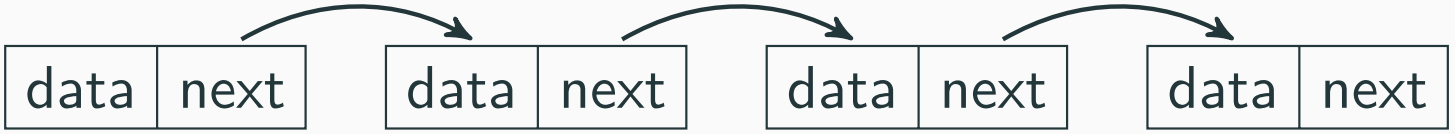
Implementace seznamu

Jednosměrně zřetězený seznam

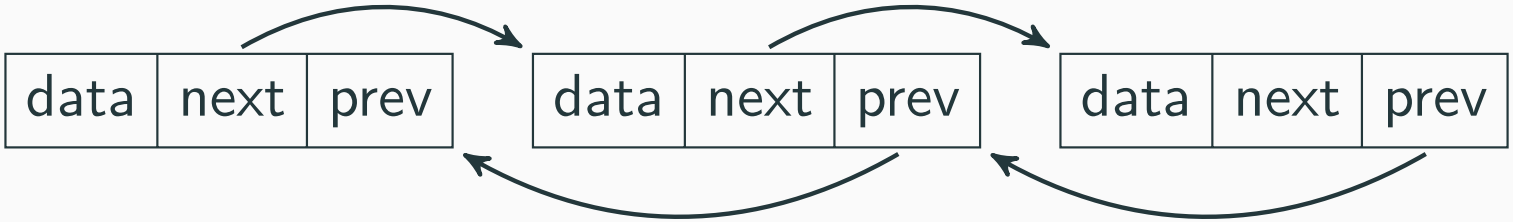


Implementace seznamu

Jednosměrně zřetězený seznam

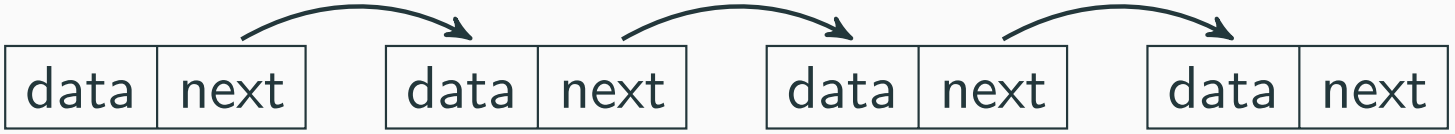


Obousměrně zřetězený seznam

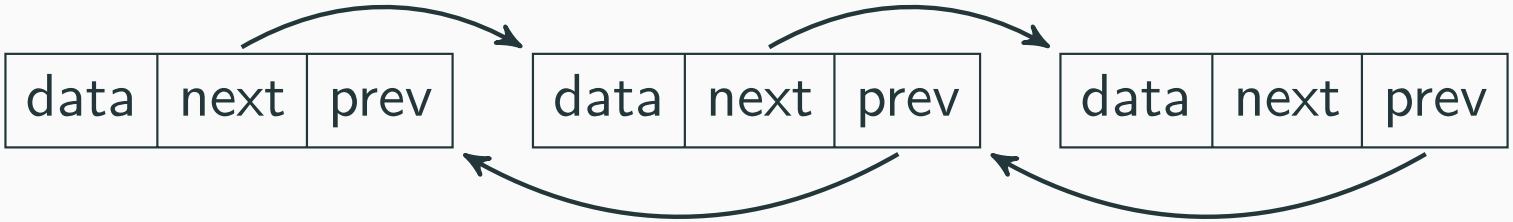


Implementace seznamu

Jednosměrně zřetězený seznam



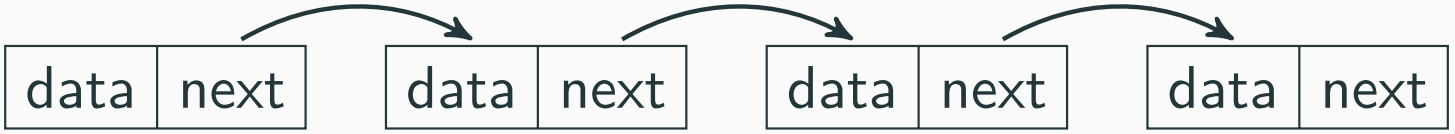
Obousměrně zřetězený seznam



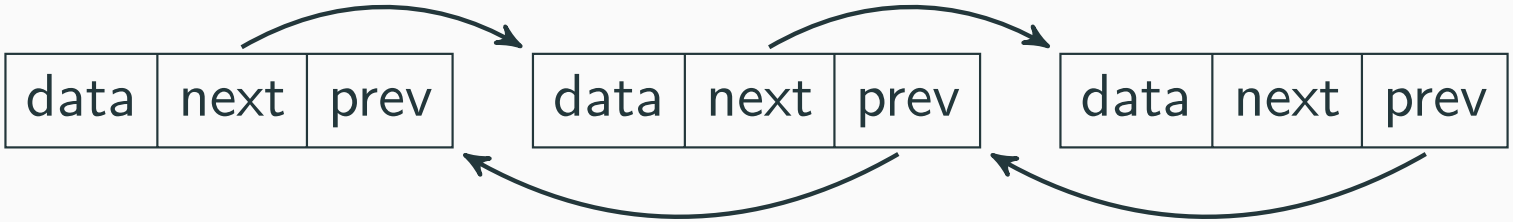
Dynamické pole

Implementace seznamu

Jednosměrně zřetězený seznam



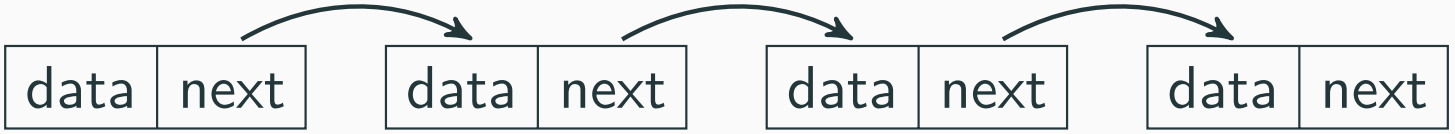
Obousměrně zřetězený seznam



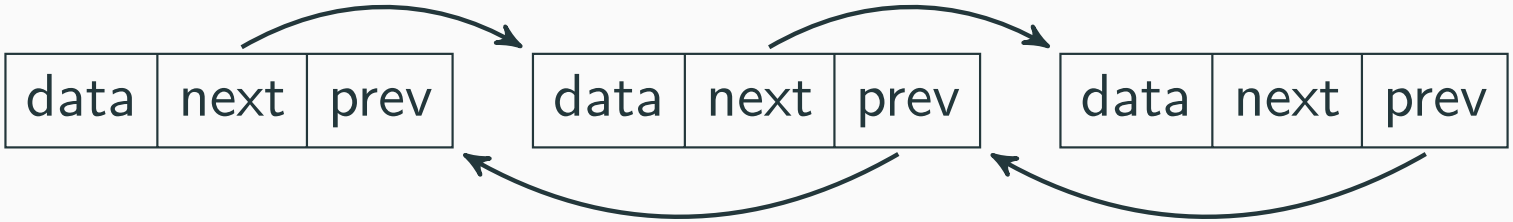
Dynamické pole

Implementace seznamu

Jednosměrně zřetězený seznam



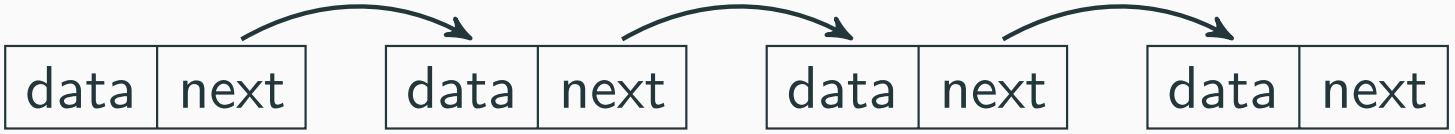
Obousměrně zřetězený seznam



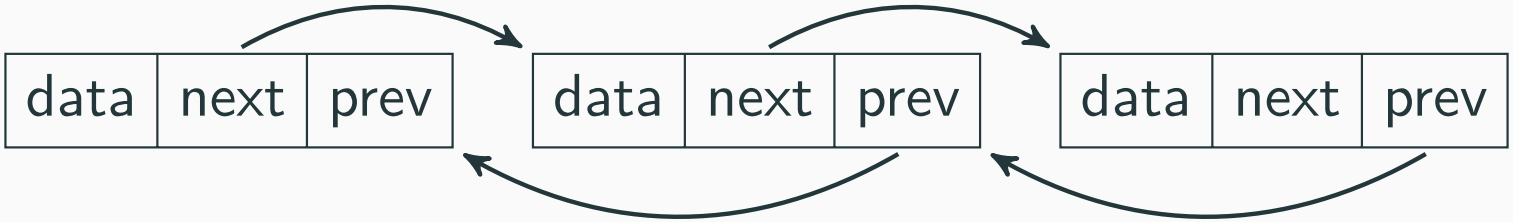
Dynamické pole

Implementace seznamu

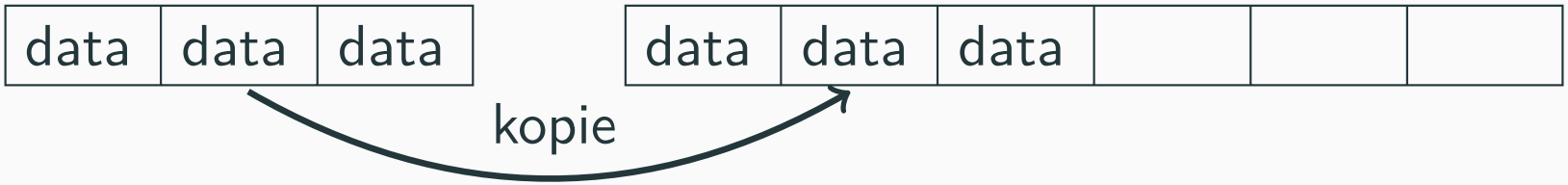
Jednosměrně zřetězený seznam



Obousměrně zřetězený seznam

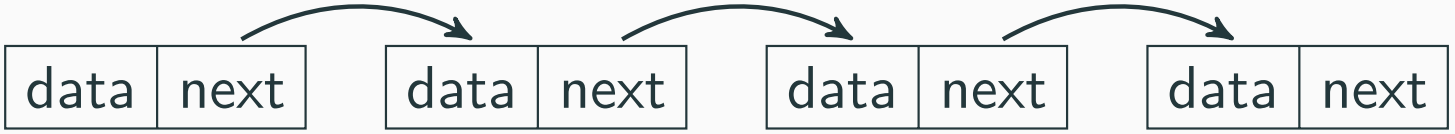


Dynamické pole

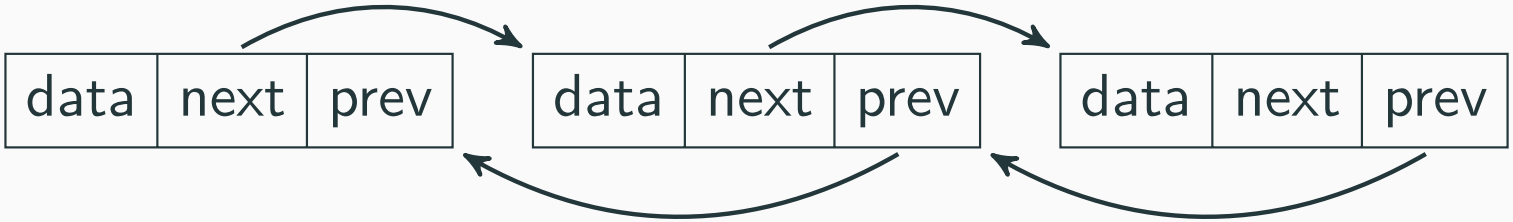


Implementace seznamu

Jednosměrně zřetězený seznam



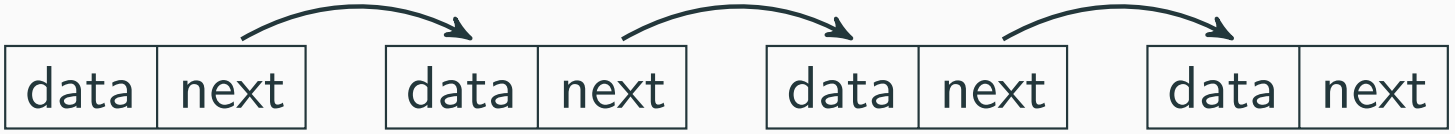
Obousměrně zřetězený seznam



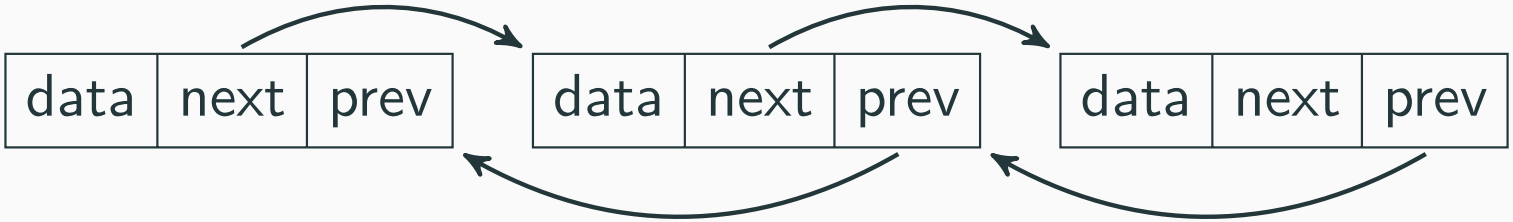
Dynamické pole

Implementace seznamu

Jednosměrně zřetězený seznam



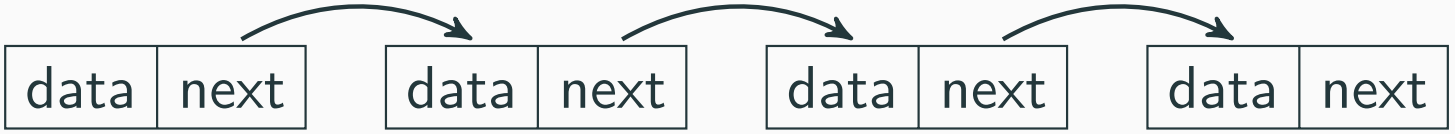
Obousměrně zřetězený seznam



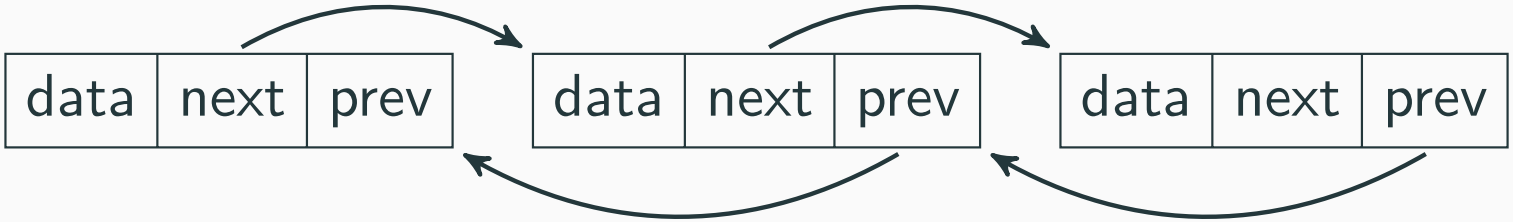
Dynamické pole

Implementace seznamu

Jednosměrně zřetězený seznam



Obousměrně zřetězený seznam

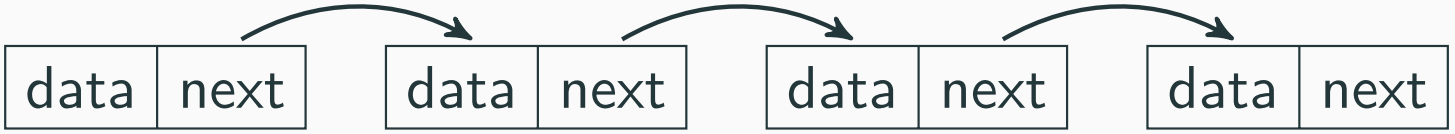


Dynamické pole

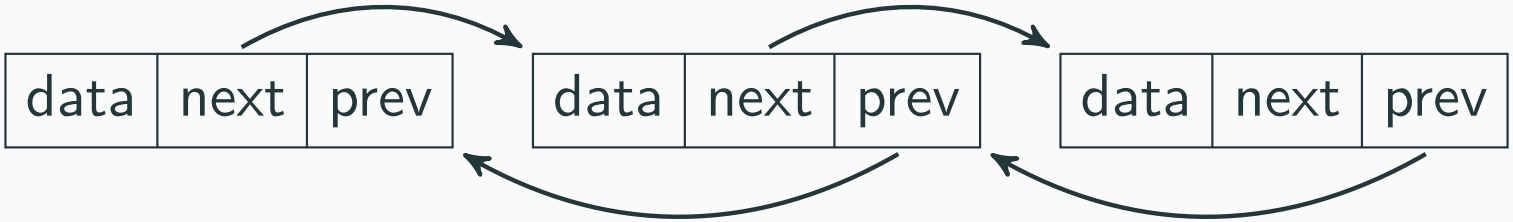
Jakou implementaci používá Pythonovský seznam?

Implementace seznamu

Jednosměrně zřetězený seznam



Obousměrně zřetězený seznam



Dynamické pole

Jakou implementaci používá Pythonovský seznam?
Mělo by nás to zajímat?

Implementace seznamu

Jednosměrně/obousměrně zřetězený seznam (SLL/DLL)

- Rychlé vkládání prvků (na začátek, na konec, na určené místo).
  - SLL: konec/určené místo – nutné dodatečné odkazy.
- Pomalé indexování, proč?

Implementace seznamu

Jednosměrně/obousměrně zřetězený seznam (SLL/DLL)

- Rychlé vkládání prvků (na začátek, na konec, na určené místo).
  - SLL: konec/určené místo – nutné dodatečné odkazy.
- Pomalé indexování, proč?
  - Ke konkrétnímu prvku musíme „doskákat“.
- (Později si zkusíme sami implementovat.)

Pole (klasické, statické) – *array*

- Souvislý úsek paměti, prvky stejné velikosti.
- Indexování je velmi rychlé, proč?

Implementace seznamu

Jednosměrně/obousměrně zřetězený seznam (SLL/DLL)

- Rychlé vkládání prvků (na začátek, na konec, na určené místo).
  - SLL: konec/určené místo – nutné dodatečné odkazy.
- Pomalé indexování, proč?
  - Ke konkrétnímu prvku musíme „doskákat“.
- (Později si zkusíme sami implementovat.)

Pole (klasické, statické) – *array*

- Souvislý úsek paměti, prvky stejné velikosti.
- Indexování je velmi rychlé, proč?
  - Adresa prvku `pole[i]` je *začátek pole* + *i* * *velikost prvku*.
- Nedá se snadno rozšiřovat, proč?

Implementace seznamu

Jednosměrně/obousměrně zřetězený seznam (SLL/DLL)

- Rychlé vkládání prvků (na začátek, na konec, na určené místo).
  - SLL: konec/určené místo – nutné dodatečné odkazy.
- Pomalé indexování, proč?
  - Ke konkrétnímu prvku musíme „doskákat“.
- (Později si zkusíme sami implementovat.)

Pole (klasické, statické) – *array*

- Souvislý úsek paměti, prvky stejné velikosti.
- Indexování je velmi rychlé, proč?
  - Adresa prvku `pole[i]` je *začátek pole* + *i* * *velikost prvku*.
- Nedá se snadno rozšiřovat, proč?
  - Máme jen vyhrazený úsek paměti,
  - paměť dál už může patřit někomu jinému.

Implementace seznamu

Dynamické pole

- Vyhradíme si úsek paměti jako pro klasické pole.
- Když nám paměť dojde, vyhradíme si větší úsek paměti a původní prvky do ní přesuneme (nelze-li zvětšit na místě).
  - Typicky dvakrát větší, ale jsou i jiné možnosti.
- Implementace, kterou používá Pythonovský typ `list`.
- Tyto základní operace mají konstantní časovou složitost:
  - indexování `seznam[index]`,
  - zjištění délky `len(seznam)` – ta je někde napsaná bokem,
  - odstraňování prvků z konce `seznam.pop()`,
  - přidávání prvků na konec `seznam.append(prvek)`[2].

[2] Ve skutečnosti jen tzv. *amortizovaně konstantní*, ale v IB111 ignorujeme (pokročilé téma).

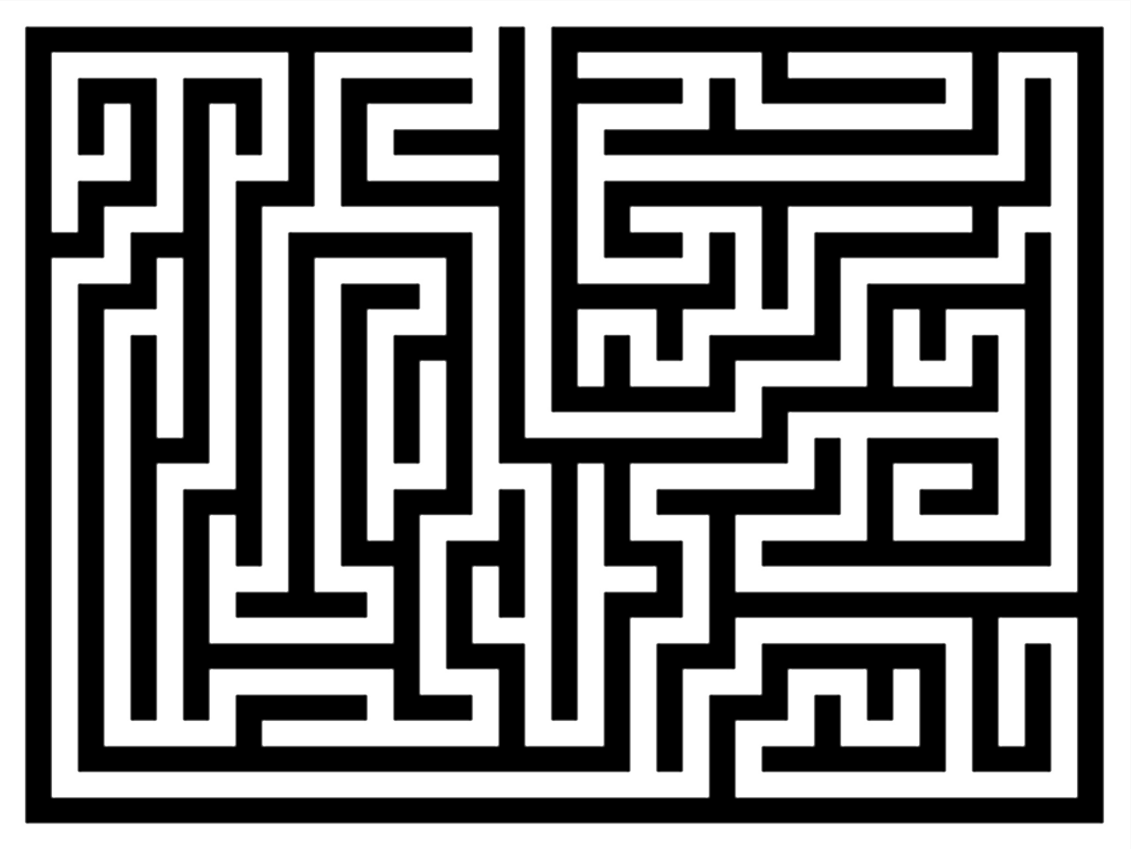
Abstraktní datový typ zásobník

Zásobník

- Obsahuje prvky v pořadí LIFO (*Last In First Out*).
- Operace:
  - push (vložení),
  - pop (odstranění),
  - top (náhled na horní prvek),
  - empty (test prázdnosti).
- Mnohá použití – např:
  - procházení grafů,
  - analýza syntaxe,
  - vyhodnocování výrazů,
  - rekurze, ...

Motivace pro zásobník

- Procházení bludiště bez smyček.



Zásobník v Pythonu

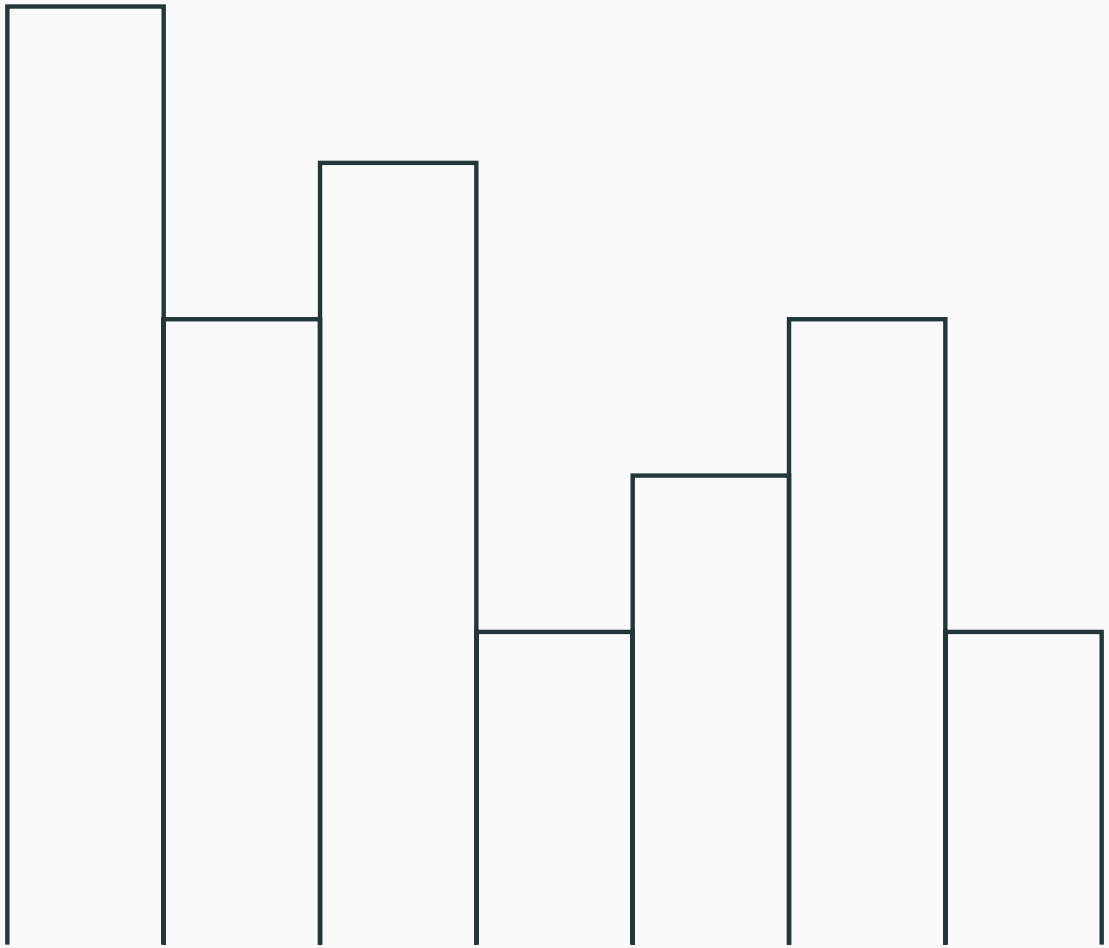
Implementace zásobníku pomocí seznamu

- Dno zásobníku je *vlevo* (na začátku, index `0`).
- Vrchol zásobníku je *vpravo* (na konci, index `-1`).

- *push:* `stack.append(element)`.
- *pop:* `stack.pop()`.
- *top:* `stack[-1]`.
- *empty:* `len(stack) == 0`.

Příklad: Mrakodrapy

- Ze kterých střech je výhled na jezero nacházející se vpravo?



Příklad: Mrakodrapy

skyscrapers.py

```
def clear_view(heights: list[int]) -> list[int]:
    stack: list[int] = []
    for current in range(len(heights)):
        while len(stack) > 0 and \
                heights[stack[-1]] < heights[current]:
            stack.pop()
        stack.append(current)

    return stack
```

K zamyšlení pro pokročilé

- Je tento algoritmus korektní?
- Jakou má časovou složitost?

Abstraktní datový typ množina

Množina

- Neuspořádaná kolekce dat bez vícenásobných prvků.
- Primárně určená pro rychlé vyhledávání.
- Operace:

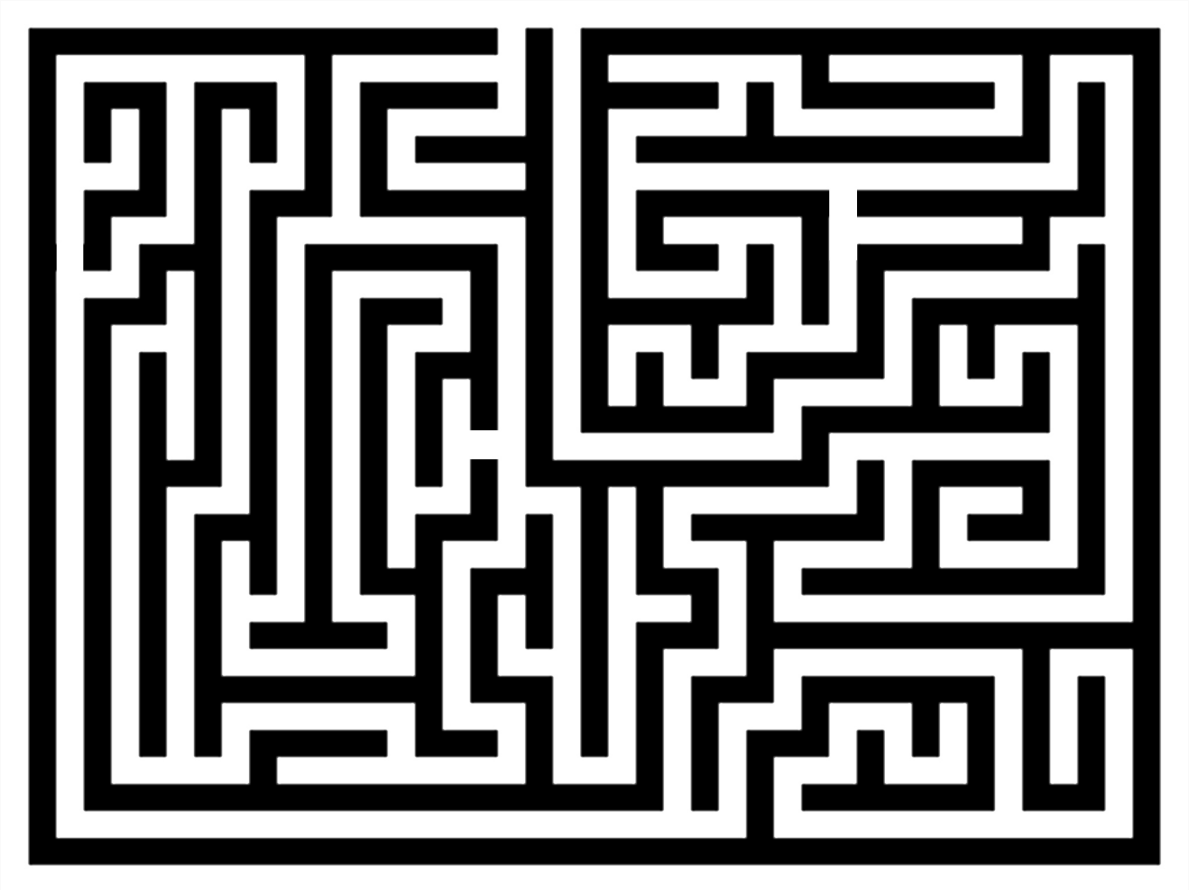
Abstraktní datový typ množina

Množina

- Neuspořádaná kolekce dat bez vícenásobných prvků.
- Primárně určená pro rychlé vyhledávání.
- Operace:
  - insert (vložení),
  - find (vyhledání prvku, test přítomnosti),
  - delete (odstranění),
  - případně i množinové operace (sjednocení, průnik, ...).
- Použití:
  - grafové algoritmy (označení navštívených vrcholů),
  - výpis unikátních slov, ...

Motivace pro množinu

- Procházení bludiště se smyčkami.



Jak implementovat množinu?

Jak implementovat množinu?

Pomocí seznamu prvků (typu `list`)

- Vkládání, hledání, mazání je v nejhorším případě

Jak implementovat množinu?

Pomocí seznamu prvků (typu `list`)

- Vkládání, hledání, mazání je v nejhorším případě *lineární*:
  - je třeba seznam projít prvek po prvku,
  - (v průměrném případě také).
- Co kdyby byl seznam *seřazený*?
  - Vylepší se hledání (uvidíme příště),
  - vkládání a mazání bude pořád lineární.

Jak implementovat množinu?

Pomocí seznamu prvků (typu `list`)

- Vkládání, hledání, mazání je v nejhorším případě *lineární*:
  - je třeba seznam projít prvek po prvku,
  - (v průměrném případě také).
- Co kdyby byl seznam *seřazený*?
  - Vylepší se hledání (uvidíme příště),
  - vkládání a mazání bude pořád lineární.

Pomocí pole pravdivostních hodnot (`list[bool]` pevné délky)

- Jsou-li prvky z malého rozsahu celý čísel (typicky 0 až malé k).
- Už jsme viděli, kde?

Jak implementovat množinu?

Pomocí seznamu prvků (typu `list`)

- Vkládání, hledání, mazání je v nejhorším případě *lineární*:
  - je třeba seznam projít prvek po prvku,
  - (v průměrném případě také).
- Co kdyby byl seznam *seřazený*?
  - Vylepší se hledání (uvidíme příště),
  - vkládání a mazání bude pořád lineární.

Pomocí pole pravdivostních hodnot (`list[bool]` pevné délky)

- Jsou-li prvky z malého rozsahu celý čísel (typicky 0 až malé k).
- Už jsme viděli, kde? Vzpomeňte si na Eratosthenovo síto.
- Vkládání, hledání, mazání je *konstantní*.
  - Za cenu dražší inicializace.

Jak implementovat množinu?

Pokročilejší implementace (více v navazujících předmětech)

- Stromové struktury:
  - typicky tzv. vyhledávací stromy,
  - *logaritmická* složitost operací,
  - jiné možnosti (trie – tzv. prefix tree pro množinu řetězců, ...).

- Hashovací tabulky:
  - v nejhorším případě až lineární složitost,
  - ale očekávaná složitost je konstantní (nejhorší případ nastává velmi zřídka),
  - reálná efektivita záleží na řadě vlivů,
  - použito v Pythonu pro datový typ `set`.

Množina v Pythonu – typ `set`

Vlastnosti

- Prvky v množině musí být neměnné objekty:
  - čísla, řetězce, ntice čísel a řetězců, ...,
  - ale ne seznamy či ntice s vnořenými seznamy apod.
- Pořadí prvků v množině není nijak zaručeno.

Vytvoření

- `set()` prázdná množina[3], `{1, 3, 7}` množina zadaná výčtem.
- `set(l)` vytvoří množinu ze seznamu.
  - `list(s)` naopak vytvoří seznam z množiny.

Typová anotace – `set[typ]`.

[3] Pozor: `{}` má jiný význam – je to prázdný *slovník* (viz dále).

Množina v Pythonu – typ `set`

Základní operace (v IB111 považujeme za konstantní)

- `len(s)` – počet prvků množiny `s`.
- `s.add(x)` – přidá prvek do množiny (vrací `None`).
- `s.remove(x)` – odebere prvek z množiny (vrací `None`).
  - Pokud prvek v množině není, dojde k chybě.
- `x in s` se vyhodnotí na `True`/`False`, jestliže `s` obsahuje `x`.

Iterace

- Procházení prvků množiny: `for x in s:`.
- Pozor na to, že není zaručeno pořadí.
- Množinu není uvnitř těla cyklu dovoleno měnit.

Příklad: Unikátní prvky seznamu

unique.py

- Chceme vybrat ze seznamu všechny jeho prvky bez opakování.
- Přímočaré řešení pomocí seznamů – funkce `contains` je v přiloženém souboru (sekvenční průchod seznamem):

```
def unique_elements1(my_list: list[int]) -> list[int]:
    result: list[int] = []
    for element in my_list:
        if not contains(result, element):
            result.append(element)
    return result
```

- Časová složitost:

Příklad: Unikátní prvky seznamu

unique.py

- Chceme vybrat ze seznamu všechny jeho prvky bez opakování.
- Přímočaré řešení pomocí seznamů – funkce `contains` je v přiloženém souboru (sekvenční průchod seznamem):

```
def unique_elements1(my_list: list[int]) -> list[int]:
    result: list[int] = []
    for element in my_list:
        if not contains(result, element):
            result.append(element)
    return result
```

- Časová složitost: *kvadratická*.

Příklad: Unikátní prvky seznamu

unique.py

- Chceme vybrat ze seznamu všechny jeho prvky bez opakování.
- Přímočaré řešení pomocí seznamů – funkce `contains` je v přiloženém souboru (sekvenční průchod seznamem):

```
def unique_elements1(my_list: list[int]) -> list[int]:
    result: list[int] = []
    for element in my_list:
        if not contains(result, element):
            result.append(element)
    return result
```

- Časová složitost: *kvadratická*.
- Řešení pomocí seřazení seznamu – v pozdější přednášce.

Příklad: Unikátní prvky seznamu

unique.py

- Řešení pomocí množin:

```
def unique_elements2(my_list: list[int]) -> list[int]:
    return list(set(my_list))
```

- Co kdybychom chtěli zachovat jejich pořadí?

Příklad: Unikátní prvky seznamu

unique.py

- Řešení pomocí množin:

```
def unique_elements2(my_list: list[int]) -> list[int]:
    return list(set(my_list))
```

- Co kdybychom chtěli zachovat jejich pořadí?

```
def unique_elements3(my_list: list[int]) -> list[int]:
    seen = set()
    result = []
    for elem in my_list:
        if elem not in seen:
            result.append(elem)
            seen.add(elem)
    return result
```

Abstraltní datový typ slovník

Slovník (dictionary, map, asociativní pole)

- Neuspořádaná množina dvojic (klíč, hodnota).
- Klíče jsou unikátní.
- Operace podobné množině:
  - vložení dvojice (klíč, hodnota),
  - hledání podle klíče, přístup k odpovídající hodnotě,
  - mazání podle klíče.
- Použití:
  - překlad (např. UČO na jméno, jméno na tel. číslo apod.),
  - zjištění počtu výskytů (např. slov v textu),
  - „cache“ výsledků náročných výpočtů, ...

Jak implementovat slovník?

Pomocí seznamu dvojic (typu `list`)

- Každý prvek seznamu je dvojice (klíč, hodnota).
- Podobně jako u množin: pomalé...

Pomocí pole hodnot

- Klíče jsou indexy do pole (pro malý rozsah přirozených čísel použitých jako klíče).

Pokročilejší implementace

- Stromy, hashovací tabulky (jako u množin).
- Python – datový typ `dict`:
  - hashovací tabulka,
  - očekávaná složitost základních operací je konstantní.

Slovník v Pythonu – typ `dict`

Vlastnosti

- Klíče musí být neměnného typu, hodnoty mohou být libovolné.
- Na pořadí položek ve slovníku je lepší se nespoléhat.[4]

Vytvoření

- Zápis do složených závorek `{}`.
- Klíč a hodnotu oddělujeme dvojtečkou,
  položky oddělujeme čárkami:
  `{"Buffy": 5550101, "Xander": 5550168}`.

Typová anotace – `dict[klíč, hodnota]`.

[4] Od Pythonu 3.7 je dáno pořadím vložení, ale je to poněkud obskurní a v jiných jazycích to nenajdete.

Slovník v Pythonu – typ `dict`

Základní operace (v IB111 považujeme za konstantní)

- Vytvoření nebo změna položky: `d[key] = value`.
- Čtení hodnoty položky:

```
d[key]                  # pokud klíč není, chyba
d.get(key, default)     # pokud klíč není, vrátí default
d.get(key)              # pokud klíč není, vrátí None
```

- Hledání: `key in d` (vrací `True`/`False`).
- Odstranění položky: `d.pop(key)`,
  - vrátí příslušnou hodnotu.
- Počet položek ve slovníku: `len(d)`.

Slovník v Pythonu – typ `dict`

Iterace

- Procházení klíčů slovníku:
  `for key in my_dict.keys():`.
- Procházení celých položek slovníku:
  `for key, value in my_dict.items():`.
- Procházení hodnot slovníku (bez odstranění příp. duplicit hodnot):
  `for value in my_dict.values():`.
- Objekty vrácené `.keys`, `.items`, `.values` nejsou seznamy, ale dají se na seznamy konvertovat pomocí `list(…)`.

Příklad: Četnosti prvků v seznamu

`freq.py`

- Vstup: seznam prvků (např. řetězců).
- Výstup: slovník, který každému prvku přiřazuje jeho četnost
  - (kolikrát se v seznamu vyskytuje).

Příklad: Četnosti prvků v seznamu

freq.py

- Vstup: seznam prvků (např. řetězců).
- Výstup: slovník, který každému prvku přiřazuje jeho četnost
  - (kolikrát se v seznamu vyskytuje).

```
def word_freq(words: list[str]) -> dict[str, int]:
    freq: dict[str, int] = {}
    for word in words:
        freq[word] = freq.get(word, 0) + 1
    return freq
```

- Jaký je význam předposledního řádku?

Příklad: Četnosti prvků v seznamu

freq.py

- Vstup: seznam prvků (např. řetězců).
- Výstup: slovník, který každému prvku přiřazuje jeho četnost
  - (kolikrát se v seznamu vyskytuje).

```
def word_freq(words: list[str]) -> dict[str, int]:
    freq: dict[str, int] = {}
    for word in words:
        freq[word] = freq.get(word, 0) + 1
    return freq
```

- Jaký je význam předposledního řádku?
  - Pokud ve slovníku `freq` není položka s klíčem `word`, vytvoří se s hodnotou 1.
  - Jinak se k hodnotě položky s klíčem `word` přičte 1.

Příklad: Seskupení dle příjmení

`surnames.py`

- Vstup: seznam dvojic (jméno, příjmení).
- Výstup: slovník, který každému příjmení přiřazuje seznam všech příslušných jmen.

Příklad: Seskupení dle příjmení

surnames.py

- Vstup: seznam dvojic (jméno, příjmení).
- Výstup: slovník, který každému příjmení přiřazuje seznam všech příslušných jmen.

```
def group_by_surname(names: list[tuple[str, str]]) \
        -> dict[str, list[str]]:
    groups: dict[str, list[str]] = {}
    for name, surname in names:
        if surname not in groups:
            groups[surname] = []

        groups[surname].append(name)

    return groups
```

K zamyšlení

bad_code.py

Co je špatně na tomto kódu?

```
def allow_import(imports: list[str],
                 allowed: list[str]) -> bool:
    for lib in imports:
        if lib not in set(allowed):
            return False
    return True
```

K zamyšlení

bad_code.py

Co je špatně na tomto kódu?

```
def allow_import(imports: list[str],
                 allowed: list[str]) -> bool:
    for lib in imports:
        if lib not in set(allowed):
            return False
    return True
```

- `set(allowed)` pokaždé vytvoří novou množinu
  - a tedy zejména projde všechny prvky seznamu `allowed`.
- Lepší řešení

K zamyšlení

bad_code.py

Co je špatně na tomto kódu?

```
def allow_import(imports: list[str],
                 allowed: list[str]) -> bool:
    for lib in imports:
        if lib not in set(allowed):
            return False
    return True
```

- `set(allowed)` pokaždé vytvoří novou množinu
  - a tedy zejména projde všechny prvky seznamu `allowed`.
- Lepší řešení – vytvořit si množinu jednou na začátku.

K zamyšlení

bad_code.py

Co je špatně na tomto kódu?

```
def lookup(phonebook: dict[str, int],
           person_name: str) -> int | None:
    for name, phone in phonebook.items():
        if name == person_name:
            return phone
    return None
```

K zamyšlení

bad_code.py

Co je špatně na tomto kódu?

```
def lookup(phonebook: dict[str, int],
           person_name: str) -> int | None:
    for name, phone in phonebook.items():
        if name == person_name:
            return phone
    return None
```

Doing linear scans over an associative array is like trying to club someone to death with a loaded Uzi. *(Larry Wall)*