Python

Algorytmy i struktury danych

Struktury danych (Python)

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]

Kopiowanie słownika

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

Kopiowanie słownika - copy vs deepcopy

```
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]}
```

Kopiowanie słownika - copy vs deepcopy

```
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]}
```

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.

Biblioteka collections - przypomnienie

Moduł zawierający bardziej zaawansowane, wykorzystywane do specyficznych zadań, struktury:

- namedtuple (krotki nazwane),
- Counter (słownik liczący wystąpienia poszczególnych elementów),
- OrderedDict (słownik pamiętający kolejność dodawania do nich kluczy),
- defaultdict (słownik dostarczający brakujących wartości),
- deque (lista o dwóch końcach)

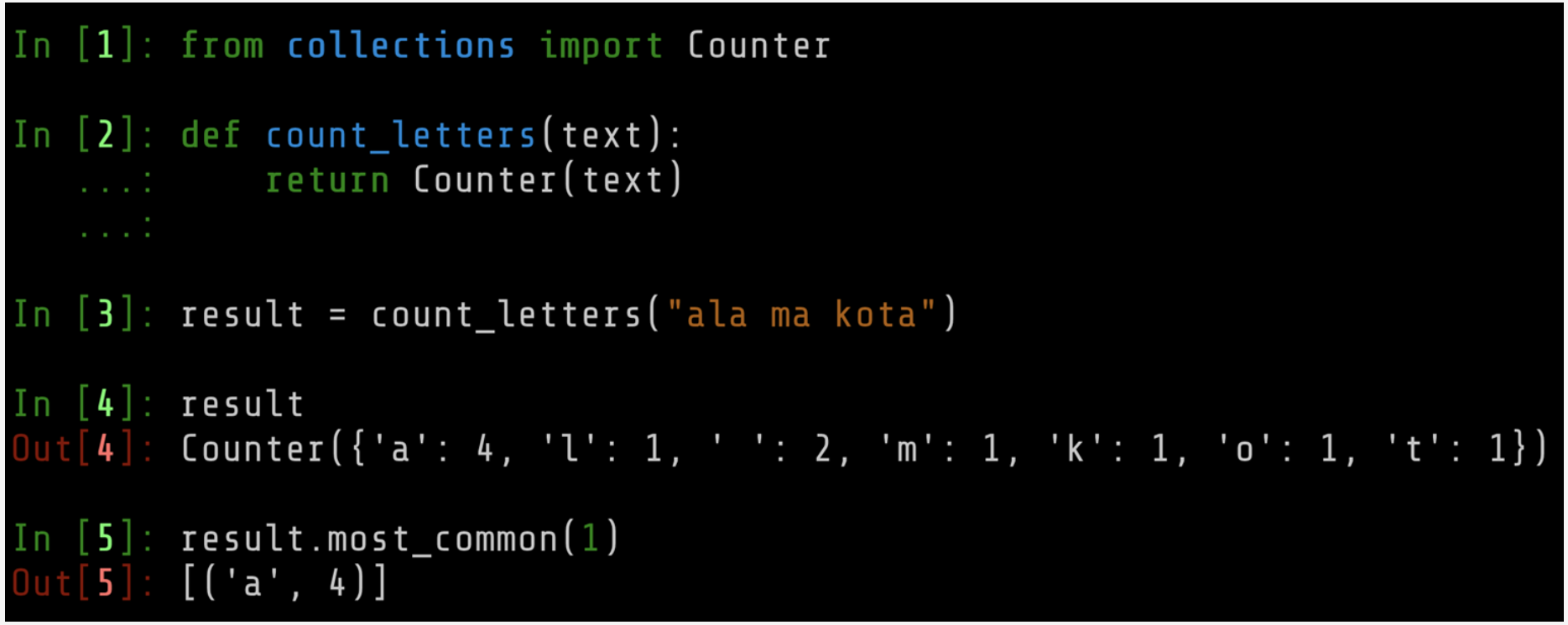
Struktury danych - namedtuple

```
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 - defaultdict

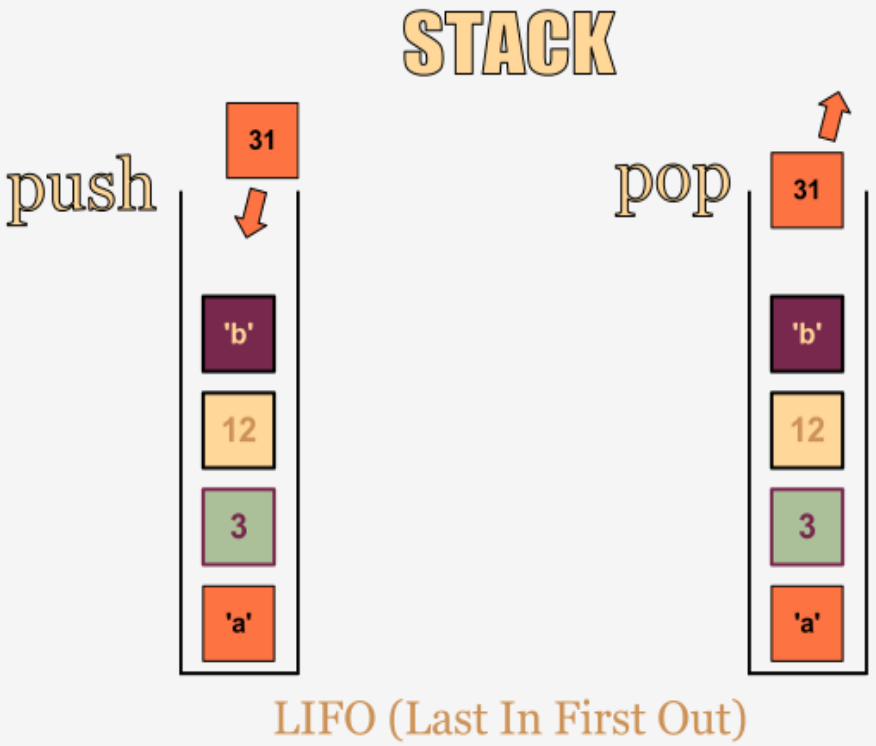
```
In [1]: from collections import defaultdict

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



Struktury danych

Struktury danych - stos




```
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]: []
```

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

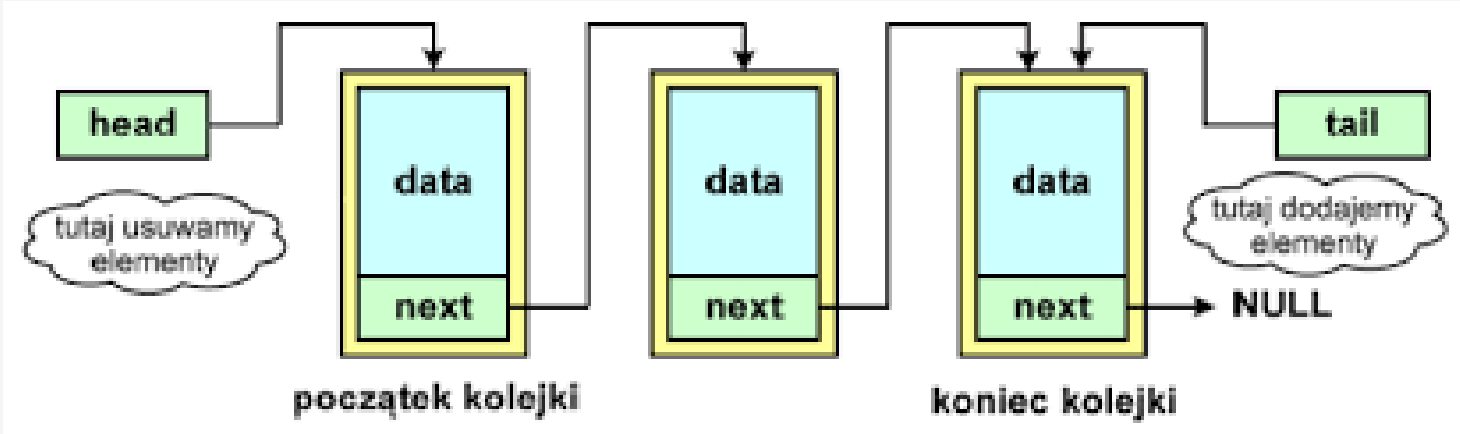

```
In [1]: from collections import deque

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([])
```

Struktury danych - kolejka

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

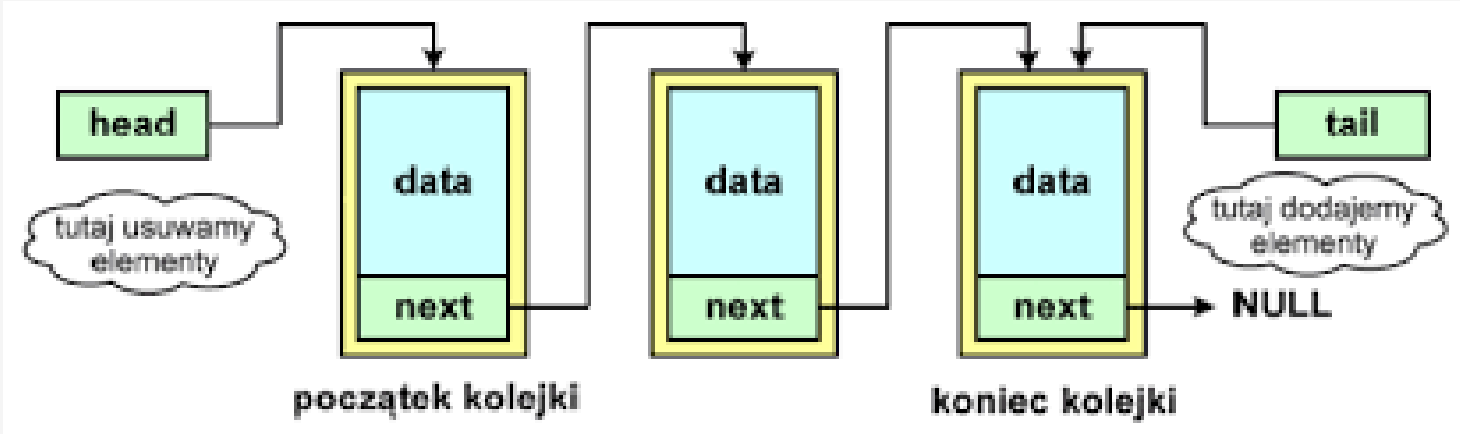
Struktury danych - lista

- **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 (lista 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 list 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).



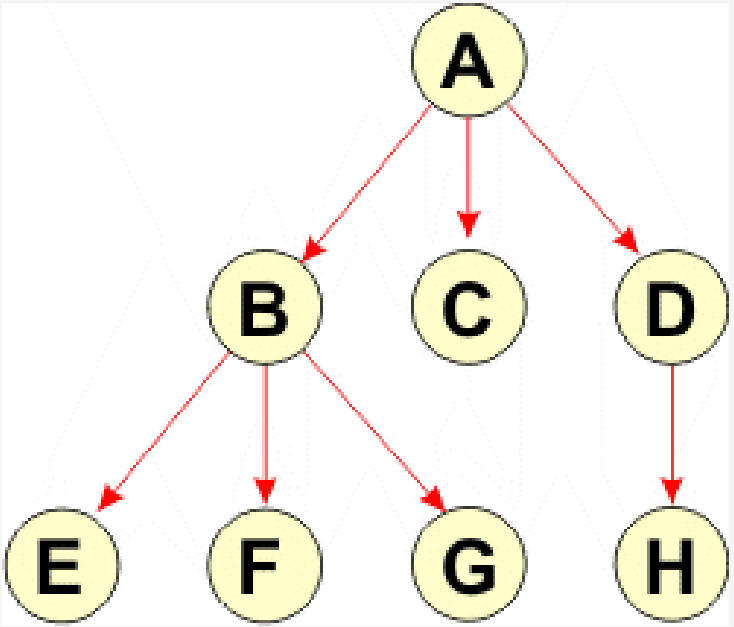
Struktury danych - lista

- 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.

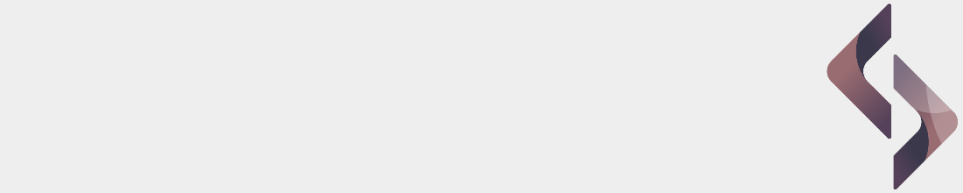


Struktury danych - drzewo

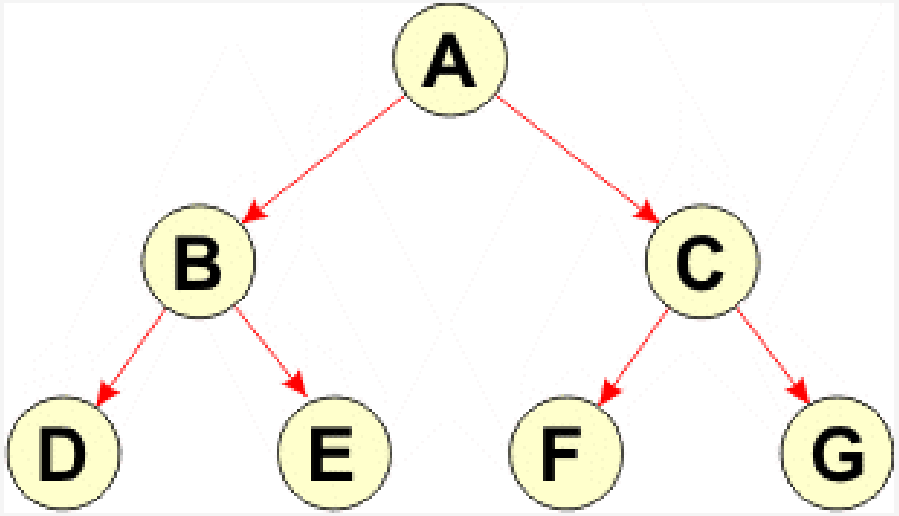
- **Drzewo** to struktura zbudowana z **węzłów** (ang. node).
- Węzły przechowują dane, są ze sobą powiązane w sposób hierarchiczny za pomocą **krawędzi** (ang. edge), reprezentowanych przez strzałki. Pierwszy węzeł drzewa nazywa się **korzeniem** (ang. root node, na rysunku: A).
- Z korzenia odchodzą 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



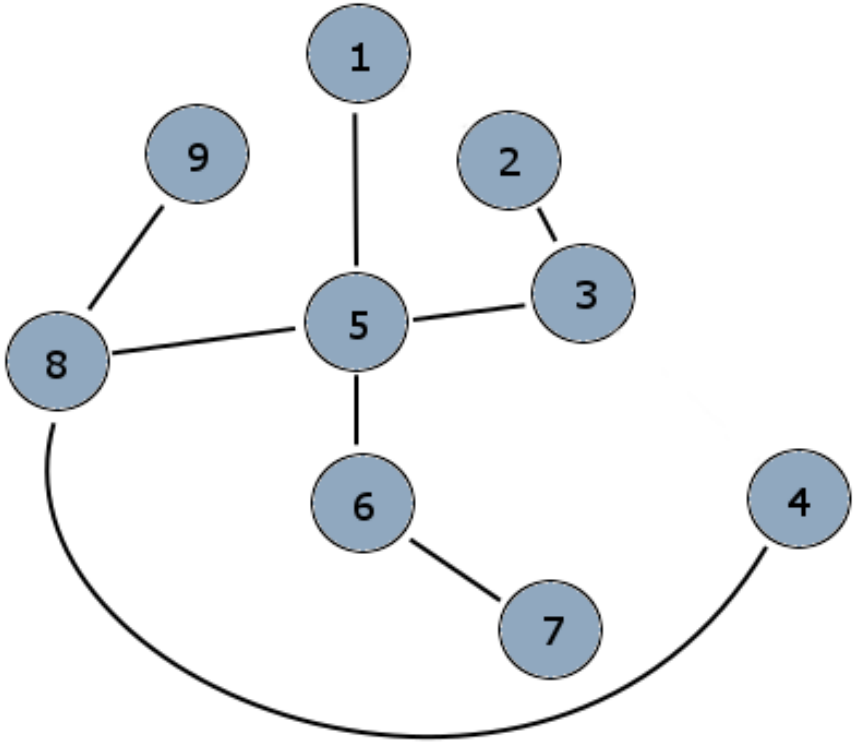
- **Drzewo binarne** to takie, w którym węzły mogą posiadać co najwyżej dwóch synów (dzieci).
- Węzły potomne nazywa się odpowiednio **dzieckiem lewym** (ang. left child node) i **dzieckiem prawym** (ang. right child node).



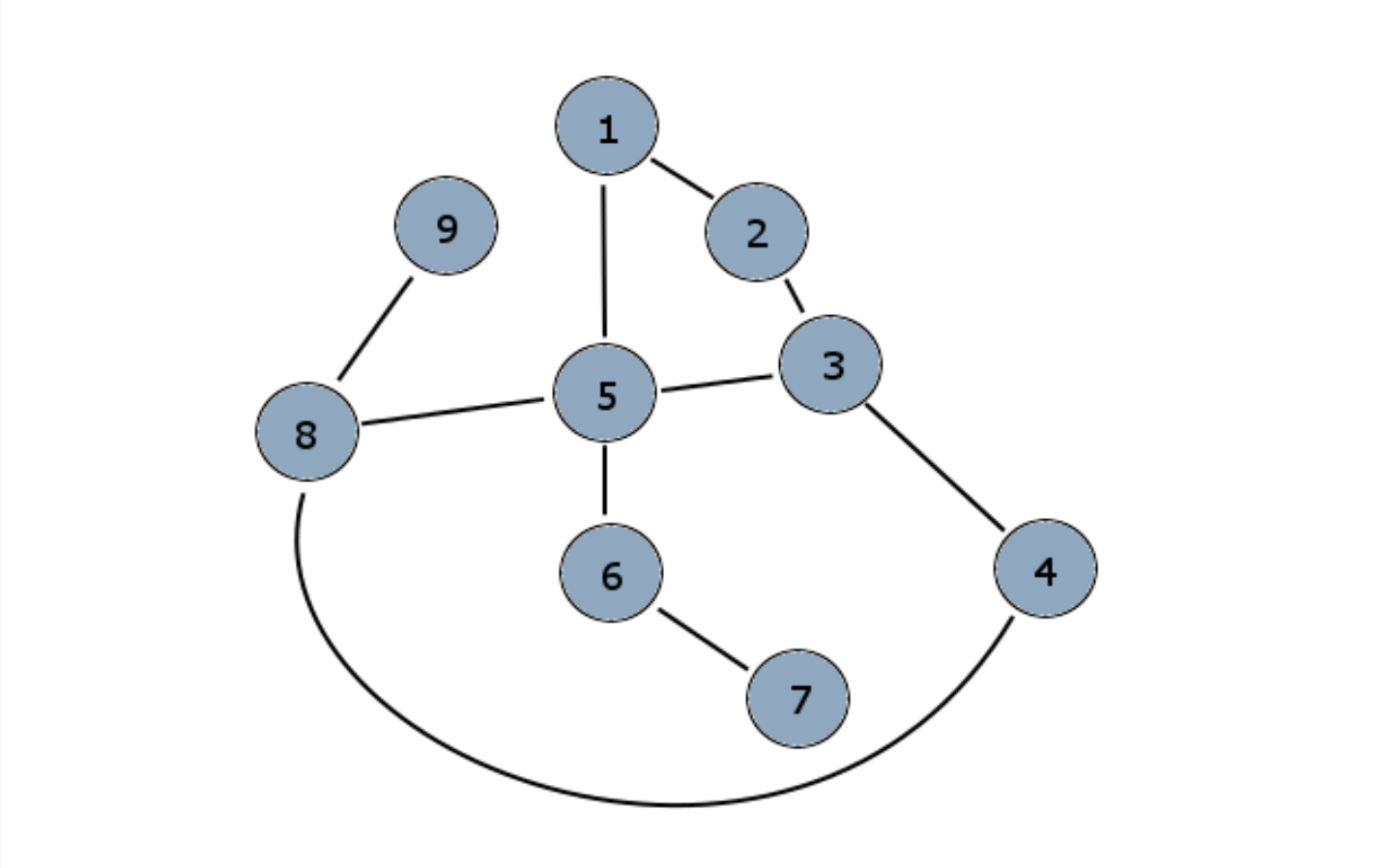
- Drzewa ułatwiają i przyspieszają wyszukiwanie, a także pozwalają w łatwy sposób operować na posortowanych danych.
- Drzewa są stosowane praktycznie w każdej dziedzinie informatyki (np. bazy danych, grafika komputerowa, przetwarzanie tekstu, telekomunikacja, serwery).

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.



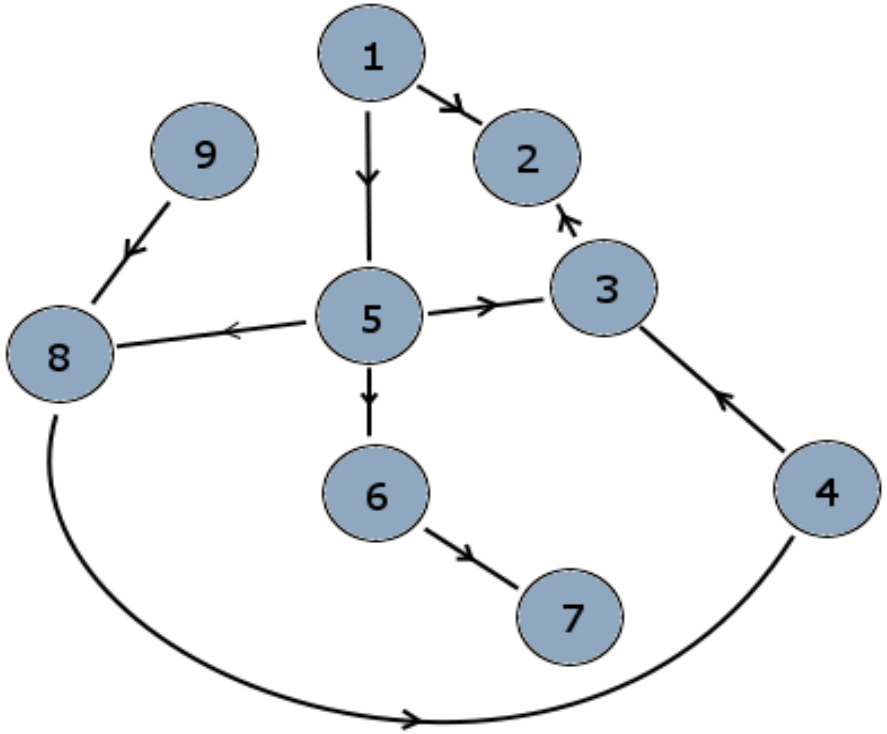
Struktury danych – graf nieskierowany



- 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**).**

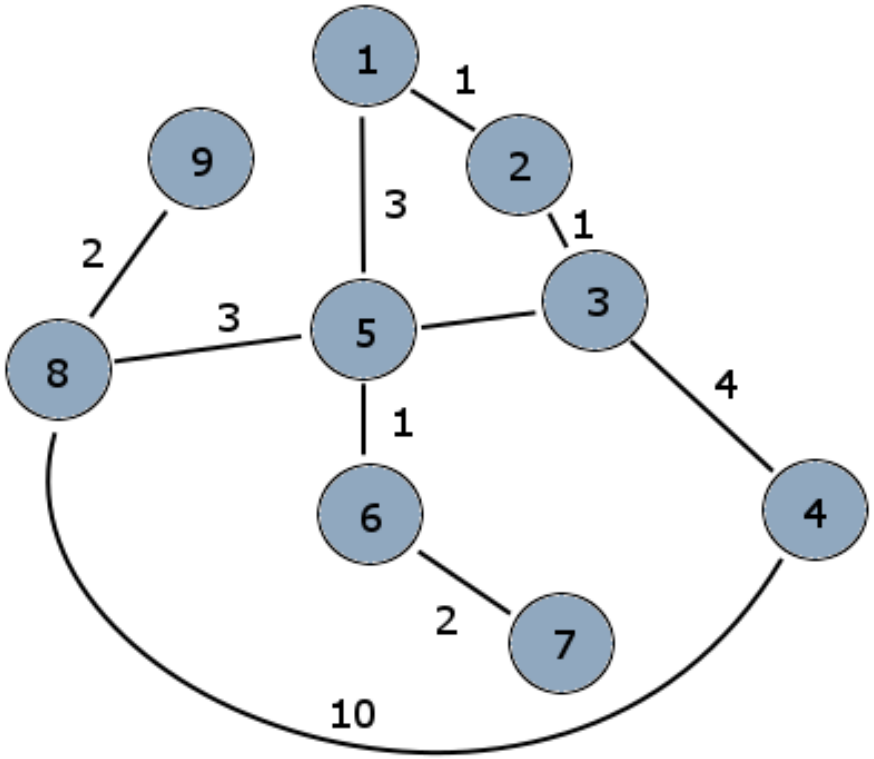
Struktury danych – graf skierowany

- W grafie skierowanym nadany jest kierunek poruszania się między dwoma wierzchołkami. **Kierunek może być zarówno jedno- jak i dwustronny (oznaczany dwoma przeciwnymi strzałkami wzdłuż jednej krawędzi).**
- 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

- 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]
```

Czasochłonne wykonywanie zadań - Celery

- Przypadek: wysyłanie polecenia wykonania długiego zadania do serwera WWW...

- Celery to program łączący się z serwerem kolejek.
- Kolejka w serwerze kolejek jest w stanie przyjąć wiadomość od użytkownika i znaleźć komputer/wolny proces (tzw. node), na którym może on być wykonany.
- W międzyczasie strona WWW może działać dalej, nie musi „zastygać”.
- Aplikacja będzie mogła sprawdzać informacje o wykonaniu zadania.
- Celery zajmuje się wszystkim: łącznie z obsługą błędów, sychronizacją i delegowaniem zadań.
- Programista dostarcza tylko funkcję do dodania do kolejki.
- Instalacja: `sudo pip install Celery`

https://docs.celeryproject.org/en/stable/getting-started/introduction.html

Czasochłonne wykonywanie zadań - Celery

tasks.py

```python
from celery.task import task
import time

@task
def add(x, y):
    time.sleep(1)
    return x + y
```

terminal

```
>>> import tasks
>>> r = tasks.add.delay(2,2)
>>> r.ready()
True
>>> r.get()
4
```