https://github.com/KacperBudnik/AiSD

In [126…

```
import time
import matplotlib.pyplot as plt
import numpy as np
import random as rng
```

Zadanie 1

In [246…

```
class QueueBaE(object):
    """
    Klasa implementująca kolejkę za pomocą pythonowej listy tak,
    że początek kolejki jest przechowywany na początku listy.
    """

    def __init__(self):
        self.list_of_items = []



    def enqueue(self, item):
        """
        Metoda służąca do dodawania obiektu do kolejki.
        Pobiera jako argument obiekt który ma być dodany.
        Niczego nie zwraca.
        """
        #self.list_of_items+=[item]
        self.list_of_items.append(item)

    def dequeue(self):
        """
        Metoda służąca do ściągania obiektu do kolejki.
        Nie pobiera argumentów.
        Zwraca ściągnięty obiekt.
        """
        return self.list_of_items.pop()

    def is_empty(self):
        """
        Metoda służąca do sprawdzania, czy kolejka jest pusta.
        Nie pobiera argumentów.
        Zwraca True jeśli kolejka jest pusta lub False gdy nie jest.
        """
        return not self.list_of_items

    def size(self):
        """
        Metoda służąca do określania wielkości kolejki.
        Nie pobiera argumentów.
        Zwraca liczbę obiektów w kolejce.
        """
        return len(self.list_of_items)


class QueueBaB(object):
    """
    Klasa implementująca kolejkę za pomocą pythonowej listy tak,
    że początek kolejki jest przechowywany na końcu listy.
    """
```

```
    def __init__(self):
        self.list_of_items = []

    def enqueue(self, item):
        """
        Metoda służąca do dodawania obiektu do kolejki.
        Pobiera jako argument obiekt który ma być dodany.
        Niczego nie zwraca.
        """
        #self.list_of_items=[item] + self.list_of_items
        self.list_of_items.insert(0,item)


    def dequeue(self):
        """
        Metoda służąca do ściągania obiektu do kolejki.
        Nie pobiera argumentów.
        Zwraca ściągnięty obiekt.
        """
        return self.list_of_items.pop(0)

    def is_empty(self):
        """
        Metoda służąca do sprawdzania, czy kolejka jest pusta.
        Nie pobiera argumentów.
        Zwraca True jeśli kolejka jest pusta lub False gdy nie jest.
        """
        return not self.list_of_items

    def size(self):
        """
        Metoda służąca do określania wielkości kolejki.
        Nie pobiera argumentów.
        Zwraca liczbę obiektów w kolejce.
        """
        return len(self.list_of_items)
```

Zadanie 2

Porównam czas dodawania i zdejmowania elementów

In [248…

```
%%timeit -n 1000
B.enqueue(12)
```

```
253 µs ± 5.79 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In [249…

```
%%timeit -n 1000
E.enqueue(12)
```

```
115 ns ± 7.7 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In [250…

```
%%timeit -n 1000
B.dequeue()
```

```
20.6 µs ± 525 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In [251…

```
%%timeit -n 1000
E.dequeue()
```

```
109 ns ± 2.05 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In [266…

```
B=QueueBaB()
x_en=[0 for _ in range(100001)]
x_en[0]=time.time()
for i in range(100000):
    B.enqueue(12)
    x_en[i+1]=time.time()-x_en[0]
```

In [267…

```
E=QueueBaE()
y_en=[0 for _ in range(100001)]
y_en[0]=time.time()
for i in range(100000):
    E.enqueue(12)
    y_en[i+1]=time.time()-y_en[0]
```

In [268…

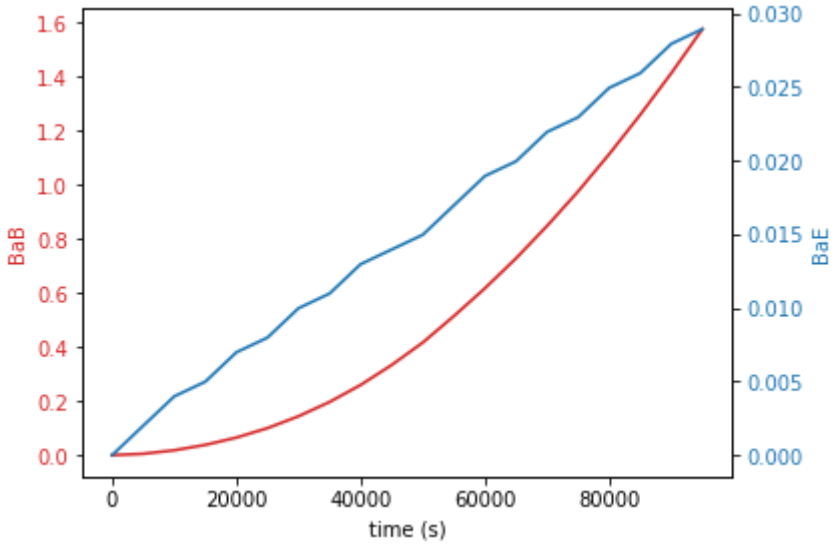
```
x_de=[0 for _ in range(100001)]
x_de[0]=time.time()
for i in range(100000):
    B.dequeue()
    x_de[i+1]=time.time()-x_de[0]
```

In [269…

```
y_de=[0 for _ in range(100001)]
y_de[0]=time.time()
for i in range(100000):
    E.dequeue()
    y_de[i+1]=time.time()-y_de[0]
```

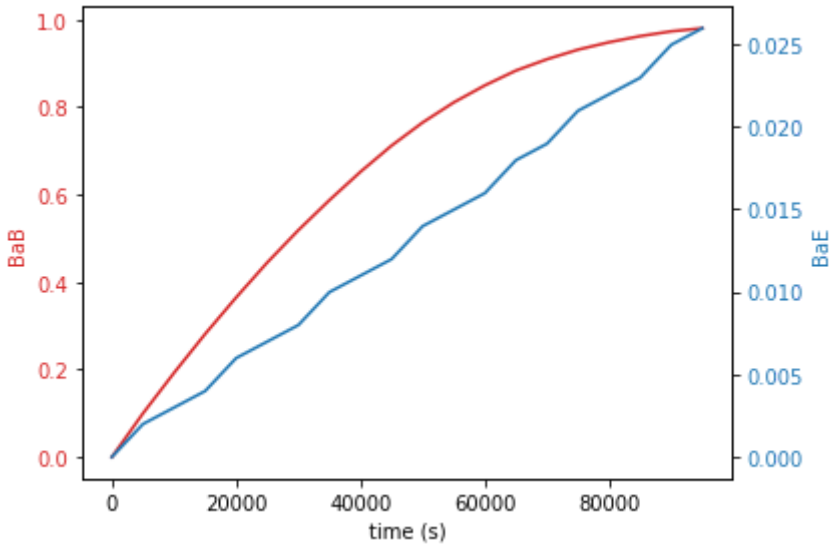
In [270…

```
t = range(100001)
fig, ax1 = plt.subplots()

color = 'tab:red'
ax1.set_xlabel('time (s)')
ax1.set_ylabel('BaB', color=color)
ax1.plot(t[1::5000], x_en[1::5000], color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis

color = 'tab:blue'
ax2.set_ylabel('BaE', color=color)  # we already handled the x-label with ax1
ax2.plot(t[1::5000], y_en[1::5000], color=color)
ax2.tick_params(axis='y', labelcolor=color)

fig.tight_layout()  # otherwise the right y-label is slightly clipped
plt.show()
```



In [272…

```
t = range(100001)
fig, ax1 = plt.subplots()

color = 'tab:red'
ax1.set_xlabel('time (s)')
ax1.set_ylabel('BaB', color=color)
ax1.plot(t[1::5000], x_de[1::5000], color=color)
ax1.tick_params(axis='y', labelcolor=color)
ax1.ticklabel_format(useOffset=False)

ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis

color = 'tab:blue'
ax2.set_ylabel('BaE', color=color)  # we already handled the x-label with ax1
ax2.plot(t[1::5000], y_de[1::5000], color=color)
ax2.tick_params(axis='y', labelcolor=color)
ax2.ticklabel_format(useOffset=False)

fig.tight_layout()  # otherwise the right y-label is slightly clipped
plt.show()
```



Zadanie 3

W pewnym urzędzie każdy wniosek musi zostać przyjęty, rozpatrzony, zeskanowany i

zarchiwizowany. Do tej pory każdy z pracowników wykonywał wszystkie czynności związane z daną osobą (przyjmował, rozpatrywał itp). Zatem cały urząd działał na podstawie koljeki (FiFo).

Dyrektor placówki postanowił coś zmienić, chciał **zwiększyć liczbę wniosków rozpatrywanych dziennie**, dlatego zebrał największe umysły jakie miał pod ręką i rozpoczęła się pierwsza rewolucja urzędowa. Po wielogodzinnej naradzie, Pani Basia, która od 30 lat działa na pierwszej lini front (okienko) wpadła na pomysł:

> "Od dzieciństwa uwielbiałam kontakt z innymi ludzmi. Od samego początku kochałam tą pracę, to że każdego dnia mogłam poznawać nowe osoby, ale niestety szybko zorientowałam się, że nie nie wszystko działa tak jak powinno. Każdego dnia widzę masę ludzi przychodzących do nas z różnymi potrzebami, z prośbą o rady, widzą w nas wybawienie, odpowiedź na ich problemy. Niestety czas potrzebny by załatwić każdy wniosek osoby muszą przez cały czas stać przy nas, nie mogą odejść nawet na chwilę. Większość nawet nie da rady przyjść do okienka, a by załatwić sprawę przychodzą z samego rana i czekają godzinami, blokują ruch w urzędzie. Jeśli zmniejszyć liczbę okienek, ale przy każdym z nich można by tylko złożyć wniosek, a rozpatrywaniem, skanowaniem i archiwizowaniem zajmowałyby się inne osoby oddalone z okienek, wyspecjalizowane w danej rzeczy,"

Pewny urząd chcąc zwiększyć liczbę rozpatrywanych wniosków dziennie postanowił coś zmienić.

Dyrektor placówki postanowił coś zmienić, chciał **zwiększyć liczbę wniosków rozpatrywanych dziennie**, dlatego zebrał największe umysły jakie miał pod ręką i rozpoczęła się pierwsza rewolucja urzędowa. Po wielogodzinnej naradzie, Pani Basia, która od 30 lat działa na pierwszej lini front (okienko) wpadła na pomysł:

Jeśli zamiast robienia wszytkiego przy okienku, to można by podzielić się obowiązkami. Każdy by robił jedną rzecz, dzięki czemu wykonywałby to szybciej, oraz zmniejszyć czas oczekiwania w kolejce na samo złożenie wniosku. Jeśli ludzie nie będą czekać godzinami, tylko chwilę by złożyć wniosek i mogli by iść do domu, zlikwidowałoby to liczbę incydentów związanych z zdenerwowanymi wielogodzinnymi postojami w kolejce, tym samym mniejszy czas potrzebny na uspokajanie przy okienku awanturników.

Po takiej przemowie Pani Basi Dyrektor z nadzieją w oczach rozkazał by wykonać jej pomysł (tylko nie wiadomo czy uważał go za dobry, czy w głowiem miał tylko soczystego steka, którego zapach już od godziny roznosił się po całym korytarzu przed salą spotkań).

Założenia

- Czas potrzebny na:
    1. Przyjęcie wniosku: 10 min
    2. Rozpatrzenie wniosku: 30 min
    3. Zeskanowanie i zarchiwiozowanie wniosku: 10 min
- Liczba pracowników: 20
- Czas pracu urzędu: 8-16 (8 godzin)
- Czas poświęcony na uspokajanie awanturników: 10 min

- Szansa na awanturnika: 5 %/20min (co każde 20min istnieje 5% szans, że osoba będzie się awanturowała)
- Przerwa pracownicza: 0 min
- Liczba wniosków danego dnia: 100-400
- Godzina przychodzienia ludzi: 8-15
- Osoby przychodzą o równych godzinach
- Schemat przychodzenia: weights=(25,15,10,10,10,10,10,10) (25% na 8, 15% na 9 itd.)
- Jeśli nie uda się ukończyć całego wniosku w jeden dzień, trzeba zacząć od nowa następnego
- Jeśli do końca zostało parę osób (nikt już nie przyjdzie to pracownicy siedzą parę minut dłużej i wszystkie wnioski odbiorą)

Zasada działania nowego urzędu

- 4 pracowników przyjmuje wnioski
- po przyjęciu odkładają je na jeden stos
- kolejnych 12 osób odbiera wnioski i je rozpatruje (biorą pierwsze z góry LiFo)
- rozpatrzone wnioski układają na skanerze, który automatycznie je skanuje w kolejności ich otrzymywania (FiFo)
- po zeskanowaniu pozostałe (4) osoby archiwizują je (w kolejności skanowania)

In [162…

```
def f(T):
    return 1/(5*(T-6)^2)+0.05

def test():
    n = rng.randint(100,400) # Ile maksymalnie może przyjść osób

    def old():
        count_angry = 0 # ilu było awanturników
        hooman = rng.choices([8,9,10,11,12,13,14,15], weights=(25,15,10,10,10,10,10,
        angry = [0]*n # czy będzie się awanturował 0-nie 1-tak
        hooman.sort()
        worker = [0]*20 # ile jeszcze będą zajęci 1 - 1 tick
        done = 0 # ile dziś zostało zakończonych wniosków
        #print(hooman)
        # 1 tick = 10 min   (1h = 6*10 min)
        for tick in range(8*6):
            if tick % 2 == 0:
                for i in range(len(hooman)):
                    if (hooman[i]-7)*6 - tick <= 0 :
                        if rng.random() < 0.05:
                            angry[i]=1
            for i in range(20):
                if worker[i]>0:
                    worker[i]-=1
                    if worker[i]==0:
                        done+=1
                else:
                    if hooman[0]*6<=8*6+tick:
                        is_angry = angry.pop(0)
                        if is_angry == 1:
                            count_angry+=1
                        hooman.pop(0)
                        if not hooman:
                            return dict(done=done,angry=count_angry)
                        worker[i] = 5 if is_angry==0 else 6
```

```
        return dict(done=done,angry=count_angry)


    def new():
        count_angry = 0 # ilu było awanturników
        hooman = rng.choices([8,9,10,11,12,13,14,15], weights=(25,15,10,10,10,10,10,
        angry = [0]*n # czy będzie się awanturował 0-nie 1-tak
        hooman.sort()
        worker = [0]*20 # ile jeszcze będą zajęci 1 - 1 tick
        done = 0 # ile dziś zostało zakończonych wniosków
        accepted =[] # przyjęte do rozpatrzenia
        to_scan = [] # do zeskanowania
        worker_eff=[0]*20


        # 1 tick = 10 min   (1h = 6*10 min)
        for tick in range(8*6):
            if tick % 2 == 0:
                for i in range(len(hooman)):
                    if (hooman[i]-7)*6 - tick <= 0 :
                        if rng.random() < 0.05:
                            angry[i]=1

            for i in range(4):
                if worker[i]>0:
                    worker[i]-=1
                    if worker[i]==0:
                        accepted.append(i)
                else:
                    if hooman[0]*6<=8*6+tick:
                        is_angry = angry.pop(0)
                        if is_angry == 1:
                            count_angry+=1
                        hooman.pop(0)
                        if not hooman:
                            return dict(done=done,angry=count_angry)
                        worker[i] = 1 if is_angry==0 else 2

            for i in range(4,4+12):
                if worker[i]>0:
                    worker[i]-=1
                    if worker[i]==0:
                        to_scan.append(i)
                else:
                    if accepted:
                        worker_eff[accepted.pop()]+=1
                        worker[i]+=3

            for i in range(12+4,12+4+4):
                if to_scan:
                    worker_eff[to_scan.pop(0)]+=1
                    done+=1 #zawsze robi się to 1 tick, więc do następnego wszystko
                    worker_eff[i]+=1

        return dict(done=done,angry=count_angry)



    return dict(old=old(),new=new())
```

In [164…

```
test()
```

Out[164… {'old': {'done': 143, 'angry': 31}, 'new': {'done': 78, 'angry': 26}}

In [165…
```
old_win=0
old_angry=0
new_win=0
new_angry=0
draw=0

for i in range(1000):
    result=test()
    if result["old"]["done"]>result["new"]["done"]:
        old_win+=1
    elif result["old"]["done"]<result["new"]["done"]:
        new_win+=1
    else:
        draw+=1

    if result["old"]["angry"]>result["new"]["angry"]:
        old_angry+=1
    elif result["old"]["angry"]<result["new"]["angry"]:
        new_angry+=1
```

In [165…
```
print("old/angry:",old_win,"/",old_angry,"\nnew/angry:",new_win,"/",new_angry,"\ndra
```

```
old/angry: 999 / 614
new/angry: 0 / 374
draw: 1
```

Możemu tu zobaczyć, że Pani Basia zdecydowanie nie miała racji

W nowym systemi osoby czekają znacznie mniej w kolejce, ale jest mniej effectywny. Jeśli zwrócimy uwagę na to, że to była gwałtowna rewolucja. Osoby nadal przychodzą najczęściej o 8 rano i czekają długo w kolejce. Po dłuższym czasie nasz trwabnia eksperymentu, ludzie nauczyli by się, że można przychodzić o każdej porze, co jeszcze zminejszyo by liczbę niezadowolonych.

Przewaga w ilości przerobionych wniosków wynika również z uproszczenia. Kolejnego dnia musimy zaczynać wszystko od nowa, więc przez pierwsze 10 min pracuje jedynie 20% pracowników, przez kolejnie 30 min tylko 80%, oraz nie zwracamy uwagi na wyspecjalizowanie pracowników (robiąc tylko jedną rzecz robimy ją lepiej)

Zadanie 4

In [1]:
```
from html.parser import HTMLParser
```

In [363…
```
class MyHTMLParser(HTMLParser):
    def __init__(self,**args):
        self.tagu=[]
        super().__init__(**args)

    def handle_starttag(self, tag, attrs):
        self.tagu.append(tag)

    def handle_endtag(self, tag):
        self.tagu.append('/'+tag)

    def handle_data(self, data):
        pass
```

```
        #print("Encountered some data  :", data)

def checking_HTML_correctness(filename):
    """
    Funkcja ma za zadanie sprawdzać poprawność składni dokumentu HTML.
    Jako argument przyjmuje nazwę pliku, który ma sprawdzić.
    Zwraca True jeśli dokument jest poprawny składniowo i False jeśli nie jest.
    """
    file_obj = open(filename, 'r')
    text = file_obj.read()
    parser = MyHTMLParser()
    parser.feed(text)
    g=[]
    to_pass=["area","base","col","command","embed","hr","img","input","keygen","link
    for i in parser.tagu:
        if i[0]!="/":
            g.append(i)
        else:
            d=g.pop()

            if not d in to_pass and i[1:]!=d:
                print(i,d)
                return False
    return True
```

In [364…

```
checking_HTML_correctness(r"D:\GitHub\AiSD\List 4\L4_ZAD4_sampleHTML_3.txt")
```

Out[364… True

Zadanie 5

In [3]:

```
import numpy as np

class Node:

    def __repr__(self):

        return str(self.data)

    def __init__(self,init_data):
        self.data = init_data
        self.next = None

    def get_data(self):
        return self.data

    def get_next(self):
        return self.next

    def set_data(self,new_data):
        self.data = new_data

    def set_next(self,new_next):
        self.next = new_next


class UnorderedList(object):

    def __init__(self):
        self.head = None
```

```
        self.n=0

    def is_empty(self):
        return self.head == None

    def add(self, item):
        temp = Node(item)
        temp.set_next(self.head)
        self.head = temp

    def size(self):
        current = self.head
        count = 0
        while current != None:
            count = count + 1
            current = current.get_next()
        return count

    def search(self,item):
        current = self.head
        found = False
        while current != None and not found:
            if current.get_data() == item:
                found = True
            else:
                current = current.get_next()
        return found

    def remove(self, item):
        current = self.head
        previous = None
        found = False

        while not found:
            if current.get_data() == item:
                found = True
            else:
                previous = current
                current = current.get_next()

        if previous == None: #jeśli usuwamy pierwszy element
            self.head = current.get_next()
        else:
            previous.set_next(current.get_next())

##############################################################################
#                                                Uzupełnienie
##############################################################################

    def append(self, item):
        """
        Metoda dodająca element na koniec listy.
        Przyjmuje jako argument obiekt, który ma zostać dodany.
        Niczego nie zwraca.
        """
        if self.head:
            current=self.head
            while current:
                previous=current
                current=current.get_next()
            previous.set_next(Node(item))
        else:
            self.add(item)
```

```
    def index(self, item):
        """
        Metoda podaje miejsce na liście,
        na którym znajduje się określony element -
        element pod self.head ma indeks 0.
        Przyjmuje jako argument element,
        którego pozycja ma zostać określona.
        Zwraca pozycję elementu na liście lub None w przypadku,
        gdy wskazanego elementu na liście nie ma.
        """
        current = self.head
        found = False
        i=0
        while current != None and not found:
            if current.get_data() == item:
                found = True
            else:
                current = current.get_next()
                i+=1
        return i if found else None


    def insert(self, pos, item):
        """
        Metoda umieszcza na wskazanej pozycji zadany element.
        Przyjmuje jako argumenty pozycję,
        na której ma umiescić element oraz ten element.
        Niczego nie zwraca.
        Rzuca wyjątkiem IndexError w przypadku,
        gdy nie jest możliwe umieszczenie elementu
        na zadanej pozycji (np. na 5. miejsce w 3-elementowej liście).
        """
        current = self.head
        previous=None
        while current != None and pos >0:
            pos-=1
            previous=current
            current=current.get_next()

        if pos == 0:
            if previous:
                if current:
                    previous.set_next(Node(item))
                    previous.get_next().set_next(current)
                else:
                    previous.set_next(Node(item))
            else:
                self.add(item)
        else:
            raise IndexError("Wyjście poza zakres")


    def pop(self, pos=-1):
        """
        Metoda usuwa z listy element na zadaniej pozycji.
        Przyjmuje jako opcjonalny argument pozycję,
        z której ma zostać usunięty element.
        Jeśli pozycja nie zostanie podana,
        metoda usuwa (odłącza) ostatni element z listy.
        Zwraca wartość usuniętego elementu.
        Rzuca wyjątkiem IndexError w przypadku,
        gdy usunięcie elementu z danej pozycji jest niemożliwe.
        """
```

```
        n=self.size()
        if pos + 1 > n or -pos > n:
            raise IndexError("Wyjście poza zekres")

        if pos > 0:
            i = pos
        elif pos == 0:
            temp=self.head
            self.head=self.head.get_next()
            return temp.get_data()
        else:
            i = n + pos

        current = self.head
        for _ in range(i-1):
            current = current.get_next()
        temp=current.get_next()
        current.set_next(current.get_next().get_next())
        return temp.get_data()


    ############################################################################################
    #                                                  Dodatkowe
    ############################################################################################

    def __repr__(self):
        current=self.head
        tab=[]
        while current:
            tab.append(current.get_data())
            current=current.get_next()
        return str(tab)

    def reverse(self):
        """
        Metoda odwraca kolejność elementów w liście
        """
        temp=[]
        current=self.head
        while current:
            temp.append(current)
            current=current.get_next()

        current=temp.pop()
        current.set_next(temp.pop())
        self.head=current


        while temp:
            current=current.get_next()
            current.set_next(temp.pop())
        current. get_next().set_next(None)

    def count(self, item):
        """
        Metoda zwraca liczbę danego elementu z listy
        """
        i=0
        current=self.head
        while current:
            i += 1 if current.get_data()==item else 0
            current = current.get_next()
        return i
```

```
    def shuffle(self):
        """
        Metoda permutuje listę
        """
        temp=[]
        current=self.head
        while current:
            temp.append(current)
            current=current.get_next()

        temp = [i for i in np.random.permutation(temp)]

        current=temp.pop()
        current.set_next(temp.pop())
        self.head=current

        while temp:
            current=current.get_next()
            current.set_next(temp.pop())
        current.get_next().set_next(None)

    def __eq__(self, other):

        if type(other) == UnorderedList:
            current_self = self.head
            current_other = other.head

            while current_self and current_other:
                if current_self.get_data() != current_other.get_data():
                    return False
                current_self = current_self.get_next()
                current_other = current_other.get_next()
            if current_self or current_other:
                return False
            return True
        raise TypeError("'==' not supported between instances of 'UnorderedList' and


    def same_element(self,other):
        """
        Sprawdza, czy kolejki mają takie same elementy (i ich liczbę)
        """
        temp_self = []
        temp_other = []
        current=self.head
        while current:
            temp_self.append(current.get_data())
            current = current.get_next()

        current=other.head
        while current:
            temp_other.append(current.get_data())
            current = current.get_next()

        while temp_self and temp_other:
            temp = temp_self.pop()
            for i in range(len(temp_other)):
                if temp==temp_other[i]:
                    temp_other.pop(i)
                    break
            else:
                return False
        return False if temp_self or temp_other else True
```

```
    def same_type_of_elements(self, other):
        """
        Sprawdza, czy wszystkie elementy jednego są w drugim
        """
        temp_self = []
        temp_other = []
        current=self.head
        while current:
            temp_self.append(current.get_data())
            current = current.get_next()

        current=other.head
        while current:
            temp_other.append(current.get_data())
            current = current.get_next()

        return all(i in temp_self for i in temp_other) and all(i in temp_other for i

    def __getitem__(self, pos):
        n=self.size()

        if pos + 1 > n or -pos > n:
            raise IndexError("Wyjście poza zekres")

        if pos > 0:
            i = pos
        elif pos == 0:
            return self.head.get_data()
        else:
            pos = n + pos

        current=self.head
        while current and pos > 0:
            current = current.get_next()
            pos-=1

        return current.get_data()

    def __contein__(self, item):
        return self.search(item)

    def __len__(self):
        return self.size()
```

In [4]:
```
A=UnorderedList()
```

In [5]:
```
A.add(1)
```

In [6]:
```
A
```

Out[6]: [1]

In [7]:
```
1 in A
```

Out[7]: True

In [8]:
```
A[7]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-8-7423ee27b9de> in <module>
----> 1 A[7]

<ipython-input-3-e1eed524489d> in __getitem__(self, pos)
    306
    307         if pos + 1 > n or -pos > n:
--> 308             raise IndexError("Wyjście poza zekres")
    309
    310         if pos > 0:

IndexError: Wyjście poza zekres
```

In [9]:
```
len(A)
```

Out[9]: 1

In [10]:
```
A.append(232)
A.append(123)
A.add(3)
A.add(2)
A.add(1)
A.add(0)
```

In [11]:
```
A
```

Out[11]: [0, 1, 2, 3, 1, 232, 123]

In [12]:
```
A.reverse()
```

In [13]:
```
A
```

Out[13]: [123, 232, 1, 3, 2, 1, 0]

In [15]:
```
A.count(1)
```

Out[15]: 2

In [14]:
```
A.count(1234)
```

Out[14]: 0

In [16]:
```
A.insert(0,4)
```

In [17]:
```
A.index(5)
```

In [18]:
```
A
```

Out[18]: [4, 123, 232, 1, 3, 2, 1, 0]

In [19]:
```
A.index(1)
```

Out[19]: 3

In [20]:
```
A.pop(-1)
```

Out[20]: 0

In [21]:
```
A=UnorderedList()
A.append(232)
A.append(123)
A.add(3)
A.add(2)
A.add(1)
A.add(0)
```

In [22]:
```
B=UnorderedList()
B.append(232)
B.append(123)
B.add(3)
B.add(2)
B.add(1)
B.add(1)
B.add(1)
B.add(1)
B.add(1)
B.add(1)
B.add(0)
```

In [23]:
```
A.same_element(B)
```

Out[23]: False

In [24]:
```
A.same_type_of_elements(B)
```

Out[24]: True

In [25]:
```
A
```

Out[25]: [0, 1, 2, 3, 232, 123]

In [26]:
```
B
```

Out[26]: [0, 1, 1, 1, 1, 1, 1, 2, 3, 232, 123]

Zadanie 6

In [879…
```
class StackUsingUL(object):
    def __init__(self):
        self.items = UnorderedList()
```

```
    def is_empty(self):
        """
        Metoda sprawdzajacą, czy stos jest pusty.
        Nie pobiera argumentów.
        Zwraca True lub False.
        """
        return self.item.is_empty()

    def push(self, item):
        """
        Metoda umieszcza nowy element na stosie.
        Pobiera element, który ma zostać umieszczony.
        Niczego nie zwraca.
        """
        self.item.append(item)

    def pop(self):
        """
        Metoda ściąga element ze stosu.
        Nie przyjmuje żadnych argumentów.
        Zwraca ściągnięty element.
        Jeśli stos jest pusty, rzuca wyjątkiem IndexError.
        """

        return self.item.pop()

    def peek(self):
        """
        Metoda podaje wartość elementu na wierzchu stosu
        nie ściągajac go.
        Nie pobiera argumentów.
        Zwraca wierzchni element stosu.
        Jeśli stos jest pusty, rzuca wyjątkiem IndexError.
        """
        return self.item[-1]

    def size(self):
        """
        Metoda zwraca liczę elementów na stosie.
        Nie pobiera argumentów.
        Zwraca liczbę elementów na stosie.
        """
        return self.item.size()
```

Zadanie 7

In [102…

```
class DequeueUsingUL(object):

    def __init__(self):
        self.items = UnorderedList()

    def is_empty(self):
        """
        Metoda sprawdzajacą, czy kolejka jest pusta.
        Nie pobiera argumentów.
        Zwraca True lub False.
        """
        return self.items.is_empty()
```

```
    def add_left(self, item):
        """
        Metoda dodaje element do kolejki z lewej strony.
        Pobiera jako argument element, który ma zostać dodany.
        Niczego nie zwraca.
        """
        self.items.add(item)

    def add_right(self, item):
        """
        Metoda dodaje element do kolejki z prawej strony.
        Pobiera jako argument element, który ma zostać dodany.
        Niczego nie zwraca.
        """
        self.items.append(item)

    def remove_left(self):
        """
        Metoda usuwa element z kolejki z lewej strony.
        Nie pobiera argumentów.
        Zwraca usuwany element.
        W przypadku pustej kolejku rzuca wyjątkiem IndexError
        """
        return self.items.pop(0)

    def remove_right(self):
        """
        Metoda usuwa element z kolejki z prawej strony.
        Nie pobiera argumentów.
        Zwraca usuwany element.
        W przypadku pustej kolejku rzuca wyjątkiem IndexError
        """
        return self.items.pop()

    def size(self):
        """
        Metoda zwraca liczę elementów na w kolejce.
        Nie pobiera argumentów.
        Zwraca liczbę elementów na w kolejce.
        """
        return self.items.size()

    def __repr__(self):
        return self.items.__repr__()
```

In [102…
```
D=DequeueUsingUL()
```

In [103…
```
D.add_right(2)
```

In [103…
```
D
```

Out[103… [2]

In [103…
```
D.add_left(123)
```

In [103…
```
D
```

Out[103… [123, 2]

In [103…
```
D.add_right(1332)
```

In [103…
```
D
```

Out[103… [123, 2, 1332]

In [103…
```
D.size()
```

Out[103… 3

In [103…
```
D.remove_left()
```

Out[103… 123

In [103…
```
D
```

Out[103… [2, 1332]

In [103…
```
D.remove_right()
```

Out[103… 1332

In [104…
```
D
```

Out[104… [2]

In [104…
```
D.is_empty()
```

Out[104… False

In [104…
```
D.remove_left()
D.remove_left()
D.remove_left()
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-1042-0af366a78c84> in <module>
      1 D.remove_left()
----> 2 D.remove_left()
      3 D.remove_left()

<ipython-input-1028-47a4c3c3205c> in remove_left(self)
     35         W przypadku pustej kolejku rzuca wyjątkiem IndexError
     36         """
---> 37         return self.items.pop(0)
     38
     39     def remove_right(self):

<ipython-input-989-348e97abd0ba> in pop(self, pos)
```

```
    158         n=self.size()
    159         if pos + 1 > n or -pos > n:
--> 160             raise IndexError("Wyjście poza zekres")
    161 
    162         if pos > 0:

IndexError: Wyjście poza zekres
```

In [104…
```
D
```

Out[104… `[]`

In [104…
```
D.is_empty()
```

Out[104… `True`

In [104…
```
D.remove_left()
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-1045-928a7b0bea6c> in <module>
----> 1 D.remove_left()

<ipython-input-1028-47a4c3c3205c> in remove_left(self)
     35         W przypadku pustej kolejku rzuca wyjątkiem IndexError
     36         """
---> 37         return self.items.pop(0)
     38 
     39     def remove_right(self):

<ipython-input-989-348e97abd0ba> in pop(self, pos)
    158         n=self.size()
    159         if pos + 1 > n or -pos > n:
--> 160             raise IndexError("Wyjście poza zekres")
    161 
    162         if pos > 0:

IndexError: Wyjście poza zekres
```

In [104…
```
D.remove_right()
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-1046-a36f57180947> in <module>
----> 1 D.remove_right()

<ipython-input-1028-47a4c3c3205c> in remove_right(self)
     44         W przypadku pustej kolejku rzuca wyjątkiem IndexError
     45         """
---> 46         return self.items.pop()
     47 
     48     def size(self):

<ipython-input-989-348e97abd0ba> in pop(self, pos)
    158         n=self.size()
    159         if pos + 1 > n or -pos > n:
--> 160             raise IndexError("Wyjście poza zekres")
    161 
    162         if pos > 0:

IndexError: Wyjście poza zekres
```

Zadanie 8

Przeprowadzanie eksperymentu zaczniemy od różniczy w dodawaniu elementów na końcu listy.

In [107…
```
A=UnorderedList()
B=[]
```

In [107…
```
%%timeit -n 1000
A.append(1)
```

```
The slowest run took 14.52 times longer than the fastest. This could mean that an in
termediate result is being cached.
330 µs ± 191 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In [107…
```
%%timeit -n 1000
B.append(1)
```

```
41.4 ns ± 5.92 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Jeśli do pustej listy dodajemy 1000 elementów widzimy różnicę całego rzędu w czasie wykonania, ale możemy również zauważyć, że w przypadku naszej listy czas wykonani najszybszego i najwolniejszego dodania różni się ponad 14-sto krotnie, jeśli dodamy kolejne 1000 elementów:

In [107…
```
%%timeit -n 1000
A.append(1)
```

```
1.02 ms ± 166 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In [107…
```
%%timeit -n 1000
B.append(1)
```

```
37.6 ns ± 3.72 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

widzimy, że w naszej liście czas działania zależy od ilości dodanych wcześniej elementów.

Jeśli teraz sprawdzimy dodawanie na początku listy

In [108…
```
A=UnorderedList()
B=[]
```

In [108…
```
%%timeit -n 1000
A.add(1)
```

```
384 ns ± 9.81 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In [108…
```
%%timeit -n 1000
B.insert(0,1)
```

```
The slowest run took 7.39 times longer than the fastest. This could mean that an int
ermediate result is being cached.
1.08 µs ± 590 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

możemy zauważyć sytuację odwrotną, jeśli powtórzymy to dla kolejnych 1000 elementów

In [108…
```
%%timeit -n 1000
```

```
A.add(1)
```

```
475 ns ± 103 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

In [108…

```
%%timeit -n 1000
B.insert(0,1)
```

```
5.2 µs ± 640 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Zauważymy, że tym razem listy pythonowskie działają wolniej i zależą od ilości elementów

In [110…

```
A_pocz=UnorderedList() # nasza dodawanie na początku
A_kon=UnorderedList()  # Dodawanie na koncu
B_pocz=[] # Pythonowska dodawanie na początku
B_kon=[]  # na koncu


# tavela czasow
pyt_pocz=[0 for i in range(30000)]
pyt_kon=[0 for i in range(30000)]
nasz_pocz=[0 for i in range(30000)]
nasz_kon=[0 for i in range(30000)]


#liczenie czasow dodawania
tim=time.time()
for i in range(30000):
    B_pocz.insert(0,i)
    pyt_pocz[i]=time.time()-tim

tim=time.time()
for i in range(30000):
    B_kon.append(i)
    pyt_kon[i]=time.time()-tim

tim=time.time()
for i in range(30000):
    A_pocz.add(i)
    nasz_pocz[i]=time.time()-tim

tim=time.time()
for i in range(30000):
    A_kon.append(i)
    nasz_kon[i]=time.time()-tim
```

In [111…

```
t
```

Out[111… `range(0, 30000)`

In [111…

```
nasz_kon[-1]
```

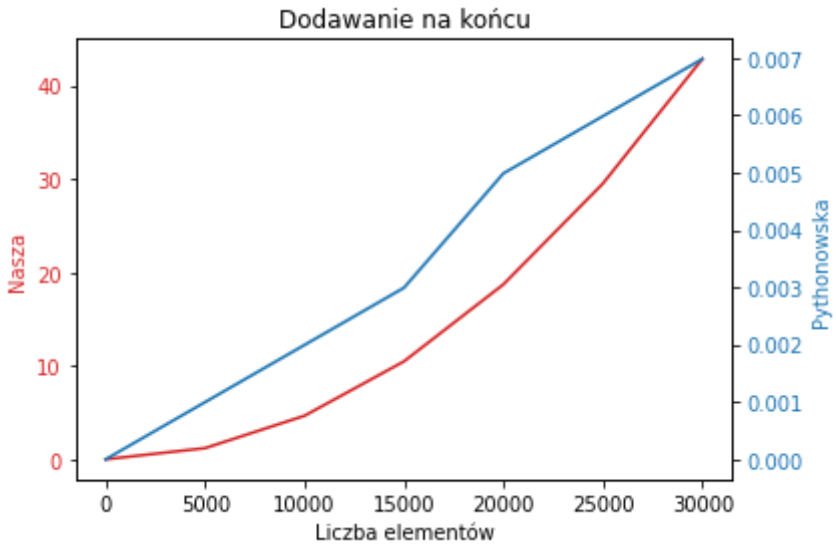
Out[111… `42.86649751663208`

In [ ]:

In [113…

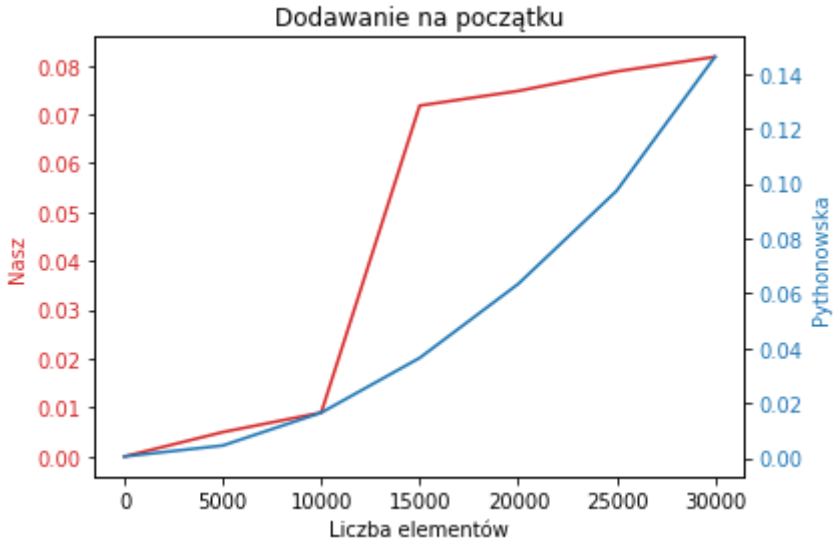
```
t = range(30000)
fig, ax1 = plt.subplots()
```

```
plt.title("Dodawanie na końcu")

color = 'tab:red'
ax1.set_xlabel('Liczba elementów')
ax1.set_ylabel('Nasza', color=color)
ax1.plot(t[::4999], nasz_kon[::4999], color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis

color = 'tab:blue'
ax2.set_ylabel('Pythonowska', color=color)  # we already handled the x-label with ax
ax2.plot(t[::4999], pyt_kon[::4999], color=color)
ax2.tick_params(axis='y', labelcolor=color)

fig.tight_layout()  # otherwise the right y-label is slightly clipped
plt.show()
```



In [113…

```
t = range(30000)
fig, ax1 = plt.subplots()

plt.title("Dodawanie na początku")

color = 'tab:red'
ax1.set_xlabel('Liczba elementów')
ax1.set_ylabel('Nasz', color=color)
ax1.plot(t[::4999], nasz_pocz[::4999], color=color)
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()  # instantiate a second axes that shares the same x-axis

color = 'tab:blue'
ax2.set_ylabel('Pythonowska', color=color)  # we already handled the x-label with ax
ax2.plot(t[::4999], pyt_pocz[::4999], color=color)
ax2.tick_params(axis='y', labelcolor=color)

fig.tight_layout()  # otherwise the right y-label is slightly clipped
plt.show()
```



Z powyższych wykresów możemy odczytać, że w dodawaniu na końcu listy pythonowskie mają nieporównywalną przewagę (przy 3e4 elementów ułamek sekundy kontra ponad 40s), natomiast nasze listy wykrywają w dodawaniu elementów na początku listy (około dwukrotna przewaga). Jeśli natomiast rozpatrzymy najszybsze dodawanie elementów to

In [114...
```
A = UnorderedList()
```

In [114...
```
%%timeit -n 1000000
A.add(1)
```

```
416 ns ± 2.88 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

In [114...
```
B=[]
```

In [114...
```
%%timeit -n 1000000
B.append(1)
```

```
55.2 ns ± 3.4 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

To listy pythonowskie również mają znaczną przewagę, ale w przeciwieństwie do przypadku dodawania na końcu listy jesteśmy w stanie porównać te dwa czasy (przy 1e6 różnica jest jednego rzędu)

Porównajmy teraz czas zdejmowania elementów

In [124...
```
A=UnorderedList()
B=[]
```

In [125...
```
for _ in range(1000):
    A.add(0)
    B.append(0)
```

In [125...
```
%%timeit -n 100
A.pop()
```

```
148 µs ± 52.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [125…
```
%%timeit -n 100
B.pop()
```

```
42.6 ns ± 12.6 ns per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [125…
```
A=UnorderedList()
B=[]
for _ in range(10000):
    A.add(0)
    B.append(0)
```

In [125…
```
%%timeit -n 100
A.pop()
```

```
2.21 ms ± 58.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [125…
```
%%timeit -n 100
B.pop()
```

```
86 ns ± 0.756 ns per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [125…
```
A=UnorderedList()
B=[]
for _ in range(1000):
    A.add(0)
    B.append(0)
```

In [125…
```
%%timeit -n 100
A.pop(0)
```

```
88.7 µs ± 33.9 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [125…
```
%%timeit -n 100
B.pop(0)
```

```
158 ns ± 14.2 ns per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [125…
```
A=UnorderedList()
B=[]
for _ in range(10000):
    A.add(0)
    B.append(0)
```

In [126…
```
%%timeit -n 100
A.pop(0)
```

```
1.31 ms ± 28 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [126…
```
%%timeit -n 100
B.pop(0)
```

```
1.38 µs ± 80.6 ns per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Możemy zauważyć, że listy pythonowskie są szybsze w obu przypadkach, niezależnie czy zabieramy element z początku, czy z końca listy, ale co ważniejsze zwiększenie wielkości listy 10-cio krotnie zwiększa zwiększa w małym stopniu czas trwania funkcji (w gorszym przypadku

liniowo), natomiast czas trwania naszej listy rośnie znacznie szybciej przz co przy dużych ilościach elementów nasze listy robią więcej szkód niż pożytku

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [119…

```
# tavela czasow
pyt=[0 for i in range(10000000)]
nasz=[0 for i in range(10000000)]



#liczenie czasow dodawania
tim=time.time()
for i in range(10000000):
    B.pop()
    pyt[i]=time.time()-tim

tim=time.time()
for i in range(10000000):
    A.pop()
    nasz[i]=time.time()-tim
```

In [ ]:

In [ ]:

In [ ]: