Python

Algorytmy i struktury danych

Struktury danych (Python)

Struktury danych - krotka (tuple)

- Krotkę poznaliśmy już nieco wcześniej ale czas aby poznać się oficjalnie :)
- Krotka jest bardzo podobna do listy - również przechowuje elementy w uporządkowanej kolejności i umożliwia dostanie się do nich po indeksie.
- Raz utworzonej krotki nie można jednak zmienić, nie posiada metod takich jak **append** czy **extend**, znanych z listy.
- Krotkę zapisujemy tak jak listę, z tą różnicą, że zamiast kwadratowych nawiasów używamy okrągłych.
- Jednoelementowa krotka musi mieć przecinek po swoim jedynym elemencie.
- Jeśli funkcja zwraca kilka elementów to tak naprawdę zwraca krotkę, w tym przypadku nie trzeba po **return** pisać nawiasów.
- Podobnie w przypisaniu nawiasy można pominąć.

```
In [1]: empty_tuple = tuple()

In [2]: single = 1,

In [3]: pair = 1, 2

In [4]: triple = 1, 2, 3

In [5]: triple[1]
Out[5]: 2

In [6]: def returns_tuple():
   ...:     return 1, 2, 3
   ...:

In [7]: returns_tuple()
Out[7]: (1, 2, 3)

In [8]: first, *rest = triple

In [9]: first
Out[9]: 1
```

Autor: Michał Nowotka

Struktury danych - mutable vs immutable

- Krotka jest tylko jednym z przykładów niemodyfikowalnych struktur danych. Łańcuchy znaków (string) również zaliczają się do tej grupy.
- Ważne aby uświadomić sobie, że zmienna, do której przepisaliśmy niemodyfikowalną strukturę danych nadal może być przypisana do zupełnie innej rzeczy.
- W takim przypadku nie zmodyfikowaliśmy zawartości zmiennej tylko sprawiliśmy, że wskazuje na inne miejsce w pamięci.
- Natomiast jeśli spróbujemy zmodyfikować np. pojedynczą literę w ciągu znaków to ta operacja się nie powiedzie.

```
In [1]: immutable_string = 'ala'

In [2]: id(immutable_string)
Out[2]: 4333410320

In [3]: immutable_string = 'ma kota'

In [4]: id(immutable_string)
Out[4]: 4333468728

In [5]: immutable_string[0] = 'M'
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-5-53e8a04047d2> in <module>
----> 1 immutable_string[0] = 'M'

TypeError: 'str' object does not support item assignment
```

Autor: Michał Nowotka

Struktury danych - mutable vs immutable

- Można zapytać jakie są konsekwencje niemodyfikowalności, poza oczywistym faktem, że danej struktury nie da się zmodyfikować w miejscu.
- Jedną z konsekwencji jest zużycie pamięci.
- Na przykładzie obok “wydłużamy” zmienną output o kolejne elementy z listy.
- Faktyczne jednak przy każdym obiegu pętli powstaje nowy obiekt a my ustawiamy jedynie zmienną **output** na ten, który obecnie jest najdłuższy.
- Pozostałe, niepotrzebne elementy zajmują miejsce w pamięci co miałoby duże znaczenie gdyby lista zawierała miliony elementów.

```
In [1]: employees = ['Ania', 'Kasia', 'Wojtek', 'Ewa', 'Ola', 'Heniek']

In [2]: output = ''

In [3]: for employee in employees:
   ...:     output += ' ' + employee
   ...:     print(id(output))
   ...:
4505317592
4503050224
4505350192
4505350192
4504785744
4502776976

In [4]: output
Out[4]: ' Ania Kasia Wojtek Ewa Ola Heniek'
```

Autor: Michał Nowotka

Struktury danych - mutable vs immutable

Typy modyfikowalne:

- Listy, zbiory, słowniki

Typy niemodyfikowalne:

- Krotki, frozensety, liczby, stringi, wartości boolowskie

Struktury danych - mutable vs immutable

Sprawdź czy rozumiesz poniższe operacje. Jaki będzie wynik?

1. matrix = [[1,2,3], [4,5,6], [7,8,9]]
2. matrix[1]
3. matrix[1][2]
4. matrix.append("to nie jest liczba")
5. matrix[: 2] = 2
6. matrix[: 2] = [1, 2, 3]
7. matrix[: 2] = [1]
8. matrix + matrix

Struktury danych - mutable vs immutable

Sprawdź czy rozumiesz poniższe operacje. Jaki będzie wynik?

1. krotka = 1, 2, 3, "4", True
2. krotka[2]
3. krotka[3] = 4
4. krotka[:] = 1,2,3
5. krotka + krotka

Struktury danych - mutable vs immutable

Sprawdź czy rozumiesz poniższe operacje. Jaki będzie wynik?

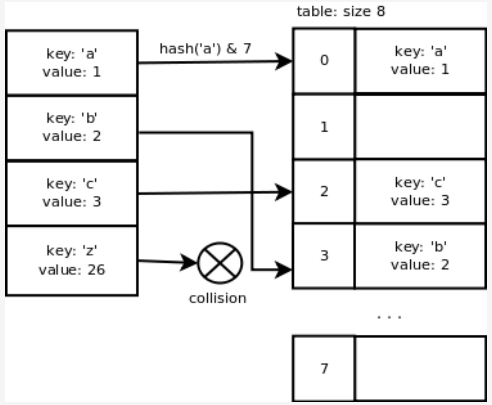
1. name = ’’Jose’’
2. name + " " + "Antonio"
3. name[0] + " " + "Morales"
4. name[len(name)-1]
5. name[-1]
6. name[0] = "H"

Struktury danych - mutable vs immutable

Sprawdź czy rozumiesz poniższe operacje. Jaki będzie wynik?

1. name + matrix
2. matrix + name
3. name + name
4. name * 2
5. name[0] + name[1] + name[2] + name[3]

Jak działa słownik - haszowanie a typy immutable



- Oczywiście niemodyfikowalność ma również swoje zalety.
- Jedną z nich jest możliwość użycia struktury jako klucza w słowniku.
- Możnaby zapytać dlaczego kluczami w słowniku mogą być jedynie typy immutable.
- Aby zrozumieć odpowiedź należy najpierw zastanowić się jak działa słownik w Pythonie.
- Podstawą działania słownika jest funkcja haszująca.
- Funkcja haszująca posiada następujące właściwości:
  - Zwraca wartość, która jest liczbą całkowitą
  - Oblicza się ją szybko, w stałym czasie niezależnie od elementu, na którym liczy się hash.
  - Dwa takie same elementy posiadają tę samą wartość hashu.
  - Dwa różne elementy, choćby różniły się od siebie bardzo niewiele, będą miały dwie zupełnie różne wartości hashu, w żaden sposób ze sobą niezwiązane.
- Aby obliczyć wartość funkcji haszującej na elemencie w Pythonie, należy użyć wbudowanej funkcji **hash**.
- Pod spodem interpreter wywołuje metodę __**hash**__ klasy, z której pochodzi dany obiekt.
- Dla wszystkich podstawowych typów funkcja __**hash**__ jest już zdefiniowana.
- Żeby wiedzieć gdzie umieścić dany element w słowniku, na kluczu obliczana jest wartość funkcji haszującej.
- Następnie liczy się **hash % rozmiar słownika** i w ten sposób wylicza się indeks pod który należy wstawić element.

Autor: Michał Nowotka

Jak działa słownik - haszowanie a typy immutable

- Gdyby kluczem był modyfikowalny element, po jego modyfikacji otrzymalibyśmy inną wartość funkcji haszującej i znalezienie elementu w słowniku byłoby niemożliwe.
- Dokładne działanie Pythonowego słownika opisuje artykuł.
- Wbrew pozorom używanie jako klucza czegoś innego niż string nie jest niczym niezwykłym, można w ten sposób reprezentować wielowymiarowe plansze do gier albo grafy.

```
In [1]: board = {}

In [2]: board[0,0] = False

In [3]: hash('ala ma kota')
Out[3]: 4831261069132572937

In [4]: hash('ala ma kota')
Out[4]: 4831261069132572937

In [5]: hash('ala ma koty')
Out[5]: 5029174871914192939

In [6]: hash(1)
Out[6]: 1

In [7]: hash(2)
Out[7]: 2

In [8]: hash([])
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-8-4c7351eba020> in <module>
----> 1 hash([])

TypeError: unhashable type: 'list'
```

```
In [1]: tic_tac_toe_board = {}

In [2]: for row in range(3):
   ...:     for col in range(3):
   ...:         tic_tac_toe_board[(row, col)] = False
   ...:

In [3]: tic_tac_toe_board
Out[3]:
{(0, 0): False,
 (0, 1): False,
 (0, 2): False,
 (1, 0): False,
 (1, 1): False,
 (1, 2): False,
 (2, 0): False,
 (2, 1): False,
 (2, 2): False}
```

Autor: Michał Nowotka

Kopiowanie słownika

Napisz kod, który na podstawie wcześniej utworzonego słownika, utworzy jego kopię.

Kopiowanie słownika - copy vs deepcopy

- Aby utworzyć kopię słownika można użyć jego metody o nazwie **copy**.
- Utworzona kopia będzie tak zwaną płytką kopią to znaczy, że powstanie nowy obiekt ale będzie on posiadał jedynie referencje do wartości trzymanych w oryginalnym słowniku.
- Można to zobaczyć na przykładzie obok - zarówno oryginalny słownik jak i jego kopia przechowują referencję do tych samych list. Jeśli zmodyfikujemy listę w oryginalnym słowniku zobaczymy również zmianę w jego kopii.

```
In [1]: my_dict = {'a': [1, 2, 3], 'b': [4, 5, 6]}

In [2]: my_copy = my_dict.copy()

In [3]: my_copy
Out[3]: {'a': [1, 2, 3], 'b': [4, 5, 6]}

In [4]: my_dict['a'].append(4)

In [5]: my_dict
Out[5]: {'a': [1, 2, 3, 4], 'b': [4, 5, 6]}

In [6]: my_copy
Out[6]: {'a': [1, 2, 3, 4], 'b': [4, 5, 6]}
```

Autor: Michał Nowotka

Kopiowanie słownika - copy vs deepcopy

- Głęboka kopia wykonuje nie tylko kopię słownika ale wszystkich jego wartości.
- Odtąd referencje w skopiowanym słowniku wskazują na osobne obiekty.
- Funkcjonalności tworzenia głębokiej kopii dostarcza moduł **copy**.
- Na przykładzie możemy zobaczyć jak zmiana dokonana na liście wchodzącej w skład oryginalnego słownika nie wpływa na jego kopię.

```
In [1]: from copy import deepcopy

In [2]: my_dict = {'a': [1, 2, 3], 'b': [4, 5, 6]}

In [3]: my_copy = deepcopy(my_dict)

In [4]: my_copy
Out[4]: {'a': [1, 2, 3], 'b': [4, 5, 6]}

In [5]: my_dict['a'].append(4)

In [6]: my_dict
Out[6]: {'a': [1, 2, 3, 4], 'b': [4, 5, 6]}

In [7]: my_copy
Out[7]: {'a': [1, 2, 3], 'b': [4, 5, 6]}
```

Autor: Michał Nowotka

Kopiowanie słownika

Napisz funkcję add_dict(dict1, dict2) generującą złączenie dwóch słowników. Funkcja powinna zwracać nowy słownik zawierający wszystkie wartości z obydwu argumentów (przyjmuje się, że są to słowniki). Jeśli w każdym z argumentów występuje ten sam klucz, w wyniku może znaleźć się dowolna z wartości.

Struktury danych

Napisz funkcję wyliczającą ilość poszczególnych znaków w napisie.

Struktury danych - defaultdict

- Wróćmy na chwilę do słownika - wyobraźmy sobie, że chcemy policzyć ilość wystąpień każdej litery w tekście.
- W takim przypadku słownik wydaje się być idealną strukturą danych - kluczami będą litery alfabetu a wartościami liczby wystąpień.
- Powstaje jednak pewien problem. Na początku słownik jest pusty i kiedy po raz pierwszy wkładamy do niego nową literę musimy jako wartość umieścić tam jedynkę a za każdym kolejnym razem zwiększamy aktualną wartość ze słownika o 1. O ile piękniej byłoby gdybyśmy mogli pozbyć się konstrukcji **if/else**...

```
In [1]: def count_letters(text):
   ...:     counter = {}
   ...:     for letter in text:
   ...:         if letter not in counter:
   ...:             counter[letter] = 1
   ...:         else:
   ...:             counter[letter] += 1
   ...:     return counter
   ...:

In [2]: count_letters("ala ma kota")
Out[2]: {'a': 4, 'l': 1, ' ': 2, 'm': 1, 'k': 1, 'o': 1, 't': 1}
```

Autor: Michał Nowotka

Struktury danych - defaultdict

- Z pomocą przychodzi nowa struktura danych: **defaultdict** ze standardowego modułu **collections**.
- Jak sama nazwa wskazuje, jeśli odnosimy się do klucza, który nie istnieje to brakująca wartość zostanie zastąpiona domyślną.
- Jak widać na poniższym przykładzie, aby stworzyć nowy obiekt **defaultdict**, należy podać nazwę funkcji-fabryki, która zostanie wywołana aby stworzyć domyślny element. Jeśli wywołamy **int()**, dostaniemy zero.

```
In [1]: from collections import defaultdict

In [2]: def count_letters(text):
   ...:     counter = defaultdict(int)
   ...:     for letter in text:
   ...:         counter[letter] += 1
   ...:     return counter
   ...:

In [3]: count_letters("ala ma kota")
Out[3]: defaultdict(int, {'a': 4, 'l': 1, ' ': 2, 'm': 1, 'k': 1, 'o': 1, 't': 1})
```

Autor: Michał Nowotka

Struktury danych - defaultdict

- **defaultdict** świetnie nadaje się do grupowania elementów. Wyobraźmy sobie, że mamy kolekcję par, w której pierwszy element pary jest etykietą a drugi wartością. Chcemy zgrupować wszystkie elementy, które posiadają tę samą etykietę.
- W takim przypadku stworzymy obiekt **defaultdict**, którego domyślnym elementem jest pusta lista.

```
In [1]: from collections import defaultdict

In [2]: pairs = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]

In [3]: d = defaultdict(list)

In [4]: for k, v in pairs:
   ...:     d[k].append(v)
   ...:

In [5]: d
Out[5]: defaultdict(list, {'yellow': [1, 3], 'blue': [2, 4], 'red': [1]})
```

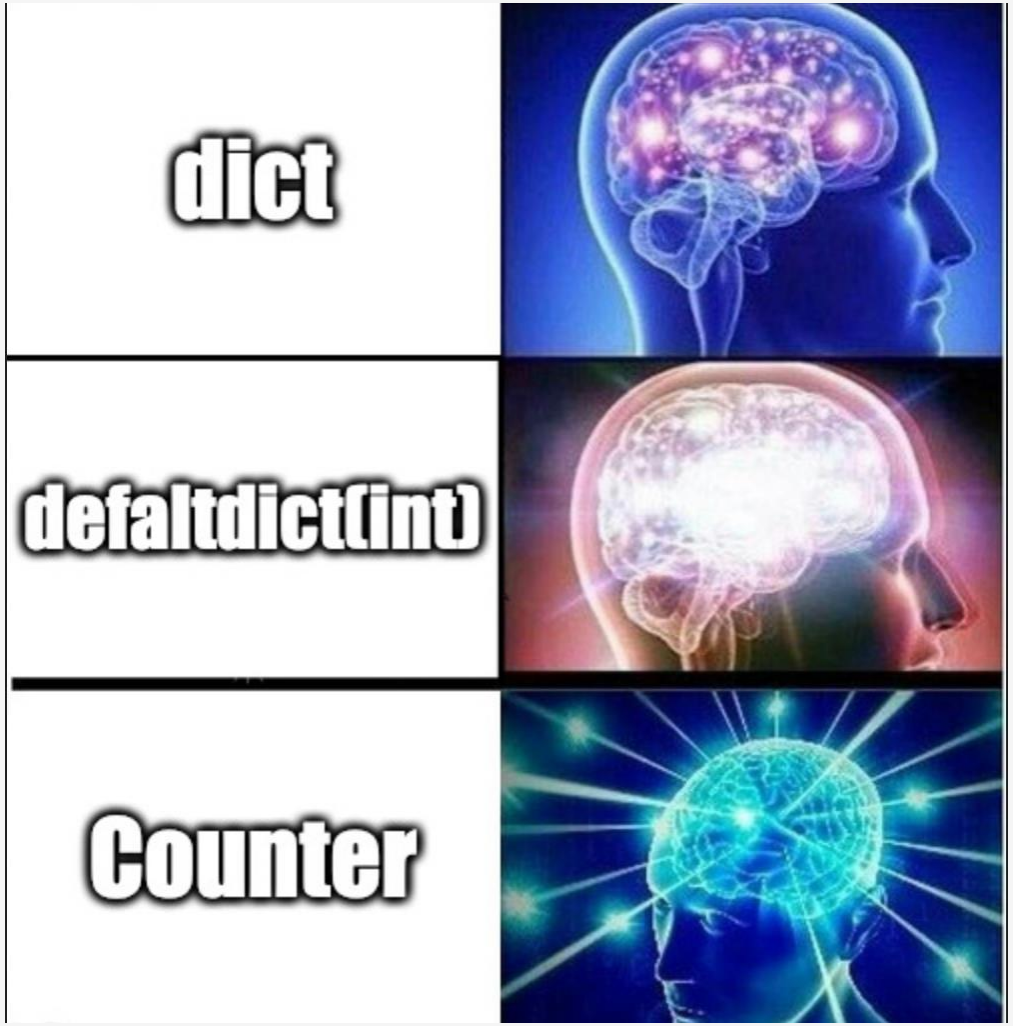
Struktury danych - Counter

- Wróćmy na chwilę do zliczania wystąpień liter w tekście.
- Istnieje jeszcze jedna struktura, która sprawia że ta czynność jest jeszcze prostsza. Jest nią **Counter** ze standardowego modułu **collections**.
- **Counter** przyjmuje dowolną kolekcję (listę, krotkę, string) a jest typem słownika, którego kluczami są elementy wejściowej kolekcji, a wartościami ilości wystąpień.
- **Counter** posiada wszystkie metody znane ze słownika jak również metodę **most_common(n)**, zwracającą **n** najczęściej występujących elementów.

```
In [1]: from collections import Counter

In [2]: def count_letters(text):
   ...:     return Counter(text)
   ...:

In [3]: result = count_letters("ala ma kota")

In [4]: result
Out[4]: Counter({'a': 4, 'l': 1, ' ': 2, 'm': 1, 'k': 1, 'o': 1, 't': 1})

In [5]: result.most_common(1)
Out[5]: [('a', 4)]
```

Autor: Michał Nowotka

Struktury danych - Counter



Autor: Michał Nowotka

Typy mutable jako domyślne wartości funkcji

- Używanie modyfikowalnego typu (np listy) jako wartość domyślna argumentu w funkcji jest groźna i uważana za programistyczny błąd.
- W przykładzie obok mamy funkcję, która dodaje nowego pracownika do listy pracowników.
- Drugi argument ma domyślną wartość - jest nią pusta lista, zatem oczekujemy, że jeśli nie podamy listy to zostanie utworzona nowa pusta lista i do niej dodany nowy pracownik.
- Tak dzieje się jedynie za pierwszym razem. Podczas kolejnych wywołań funkcji bez podawania drugiego argumentu wartość domyślna nie jest już pustą listą.
- Okazuje się, że Python ponownie używa tej samej listy jako wartość domyślną a ponieważ zmodyfikowaliśmy ją w poprzednich wywołaniach to nie jest ona już pusta.

```
In [1]: def add_employee(emp, emp_list=[]):
   ...:     emp_list.append(emp)
   ...:     print(emp_list)
   ...:

In [2]: emps = ['Ania', 'Ola']

In [3]: add_employee('Maciek', emps)
['Ania', 'Ola', 'Maciek']

In [4]: add_employee('Staszek')
['Staszek']

In [5]: add_employee('Magda')
['Staszek', 'Magda']

In [6]: add_employee('Igor')
['Staszek', 'Magda', 'Igor']
```

Autor: Michał Nowotka

Typy mutable jako domyślne wartości funkcji

- Przykład obok pokazuje w jaki sposób możemy naprawić naszą funkcję.
- Zamiast ustawiać domyślną wartość na pustą listę ustawiamy ją na **None** a w ciele funkcji sprawdzamy czy drugi argument jest równy **None** czy nie.
- Jeśli jest równy **None** to dopiero wtedy przypisujemy do niego pustą listę.
- Poprawiona funkcja działa tak, jak można by tego oczekiwać - przy każdym wywołaniu bez drugiego argumentu zwraca jednoelementową listę zawierającą jedynie pierwszy argument.

```
In [1]: def add_employee(emp, emp_list=None):
   ...:     if not emp_list:
   ...:         emp_list = []
   ...:     emp_list.append(emp)
   ...:     print(emp_list)
   ...:

In [2]: emps = ['Ania', 'Ola']

In [3]: add_employee('Maciek', emps)
['Ania', 'Ola', 'Maciek']

In [4]: add_employee('Staszek')
['Staszek']

In [5]: add_employee('Magda')
['Magda']

In [6]: add_employee('Igor')
['Igor']
```

Autor: Michał Nowotka

Struktury danych - namedtuple

- Wiemy już czym jest krotka i na czym polega różnica pomiędzy typami **mutable** a **immutable**.
- Czas na kolejną strukturę danych: nazwaną krotkę czyli **namedtuple**.
- **namedtuple** również jest immutable oraz, jak sama nazwa wskazuje, jest bardzo podobny do krotki.
- Różnica polega na tym, że aby odnieść się do elementu krotki trzeba użyć indeksu (który nic nie mówi o przeznaczeniu danego elementu) tymczasem w przypadku **namedtuple** można odnieść się do elementu poprzez jego nazwę.

```
In [1]: number_info_tuple = ('697120906', '+48', '-')

In [2]: area_code = number_info_tuple[1]

In [3]: from collections import namedtuple

In [4]: NumberInfo = namedtuple('NumberInfo', 'number area_code delimiter')

In [5]: number_info_namedtuple = NumberInfo('697120906', '+48', '-')

In [6]: number_info_namedtuple.area_code
Out[6]: '+48'
```

Struktury danych - namedtuple

- Jak widać na przykładzie, **namedtuple** jest funkcją-fabryką, która zwraca klasę.
- Pierwszym argumentem funkcji **namedtuple** jest nazwa tworzonej klasy.
- Drugim argumentem jest lista nazw pól w postaci ciągu znaków gdzie pola są rozdzielone spacjami bądź listy.
- Po stworzeniu klasy można tworzyć jej obiekty podając w konstruktorze wartości pól, w kolejności w jakiej zostały podane na liście podczas tworzenia klasy.

```
In [1]: number_info_tuple = ('697120906', '+48', '-')

In [2]: area_code = number_info_tuple[1]

In [3]: from collections import namedtuple

In [4]: NumberInfo = namedtuple('NumberInfo', 'number area_code delimiter')

In [5]: number_info_namedtuple = NumberInfo('697120906', '+48', '-')

In [6]: number_info_namedtuple.area_code
Out[6]: '+48'
```

Struktury danych - dataclass

- **dataclass** jest dekoratorem umożliwiającym tworzenie klas zorientowanych głównie na przechowywanie danych a więc podobnych do **namedtuple**.
- Użycie tego dekoratora pozwala między innymi zaoszczędzić konieczności dostarczania implementacji specjalnej metody **__init__**.
- Podobnie nie jest konieczne dostarczanie metody **__repr__**.
- **dataclass** wymusza stosowanie adnotacji typów choć oczywiście typy nie są sprawdzane, chyba że użyjemy narzędzi takich jak **mypy**.
- Zaletą **dataclass** jest fakt, że mamy do czynienia z normalną klasą, do której możemy dodać metody.
- Ponadto możemy podać wartości domyślne pól, co jest trudne do uzyskania w przypadku **namedtuple**.
- Pewnym mankamentem jest dostępność - dekorator **dataclass** został wprowadzony dopiero w wersji 3.7 Pythona.

Autor: Michał Nowotka

Struktury danych - dataclass

- Pola klasy definiujemy jako atrybuty klasowe, obowiązkowo z adnotacją typu.
- Nie musimy już pisać konstruktora, który w przypadku tak dużej ilości pól byłby dość rozwlekły.
- Po wydrukowaniu obiektu w konsoli widzimy przyjazną, zrozumiałą reprezentację pozwalającą zrozumieć jakie wartości kryją się w polach.

```
import dataclasses
import datetime
import typing


@dataclasses.dataclass(frozen=True)
class UserData:
    first_name: str
    last_name: str
    email: str
    state_province: str
    address_line_1: str
    city: str
    country: str
    postal_code: str
    date_of_birth: typing.Optional[datetime.datetime] = None

## -- End pasted text --

In [2]: user = UserData('Maciej', 'Tarnowski', 'maciej@gmail.com',
   ...: 'Mazowieckie', 'Kapitulna 9', 'Warszawa', 'Polska', '01-123')

In [3]: print(user)
UserData(first_name='Maciej', last_name='Tarnowski', email='maciej@gmail.com', state_province='Mazowie
kie', address_line_1='Kapitulna 9', city='Warszawa', country='Polska', postal_code='01-123', date_of_b
rth=None)
```

Autor: Michał Nowotka

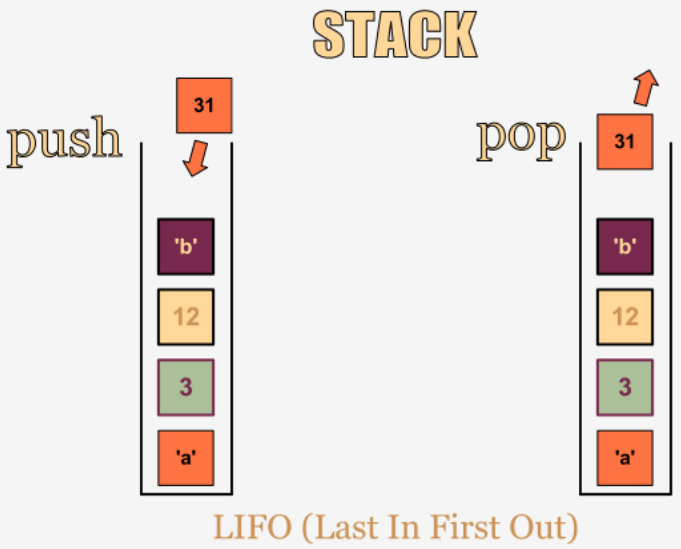
Struktury danych - dataclass

Autor: Michał Nowotka

Struktury danych

Struktury danych - stos

- Stos jest abstrakcyjną strukturą danych - w Pythonie nie ma osobnego typu dla stosu.
- Mimo to stos jest bardzo ważną koncepcją, którą warto znać.
- Stos przypomina podajnik na talerze znany z niektórych restauracji albo rurę z kulkami.
- Nowy element można dodać jedynie na wierzchołek stosu - ta operacja nazywa się **push**.
- Jeśli pobieramy element ze stosu to również jedynie ten, który znajduje się na jego szczycie, ta operacja nazywa się **pop**.
- Pierwszym elementem, który zdejmujemy ze stosu będzie ten, który dostał się tam jako ostatni, z ang. **L**ast **I**n, **F**irst **O**ut (**LIFO**).



Autor: Michał Nowotka

Struktury danych - lista jako stos

- Pomimo, że Python nie posiada dedykowanej struktury dla stosu to z powodzeniem można zostąpić go listą.
- Lista posiada metodę **pop**, która usuwa ostatni element listy i zwraca go jako wynik funkcji.
- Lista posiada również metodę **append**, która dodaje na jej koniec element - ta metoda odpowiada operacji **push**.
- Jeśli ograniczymy się jedynie do używania metod **pop** i **append** na liście, efektywnie będziemy ją traktować jak stos.
- Obie operacje - **push** (**append**) i **pop** są w przypadku listy bardzo szybkie, wykonują się w stałym czasie.

```
In [1]: stack = []

In [2]: stack.append('first')

In [3]: stack.append('second')

In [4]: stack.append('last')

In [5]: stack
Out[5]: ['first', 'second', 'last']

In [6]: stack.pop()
Out[6]: 'last'

In [7]: stack
Out[7]: ['first', 'second']

In [8]: stack.pop()
Out[8]: 'second'

In [9]: stack
Out[9]: ['first']

In [10]: stack.pop()
Out[10]: 'first'

In [11]: stack
Out[11]: []
```

Autor: Michał Nowotka

Struktury danych - stos

Napisz klasę **`Stack`** reprezentującą stos. Obiekty tej klasy powinny przechowywać listę danych i udostępniać metody **`push`** i **`pop`** do dorzucania i wyciągania z niego wartości. Stos powinien posiadać również atrybut **`length`** (reprezentujący jego aktualny rozmiar), który powinien się zmieniać wraz z wykonywaniem operacji **`push`** i **`pop`**.

Struktury danych - kolejka

- Podczas gdy w przypadku stosu ostatni dodany element jest pierwszym, który opuści stos, w kolejce jest inaczej - pierwszy dodany element opuści ją też jako pierwszy. Jest to zasada **F**irst **I**n, **F**irst **O**ut (**FIFO**).
- Kolejka również jest abstrakcyjnym typem danych jednak nie można jej zaimplementować w Pythonie przy użyciu listy, ponieważ usunięcie elementu z jej początku (albo dodanie elementu do początku listy) jest kosztowną operacją, wymagającą przekopiowania wszystkich elementów na liście a więc trwającą proporcjonalnie do długości listy.

Autor: Michał Nowotka

Struktury danych - deque jako kolejka

- Do implementacji kolejki w Pythonie służy typ **deque** dostępny w module **collections.**
- **deque** posiada metody znane z listy to znaczy **append** i **pop.**
- Posiada również metody **popleft** i **appendleft.**
- Aby uzyskać zachowanie kolejki należy używać metod **append** i **popleft** lub **appendleft** i **pop.**

```
In [1]: from collections import deque

In [2]: queue = deque()

In [3]: queue.append('first')

In [4]: queue.append('second')

In [5]: queue.append('last')

In [6]: queue
Out[6]: deque(['first', 'second', 'last'])

In [7]: queue.popleft()
Out[7]: 'first'

In [8]: queue
Out[8]: deque(['second', 'last'])

In [9]: queue.popleft()
Out[9]: 'second'

In [10]: queue
Out[10]: deque(['last'])

In [11]: queue.popleft()
Out[11]: 'last'

In [12]: queue
Out[12]: deque([])
```

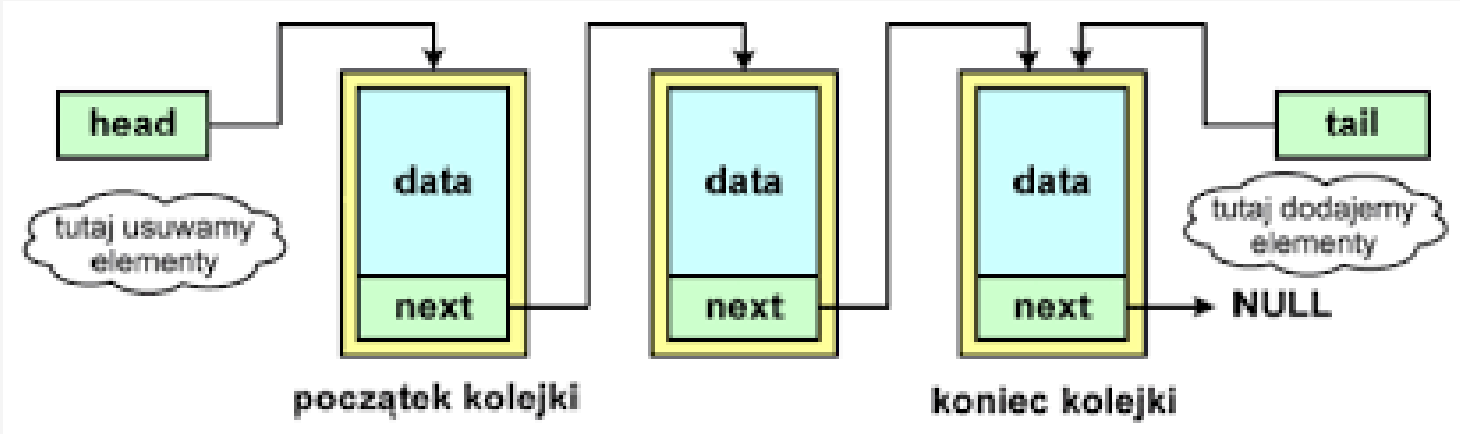
Autor: Michał Nowotka

Struktury danych - kolejka

Zaimplementuj klasę **`FifoQueue`**. Powinna korzystać ze struktury **`deque`** oraz posiadać metody **`append`** oraz **`pop`**.

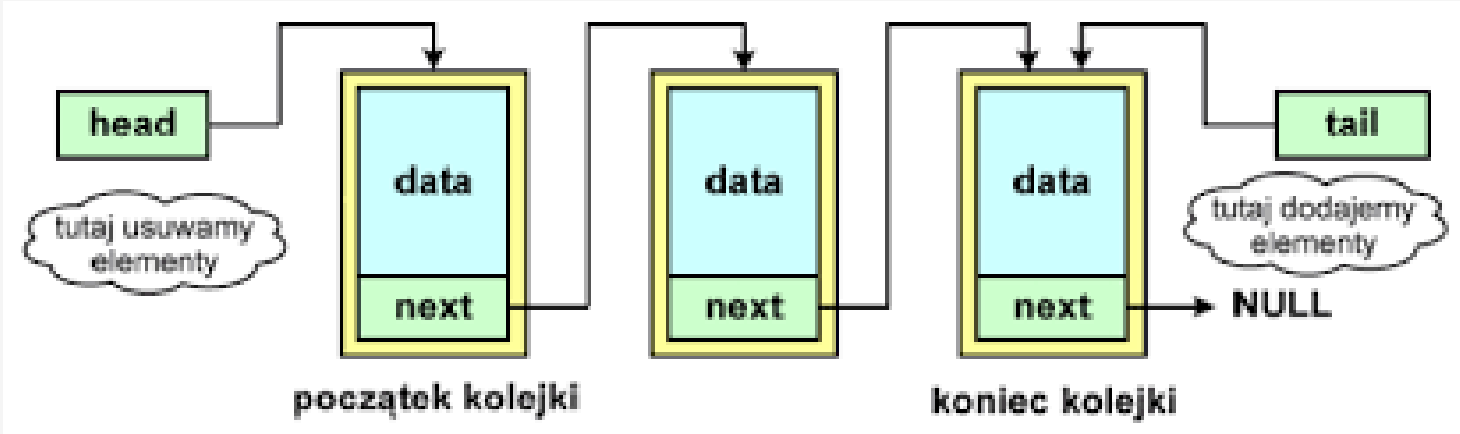
Struktury danych - lista

- **Lista** jest sekwencyjną strukturą danych, która składa się z ciągu elementów tego samego typu.
- Z danego elementu listy możemy przejść tylko do elementu następnego (lista jednokierunkowa) albo do następnego lub poprzedniego (lsita dwukierunkowa).
- Dojście do elementu *i*-tego wymaga przejścia przez kolejne elementy od pierwszego do docelowego.
- Pojedynczy element listy składa się z danych (data), wskaźnika na następny element (next) i ewentualnie wskaźnika na element poprzedni (prev – tylko w przypadku lsit dwukierunkowych).
- Tworząc listę zwykle dodaje się dwa dodatkowe wskaźniki: ***head*** (wskazuje pierwszy element listy) oraz ***tail*** (wskazuje ostatni element). Do zliczania długości listy wykorzystuje się licznik ***count*** (inkrementowany z każdym dodaniem nowego elementu).



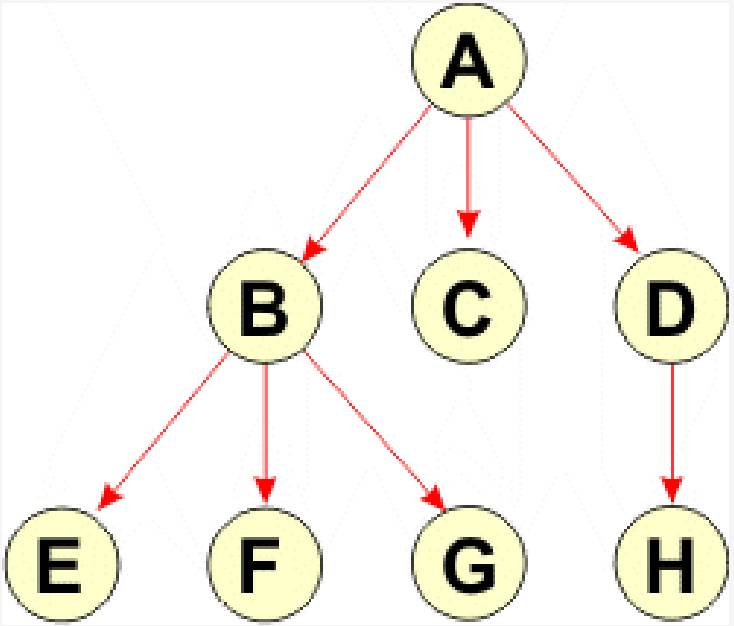
Struktury danych - lista

- Elementy listy nie muszą leżeć obok siebie w pamięci. Zatem lista nie wymaga ciągłego obszaru pamięci i może być rozłożona w różnych jej segmentach.
- Nowe elementy można szybko dołączać w dowolnym miejscu listy, zatem lista może dynamicznie rosnąć w pamięci. Również z listy można usuwać dowolne elementy, co powoduje, iż lista kurczy się w pamięci.
- Jeżeli chcemy odnaleźć konkretny element w liście, musimy przechodzić od pierwszego elementu (head) do ostatniego (tail) do momentu aż odnajdziemy poszukiwane dane. Nie istnieje możliwość dotarcia do elementu poprzez indeks.

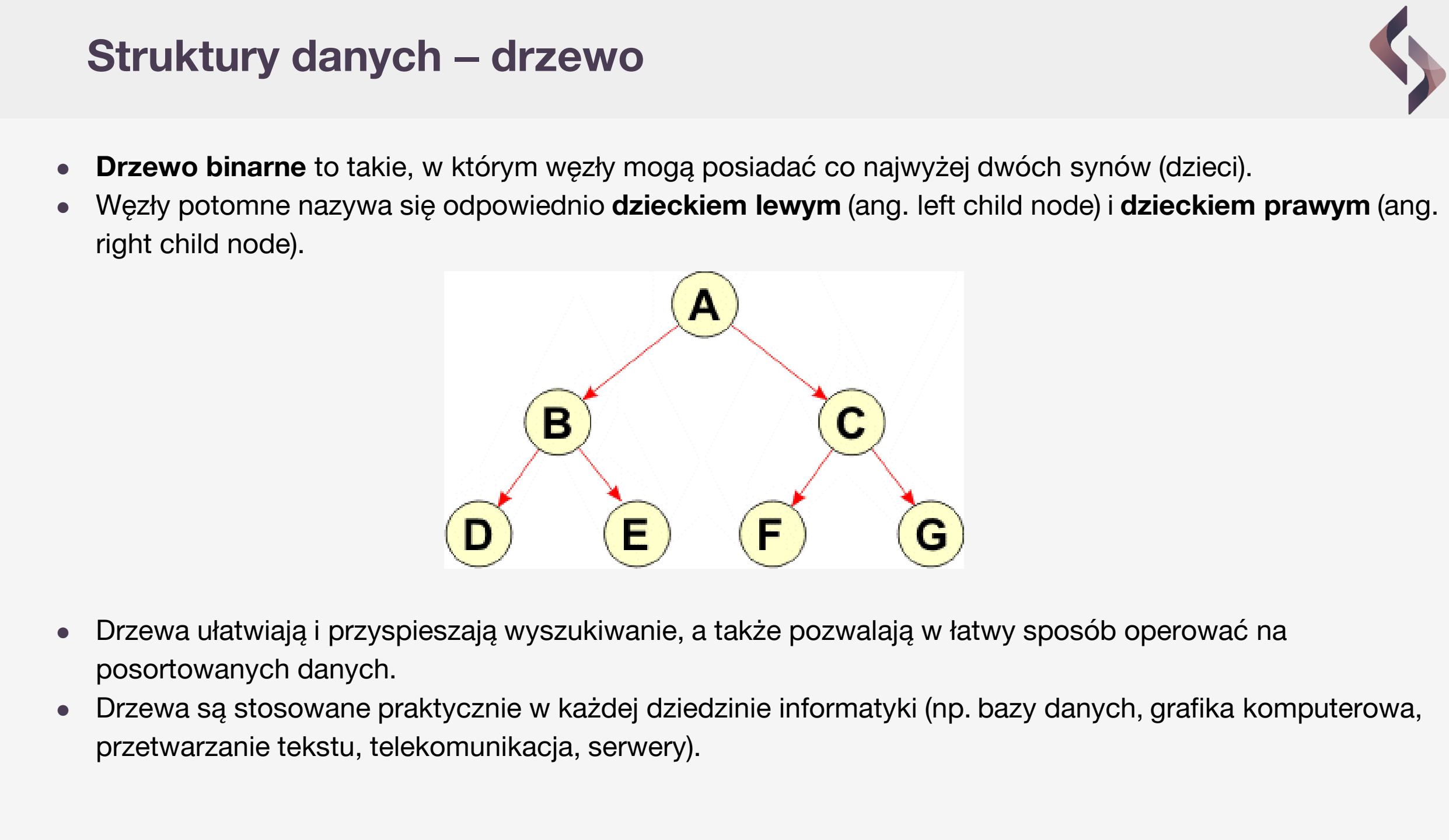


Struktury danych - drzewo

- **Drzewo** to struktura zbudowana z **węzłów** (ang. node).
- Węzły przechowują dane, są ze sobą powiązane w sposób hierarchiczny za pomocą **krawędzi** (ang. edge), reprezentowanych przez strzałki. Pierwszy węzeł drzewa nazywa się **korzeniem** (ang. root node, na rysunku: A).
- Z korzenia odchodzą pozostałe węzły, które będziemy nazywać **dziećmi** (ang. child node). Synowie są węzłami podrzędnymi w strukturze hierarchicznej. Synowie tego samego ojca są nazywani **braćmi** (ang. sibling node). Węzeł nadrzędny w stosunku do syna nazwiemy **ojcem** (ang. parent node).
- Jeśli węzeł nie posiada dzieci, to nazywa się **liściem** (ang. leaf node, na rysunku: C, E, F, G, H), w przeciwnym razie nazywa się **węzłem wewnętrznym** (ang. internal node).
- Ciąg węzłów połączonych krawędziami to **ścieżka** (ang. path). Od korzenia do określonego węzła w drzewie można się dostać tylko jedną drogą (ścieżką).



Struktury danych – drzewo

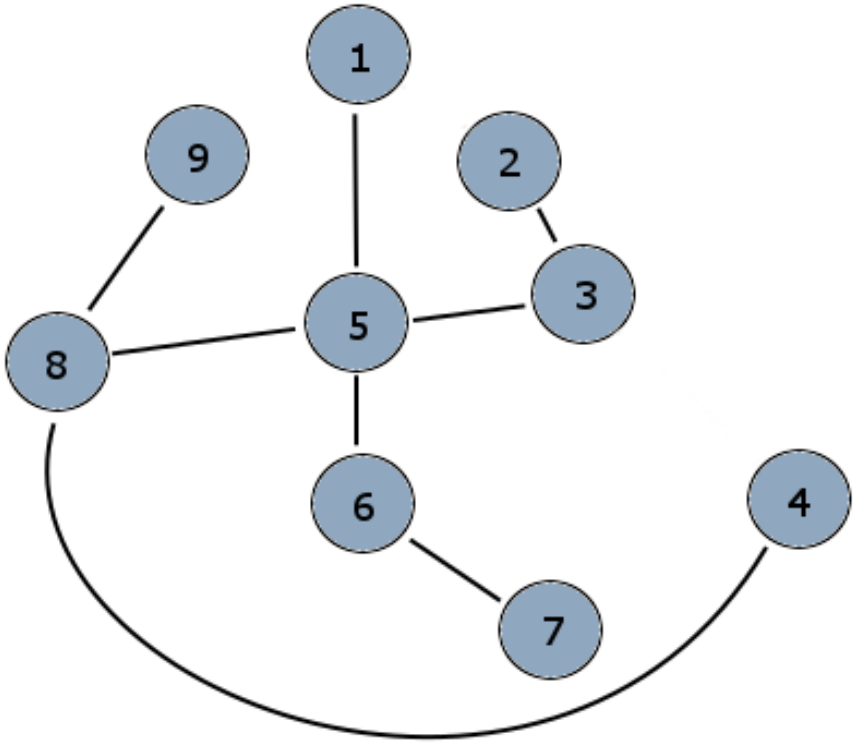


- **Drzewo binarne** to takie, w którym węzły mogą posiadać co najwyżej dwóch synów (dzieci).
- Węzły potomne nazywa się odpowiednio **dzieckiem lewym** (ang. left child node) i **dzieckiem prawym** (ang. right child node).

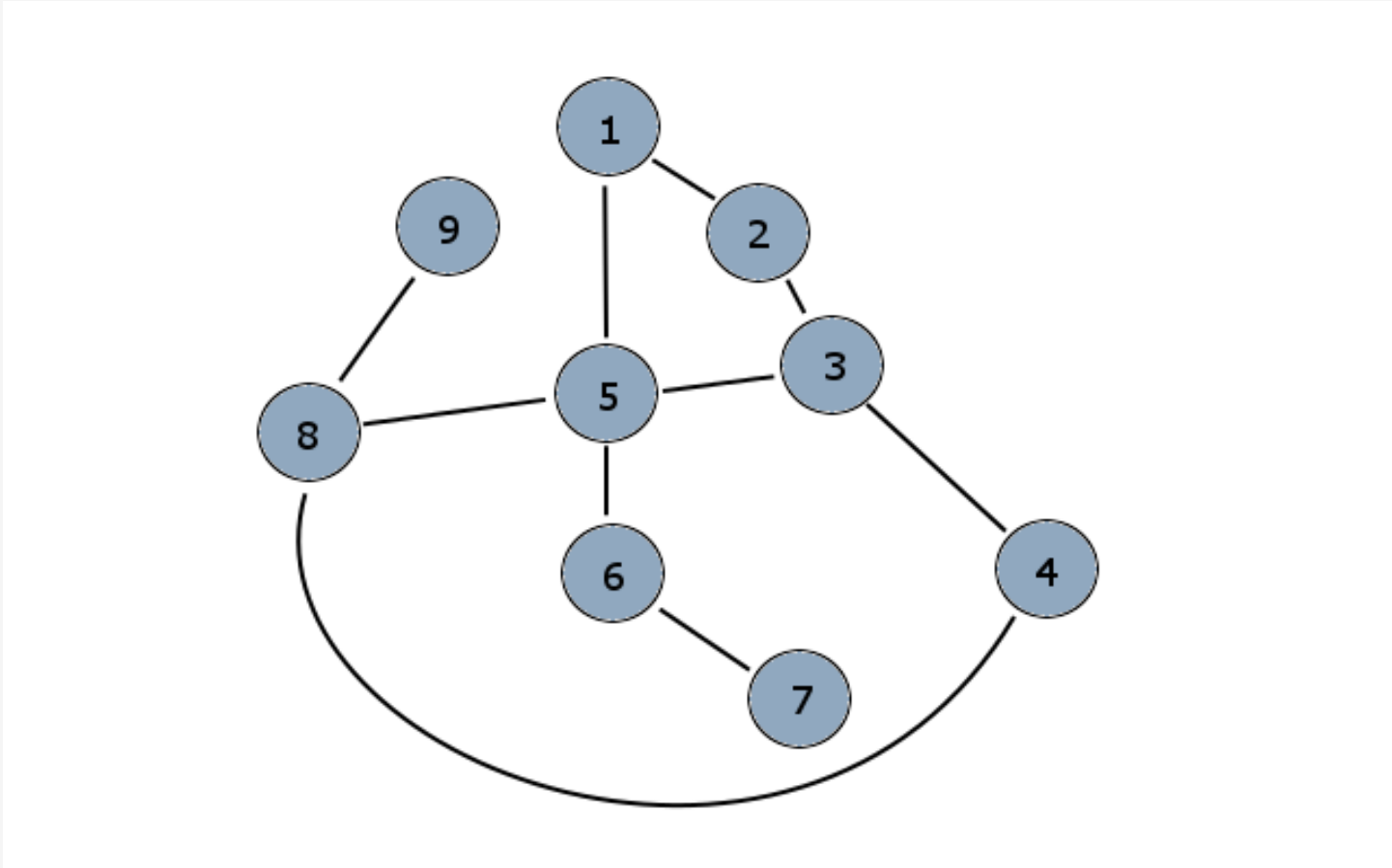
- Drzewa ułatwiają i przyspieszają wyszukiwanie, a także pozwalają w łatwy sposób operować na posortowanych danych.
- Drzewa są stosowane praktycznie w każdej dziedzinie informatyki (np. bazy danych, grafika komputerowa, przetwarzanie tekstu, telekomunikacja, serwery).

Struktury danych - graf

- Grafem nazywamy strukturę złożoną z wierzchołków i krawędzi łączących te wierzchołki.
- Drzewo to taki graf, w którym istnieje dokładnie jedna droga między dwoma wierzchołkami. W drzewie o **n** wierzchołkach jest dokładnie **n-1** krawędzi.



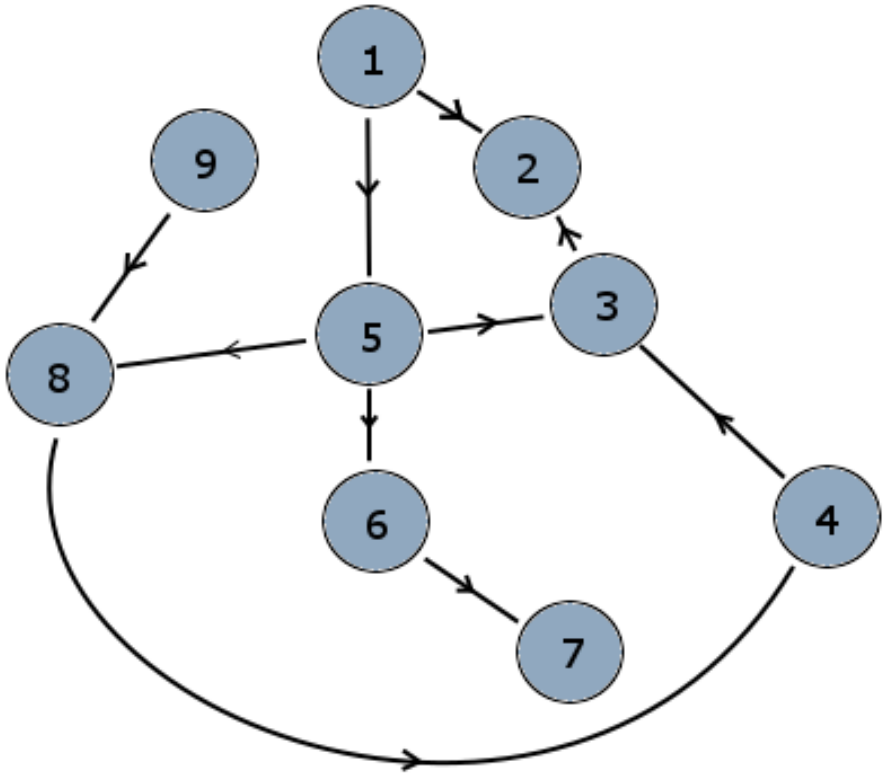
Struktury danych – graf nieskierowany



- Graf nieskierowany to taki graf, w którym połączenie między dwoma wierzchołkami **A** i **B** jest dwukierunkowe (**A <--> B**, z wierzchołka A możemy się dostać do wierzchołka B i na odwrót**).**

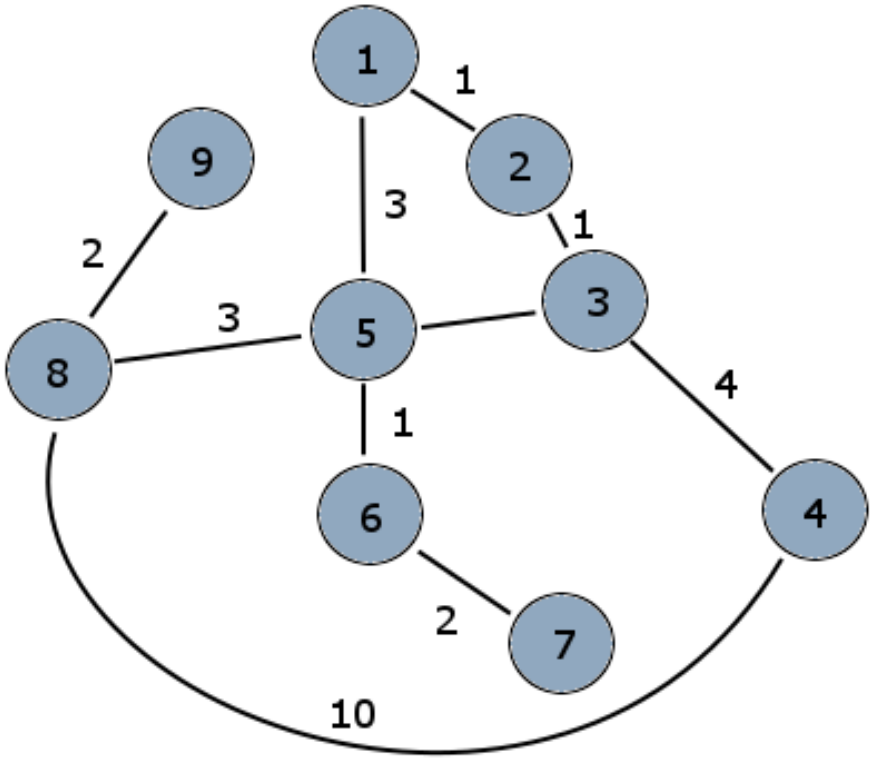
Struktury danych – graf skierowany

- W grafie skierowanym nadany jest kierunek poruszania się między dwoma wierzchołkami.
- Przejście z wierzchołka **1** do **3** jest możliwe tylko poprzez wierzchołek numer **5,** natomiast z **9** możemy tylko wyjść, ale nie ma możliwości przejścia do niego.



Struktury danych – graf wagowy

- W grafie wagowym (skierowanym lub nieskierowanym) każda krawędź ma nadaną wagę. Wierzchołki można porównać do miast, krawędzie do dróg łączących te miasta, natomiast odległości między tymi miastami to wagi.



Struktury danych - kopiec

- Ostatnią strukturą, którą omówimy jest kopiec.
- Kopiec jest drzewem binarnym o specjalnej właściwości - w korzeniu (na szczycie kopca) zawsze znajduje się najmniejsza wartość jeśli kopiec jest minimalny lub największa jeśli kopiec jest maksymalny.
- Python dostarcza implementacji minimalnego kopca w module **heapq**.
- Pomimo, że kopiec formalnie jest drzewem, w praktyce przechowuje się go w postaci listy.
- Kopiec słabo nadaje się do wyszukiwania ale świetnie sprawdza się do tego aby pobrać z jego wierzchołka najmniejszy element. Wtedy na to miejsce wskoczy następny w kolejności najmniejszy element.
- W takim razie z kopca pobieramy elementy uporządkowane rosnąco - tak właśnie działa sortowanie przez kopcowanie - z listy losowych elementów tworzymy kopiec i wyciągamy je z kopca aż będzie on pusty, dostaniemy uporządkowaną listę elementów.
- Kopiec można również traktować jak priorytetową kolejkę - w końcu zawsze wyjmiemy z niego element o ekstremalnym priorytecie.

```
In [1]: paste
from heapq import heappush, heappop

def heapsort(iterable):
    h = []
    for value in iterable:
        heappush(h, value)
    return [heappop(h) for i in range(len(h))]
## -- End pasted text --

In [2]: import random

In [3]: random_list = random.sample(range(100), 10)

In [4]: random_list
Out[4]: [15, 46, 88, 19, 69, 64, 26, 77, 78, 32]

In [5]: sorted_list = heapsort(random_list)

In [6]: sorted_list
Out[6]: [15, 19, 26, 32, 46, 64, 69, 77, 78, 88]
```

https://www.youtube.com/watch?v=t0Cq6tVNRBA

Autor: Michał Nowotka