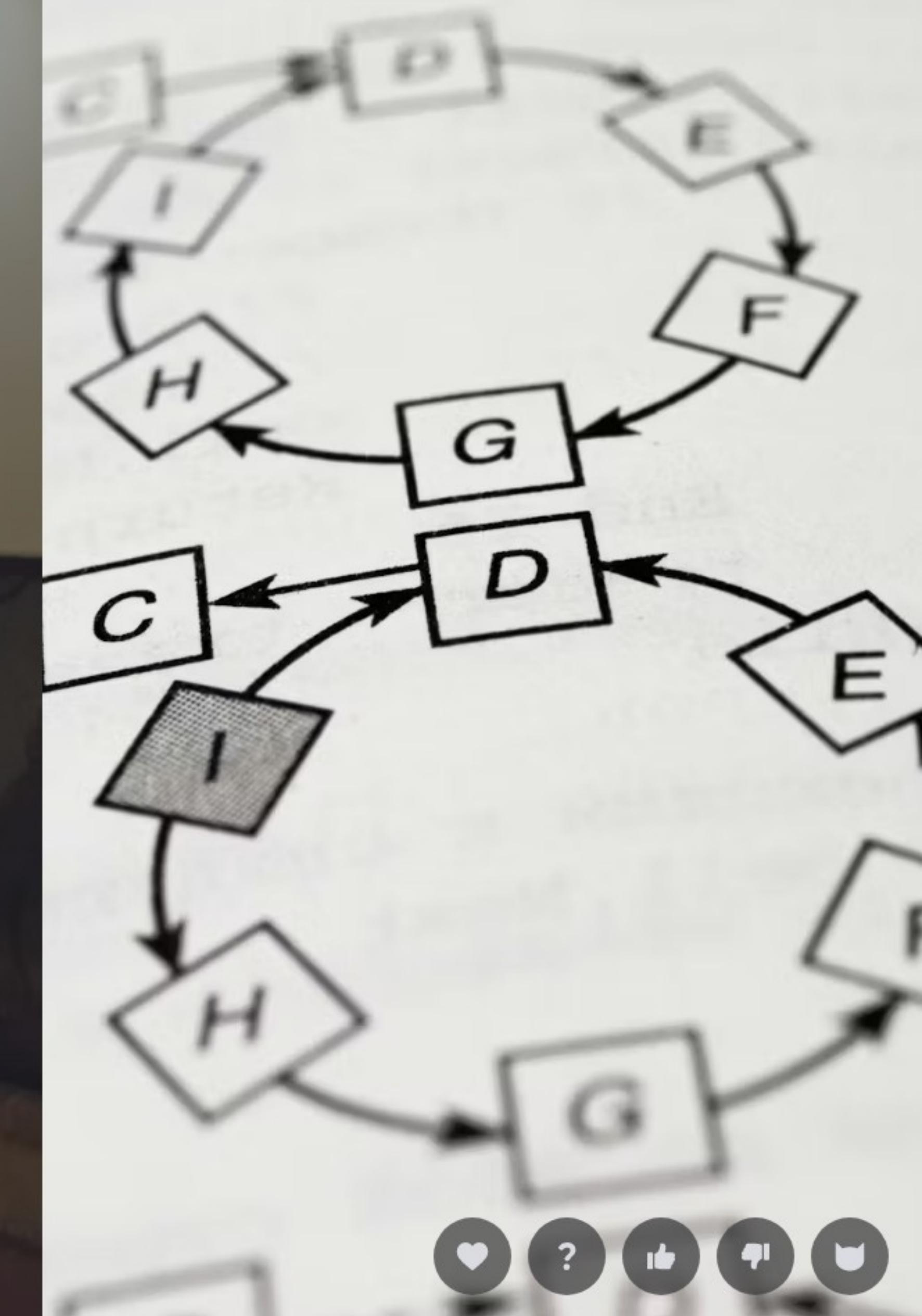


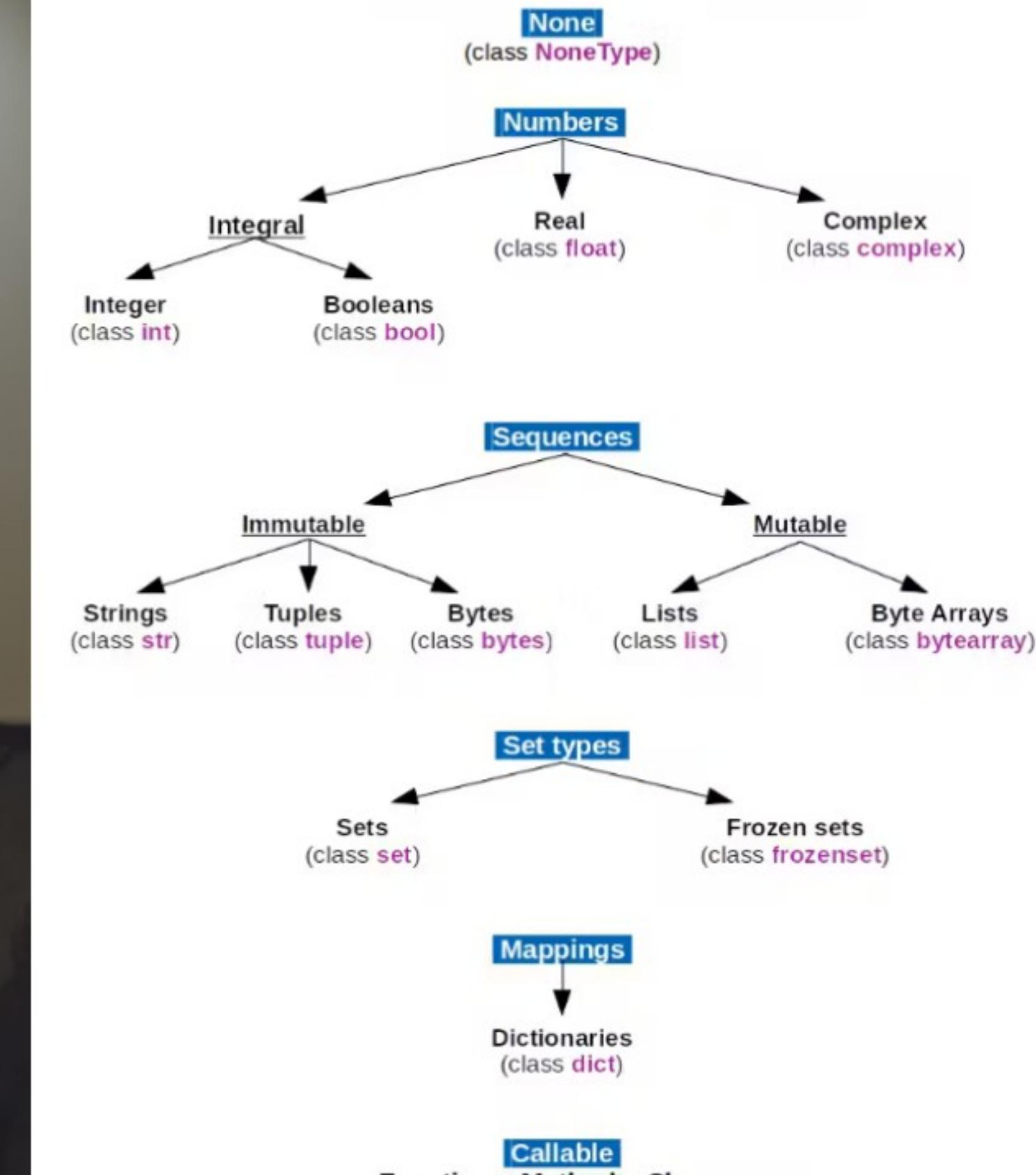
# Algorytmy i Struktury Danych

Zagadka: co to za miejsce w tle?



# Struktury danych





# Rodzaje struktur/typy danych

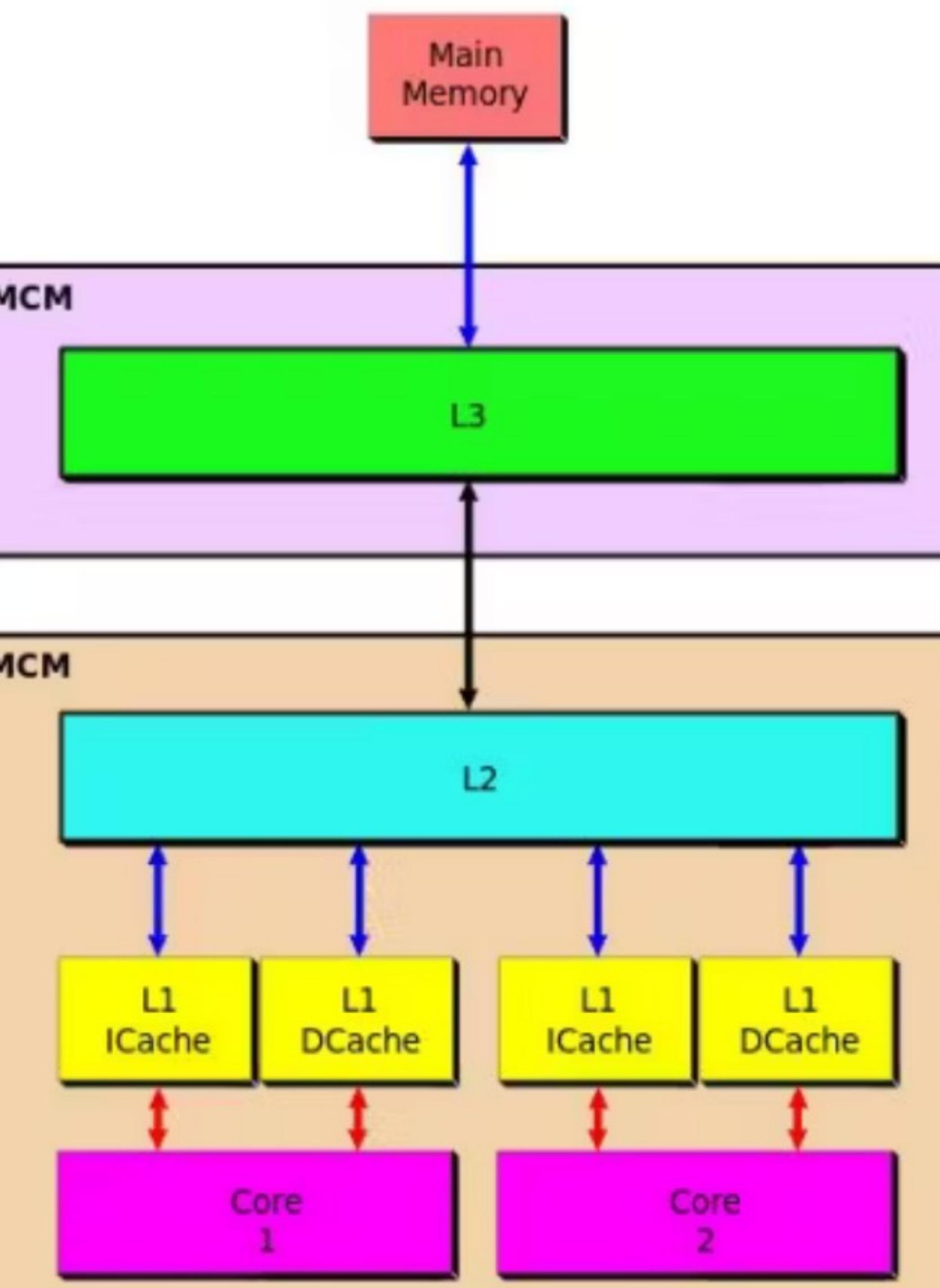
- Typy prymitywne (int, float, char...)
- Sekwencje
- Hashmapy
- Kopce ∈ drzewa ∈ grafy
- Inne struktury złożone



# Ważne pojęcia w pamięci komputera

- Wskaźniki - każda komórka pamięci ma swój adres
- Alokacja - nie możemy pisać gdzie chcemy, najpierw musimy poprosić o region pamięci
- Lokalność danych - program działa szybciej, gdy dane są obok siebie (czemu?)

## Multi-Core L2 Shared Cache

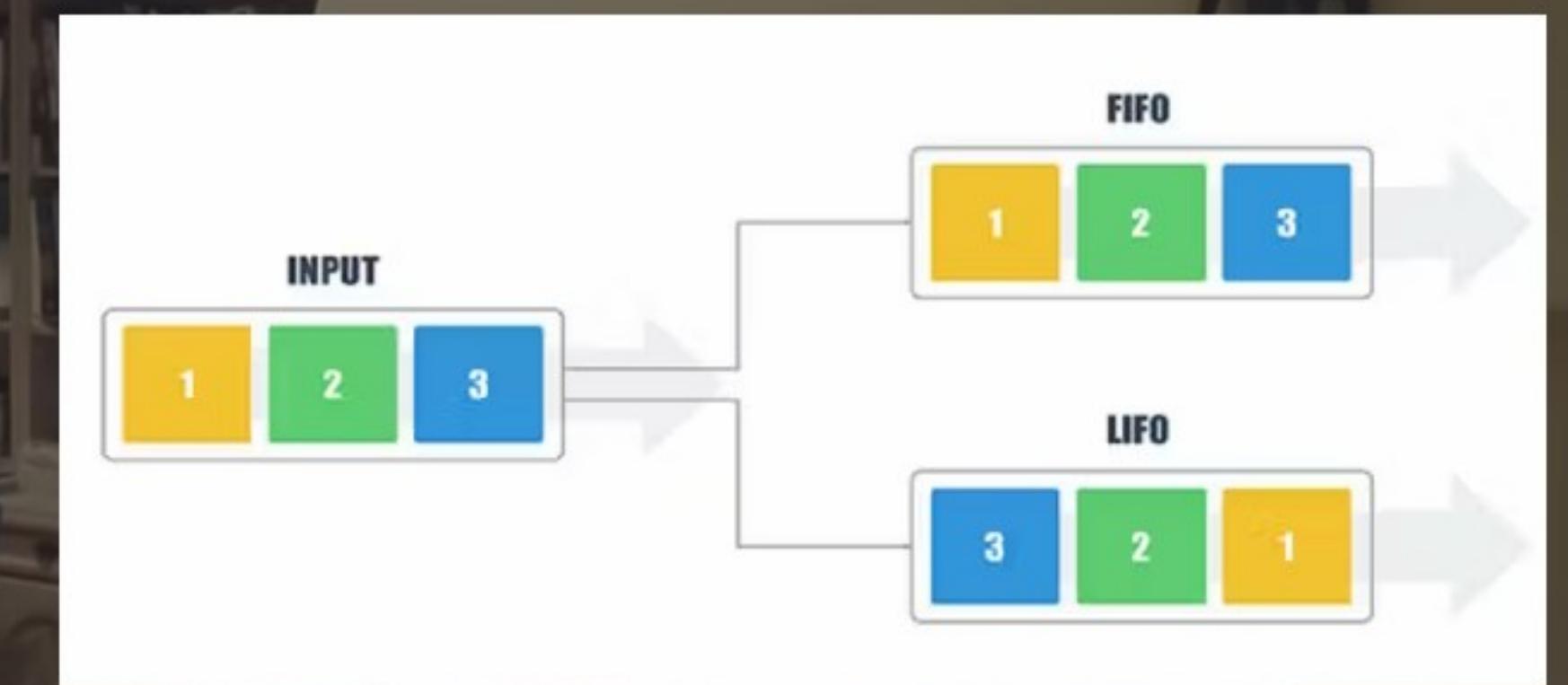


## Cache

- Szybka pamięć jest droga
- Mamy jej bardzo mało w komputerach (cache)
- Jeżeli chcemy operować na danych, musimy je sprowadzić do cache'a (zajmuje dużo czasu)
- Za jednym razem ściągamy do cache'a cały blok pamięci
- Jeśli pobierzemy za dużo danych, starsze bloki zostaną wyrzucone na wyższe poziomy cache'a (kosztowna operacja)
- Jeśli chcemy powrócić do tych danych, musimy je sprowadzić z powrotem (kosztowna operacja)

# Sekwencje

- W sekwencjach przechowujemy dane
- Czasami chcemy je dodawać lub usuwać
- Nierzadko w konkretnej kolejności
- Dwie typowe to LIFO i FIFO
- LIFO - Last In First Out - **stos**, kto ostatni wszedł, ten pierwszy wychodzi. Usuwamy najświeższe dane
- FIFO - First In First Out - usuwamy dane, które są na kolejce najdłużej



# Stos (lista)

- Klasa *list* jest stosem
- Szybko dodajemy na koniec funkcją *append*
- Szybko usuwamy ostatni element funkcją *pop*
- Jaka jest złożoność tych operacji?
- Jaka jest złożoność dodawania i usuwania na początku?

# Jaka jest złożoność `list.append()`?

✓  $O(1)$  ✗  $\theta(\log n)$  ✗  $\theta(n)$  ✓  $O(n)$

# Jaka jest złożoność `list.pop(0)`?

✗

$O(1)$

✗

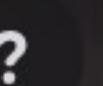
$\theta(\log n)$

✓

$O(n)$

✓

$\theta(n)$



# Kolejka

- Implementowana przez *queue.Queue* lub *collections.deque*
- Czym jest *deque*?
- Jaka jest złożoność dodawania i usuwania z obu końców?

# Jak działa *list* i *deque* w środku?

Dodawanie i usuwanie na końcu listy czy  
też z obu stron kolejki jest szybkie.

Czasami jednak na chwilę spowalnia.

Czemu?

## Prealokacja pamięci

- Żeby pisać w pamięci, musimy zarezerwować sobie pewien fragment
- Lista rezerwuje na początku pewien fragment
- Gdy brakuje miejsca na dodanie kolejnego elementu, przenosimy wszystko do nowego, większego fragmentu
- Jeżeli przeniesiemy  $n$  elementów do fragmentu o długości  $2n$  wykonamy  $n$  operacji
- Ale kolejne  $n$  elementów dodajemy pojedynczymi operacjami
- Łącznie wykonamy  $2n$  operacji na dodanie  $n$  elementów
- Średnio  $O(1)$
- Jest to tak zwany czas amortyzowany



# Struktury hashowane

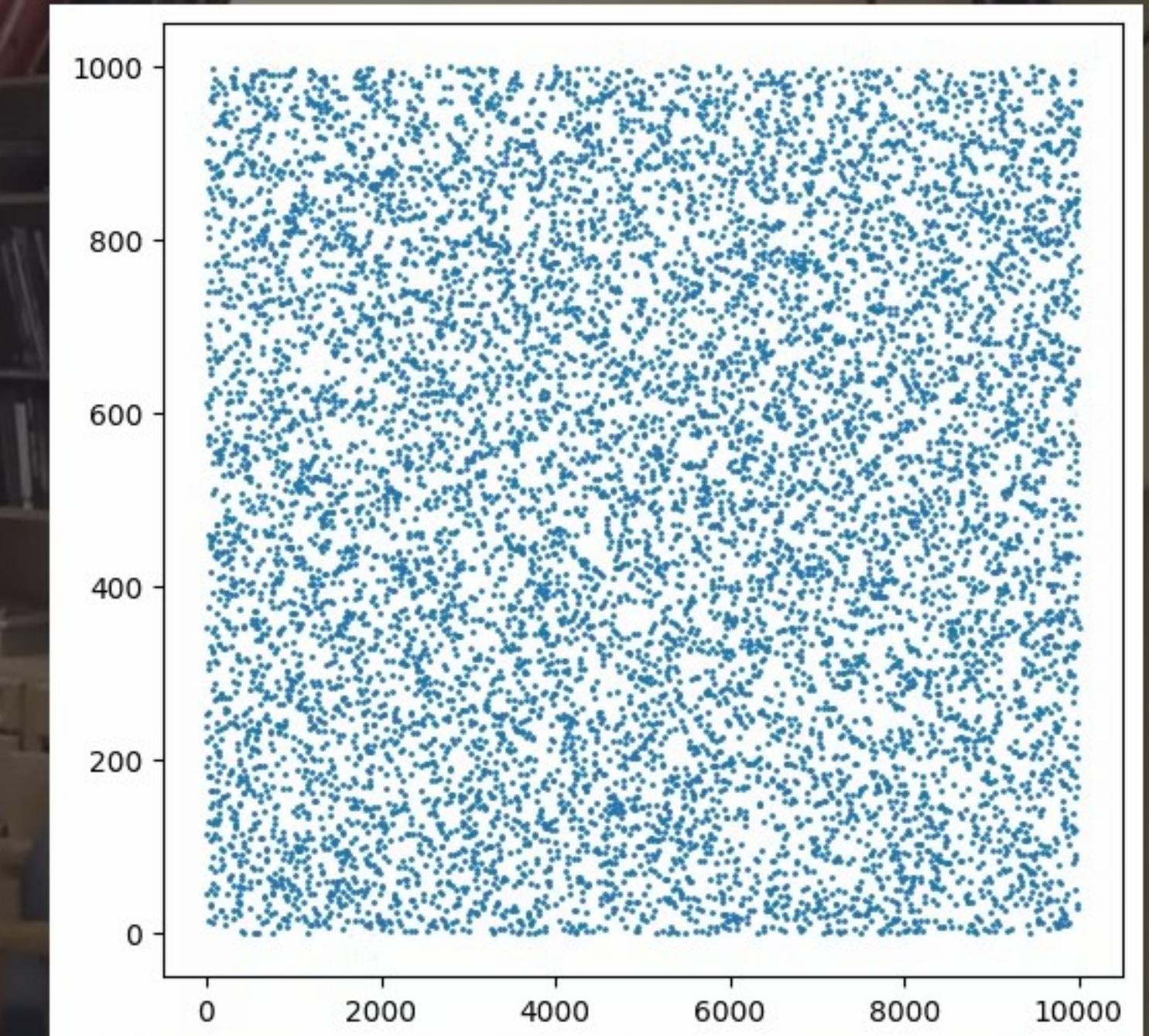
# Czy znacie pojęcie *hashowania*?

1  
TAK

4  
NIE

## Czym więc jest hashowanie?

- Funkcja, zwracająca dla danego obiektu charakterystyczną wartość
- Wynik powinien być nieodwracalny
- Rozkład wyników powinien być jak najbardziej losowy i równomierny
- Przykładowo  $x \cdot ((x + 97) - (x + 3)) \bmod 1001$
- Nadaje się do szybkiej weryfikacji wersji pliku
- Świetne narzędzie do przechowywania haseł
- Jak ma się to do struktur?



# Zbiór

- Nieuporządkowana kolekcja unikatowych elementów
- Elementy dodajemy, usuwamy i sprawdzamy w czasie  $O(1)$
- W dużym uproszczeniu to zwykła tablica
- Ale elementy trafiają pod indeksy będące ich hashami
- Istnieje też niemodyfikowalny wariant **frozenset**



# Czym jest mapa?

- W Pythonie implementowana przez **dict**
- Zbiór par klucz-wartość, gdzie hashowane są tylko klucze
- Pamięta kolejność dodawania elementów (o tym będzie później)

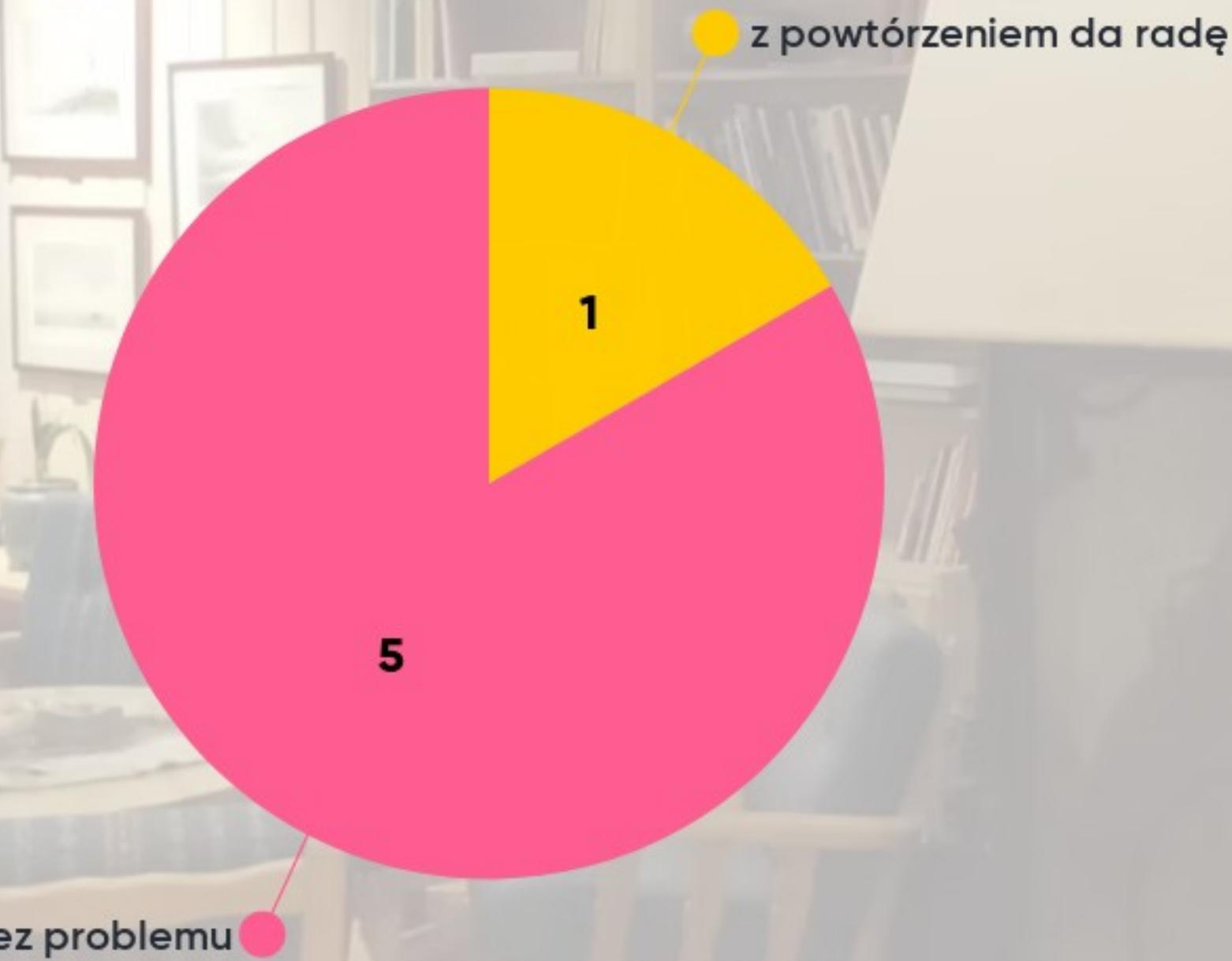
<a href="#"><u>namedtuple()</u></a>	factory function for creating tuple subclasses with named fields
<a href="#"><u>deque</u></a>	list-like container with fast appends and pops on either end
<a href="#"><u>ChainMap</u></a>	dict-like class for creating a single view of multiple mappings
<a href="#"><u>Counter</u></a>	dict subclass for counting hashable objects
<a href="#"><u>OrderedDict</u></a>	dict subclass that remembers the order entries were added
<a href="#"><u>defaultdict</u></a>	dict subclass that calls a factory function to supply missing values
<a href="#"><u>UserDict</u></a>	wrapper around dictionary objects for easier dict subclassing
<a href="#"><u>UserList</u></a>	wrapper around list objects for easier list subclassing
<a href="#"><u>UserString</u></a>	wrapper around string objects for easier string subclassing

## Pakiet collections

# Co warto pamiętać

1. Wskaźniki i alokacja
2. LIFO vs FIFO
3. deque
4. Hashowanie
5. Zbiór
6. Słownik

# Jak się czujecie z tym tematem?





# Pakiet *numpy*