

Algorytmy i struktury danych - Lista 4

Alicja Myśliwiec - gr wtorek 7:30

```
In [1]: import time, random
import matplotlib.pyplot as plt
b.enqueue(5)
b.dequeue()
```

Zad. 1 Zaimplementuj kolejki przy użyciu pythonowych list w taki sposób, aby:

- koniec kolejki znajdował się na końcu listy,

```
In [2]: class QueueBaB(object):
def __init__(self):
    self.list_of_items = []

def isEmpty(self):
    if not self.list_of_items:
        return True
    return False

def enqueue(self, item):
    #możliwość do dodanie wielu elementów na raz do kolejki
    self.list_of_items.append(item)

def dequeue(self):
    return self.list_of_items.pop(0)

def size(self):
    return len(self.list_of_items)

def __str__(self):
    return str(self.list_of_items)
```

```
In [3]: b = QueueBaB()
b.enqueue(3)
b.enqueue(4)
b.enqueue(5, 7, 9, 10)
b.dequeue()
print(b)

[4, 5, 7, 9, 10]
```

```
In [4]: b = QueueBaB()
b.enqueue(1)
print(b.isEmpty())
b.dequeue()
print(b.isEmpty())

False
True
```

• koniec kolejki znajdował się na początku listy.

```
In [5]: class QueueBaB(object):
def __init__(self):
    self.list_of_items = []

def isEmpty(self):
    if not self.list_of_items:
        return True
    return False

def enqueue(self, item):
    for item in list(items):
        self.list_of_items.insert(0, item)

def dequeue(self):
    return self.list_of_items.pop()

def size(self):
    return len(self.list_of_items)

def __str__(self):
    return str(self.list_of_items)
```

```
In [6]: b = QueueBaB()
b.enqueue(3)
b.enqueue(4)
b.enqueue(5, 7, 9, 10)
b.dequeue()
print(b, b.size())

[10, 9, 7, 5, 4] 5
```

Zad. 2 Zaprojektuj i przeprowadź eksperyment porównujący wydajność obu implementacji.

W poniższym zadaniu, porównaj metody enqueue, dequeue oraz obie jednocześnie dla obu kolejek

```
In [7]: def enqueue_times(n):
BaB = QueueBaB()
BaE = QueueBaE()

start_BaB = time.time()
for i in range(0, n):
    BaB.enqueue(i)
end_BaB = time.time()
time_BaB = end_BaB - start_BaB

start_BaE = time.time()
for i in range(0, n):
    BaE.enqueue(i)
end_BaE = time.time()
time_BaE = end_BaE - start_BaE

return [time_BaB, time_BaE], [BaB, BaE]
```

```
In [8]: def dequeue_times(n):
BaB = enqueue_times(n)[1][0]
BaE = enqueue_times(n)[1][1]

start_BaB = time.time()
for i in range(0, n):
    BaB.dequeue(i)
end_BaB = time.time()
time_BaB = end_BaB - start_BaB

start_BaE = time.time()
for i in range(0, n):
    BaE.dequeue(i)
end_BaE = time.time()
time_BaE = end_BaE - start_BaE

return [time_BaB, time_BaE]
```

```
In [88]: def enq_and_deq(n):
BaB = QueueBaB()
BaE = QueueBaE()

start_BaB = time.time()
for i in range(0, n):
    BaB.enqueue(i)
for i in range(0, n):
    BaB.dequeue(i)
end_BaB = time.time()
time_BaB = end_BaB - start_BaB

start_BaE = time.time()
for i in range(0, n):
    BaE.enqueue(i)
for i in range(0, n):
    BaE.dequeue(i)
end_BaE = time.time()
time_BaE = end_BaE - start_BaE

return [time_BaB, time_BaE]
```

Teraz, dla konkretnej liczby n, wywołaj, stwórz funkcję stałą. Następnie sprawdźmy czy faktycznie zależność jest liniowa.

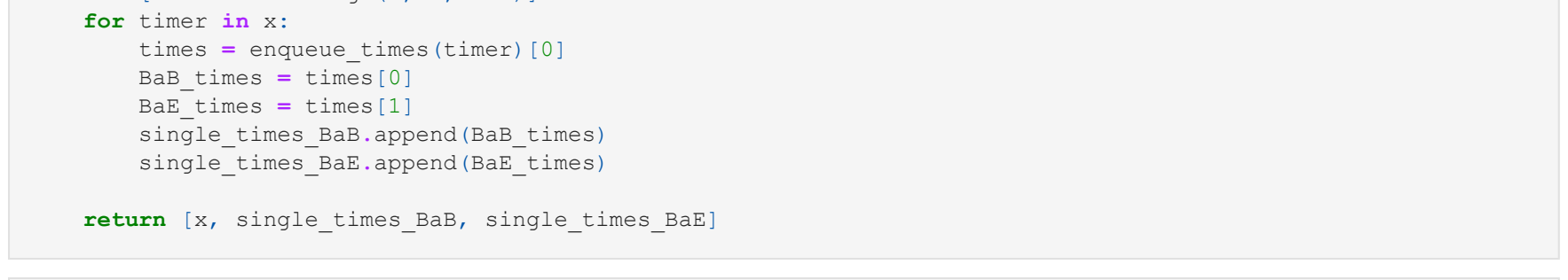
```
In [124]: def compare_enqueue(n):
x = np.linspace(0, 20, 21)
values_BaB = enqueue_times(n)[0][1] * x
values_BaE = enqueue_times(n)[0][1] * x
plt.plot(x, values_BaB, cm="blue", label="BaB")
plt.plot(x, values_BaE, cm="red", label="BaE")
plt.title("Comparison of enqueue methods")
plt.xlabel("Constant function for n = {}".format(n))
plt.ylabel("Time")
plt.legend()
plt.grid()
plt.show()
```

```
In [125]: compare_enqueue(10000)
```



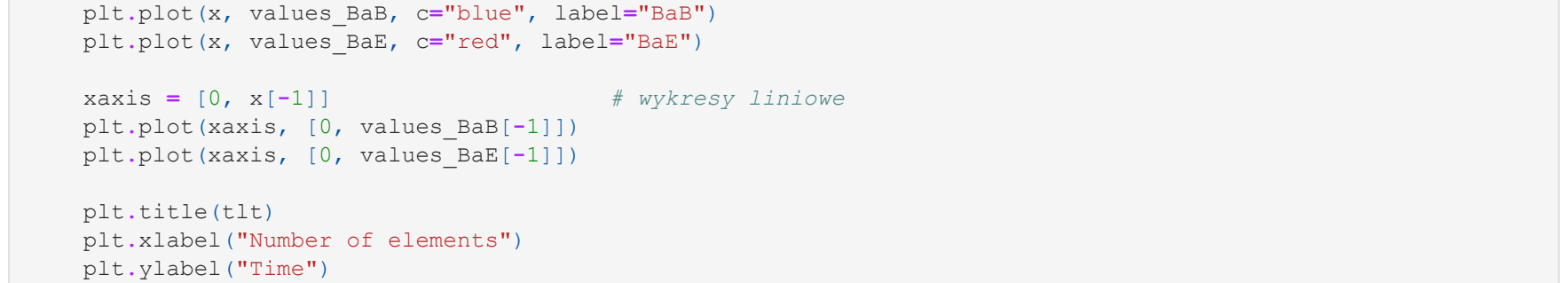
```
In [130]: def compare_dequeue(n):
x = np.linspace(0, 20, 21)
values_BaB = dequeue_times(n)[1] * x
values_BaE = dequeue_times(n)[1] * x
plt.plot(x, values_BaB, cm="blue", label="BaB")
plt.plot(x, values_BaE, cm="red", label="BaE")
plt.title("Comparison of dequeue methods")
plt.xlabel("Constant function for n = {}".format(n))
plt.ylabel("Time")
plt.legend()
plt.grid()
plt.show()
```

```
In [131]: compare_dequeue(10000)
```



```
In [128]: def compare_both(n):
x = np.linspace(0, 20, 21)
values_BaB = enq_and_deq(n)[0] * x
values_BaE = enq_and_deq(n)[1] * x
plt.plot(x, values_BaB, cm="blue", label="BaB")
plt.plot(x, values_BaE, cm="red", label="BaE")
plt.title("Comparison of enqueue and dequeue methods")
plt.xlabel("Constant function for n = {}".format(n))
plt.ylabel("Time")
plt.legend()
plt.grid()
plt.show()
```

```
In [129]: compare_both(10000)
```



Według wykresów, w ogóle lepiej sprawuje się kolejka BaB - choć przy metodzie dequeue jest wolniejsza, osobne wyniki metod wynikają z implementacji metod wbudowanych dla klasycznej pythonowej listy.

Teraz sprawdź jak wygląda wykres sprawdzając czas dla każdego n po kolei

```
In [132]: def get_enqueue_times(n):
single_times_BaB, single_times_BaE = [], []
x = [x for x in range(0, n, 100)]
for timer in x:
    times = enqueue_times(timer)[0]
    BaB_times = times[0]
    BaE_times = times[1]
    single_times_BaB.append(BaB_times)
    single_times_BaE.append(BaE_times)

return [x, single_times_BaB, single_times_BaE]
```

```
In [133]: def get_dequeue_times(n):
single_times_BaB, single_times_BaE = [], []
x = [x for x in range(0, n, 100)]
for timer in x:
    times = dequeue_times(timer)
    BaB_times = times[0]
    BaE_times = times[1]
    single_times_BaB.append(BaB_times)
    single_times_BaE.append(BaE_times)

return [x, single_times_BaB, single_times_BaE]
```

```
In [134]: def compare_time(n, name):
"""
data = get_enqueue_times(n)
titl = "Comparison of enqueue method"
elif name == "dequeue":
    data = get_dequeue_times(n)
    titl = "Comparison of dequeue method"

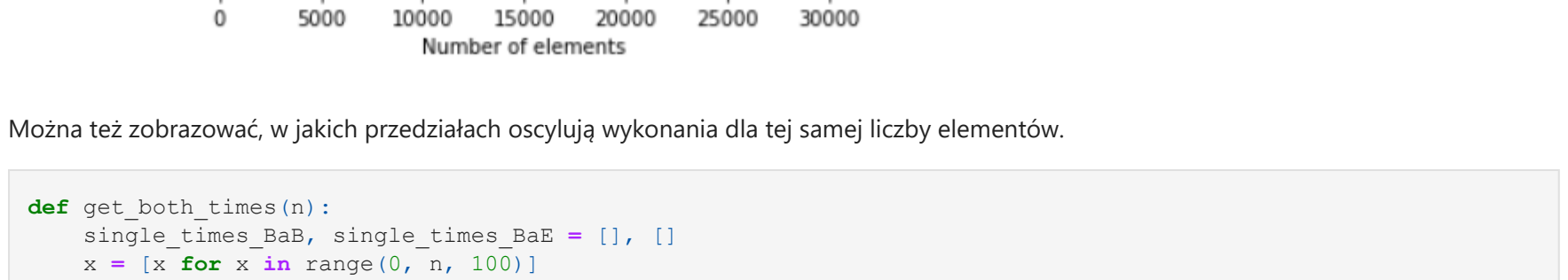
x = data[0]
values_BaB = data[1]
values_BaE = data[2]
plt.plot(x, values_BaB, cm="blue", label="BaB")
plt.plot(x, values_BaE, cm="red", label="BaE")
plt.xlabel("Number of elements")
plt.ylabel("Time")
plt.grid()
plt.show()

# wykresy liniowe
plt.plot(xaxis, [0, values_BaB[-1]])
plt.plot(xaxis, [0, values_BaE[-1]])
```

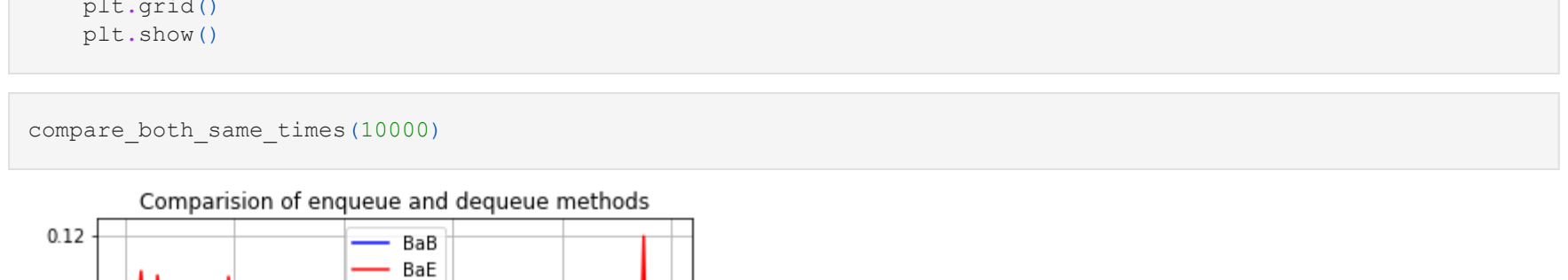
```
In [135]: compare_time(10000, "enqueue")
```



```
In [136]: compare_time(10000, "dequeue")
```



Dla większej liczby elementów wykresy się coraz bardziej pokrywają



Można też zobaczyć, w jakich przedziałach oscylują wykonania dla tej samej liczby elementów.

```
In [22]: def get_both_times(n):
single_times_BaB, single_times_BaE = [], []
x = [x for x in range(0, n, 100)]
for timer in x:
    times = enq_and_deq(n)
    BaB_times = times[0]
    BaE_times = times[1]
    single_times_BaB.append(BaB_times)
    single_times_BaE.append(BaE_times)

return [x, single_times_BaB, single_times_BaE]
```

```
In [23]: def compare_both_same_times(n):
data = get_both_times(n)
x = data[0]
values_BaB = data[1]
values_BaE = data[2]
plt.plot(x, values_BaB, cm="blue", label="BaB")
plt.plot(x, values_BaE, cm="red", label="BaE")
plt.title("Comparison of enqueue and dequeue methods")
plt.xlabel("Number of executions")
plt.ylabel("Time")
plt.grid()
plt.show()
```

```
In [24]: compare_both_same_times(10000)
```



Ponownie widać przewagę kolejki BaB

Zad. 3 Rozważ sytuację z życia wziętą, np.:

- auta w kolejce do myjni,
- kaszy w supermarkiecie,
- samoloty na pasie startowym,
- okenko w banku.

Postaw pytanie badawcze. Wykorzystując liniowe struktury danych zaprojektuj i przeprowadź symulację, która udzieli na nie odpowiedzi. Pamiętaj o określeniu wszystkich zmiennych swojego modelu.

Bistro

Z uwagi na panującą obstrzernia, w pewnym bistro jest określona maksymalna liczba osób, która może przebywać jednocześnie w lokalu.

Symulację przeprowadzane są przy założeniu, iż restrykcje się zmieniają oraz pozostaje dysproporcja w tempie pracy poszczególnych kasjerek. Czas obsługi konkretnego klienta również jest zróżnicowany (np. nakładą większą ilość jedzenia). Z zesזורzonych danych również wiemy, że bistro odwiedza od 8 do 15 osób na godzinę.

Każdy klient, który w godzinę zamknięcia bistro stoi w kolejce poza lokalem, niestety nie może zostać obsłużony. Ilic średnio osób dziennie, nie zostaje obsługiwanych?

```
In [137]: class Bistro:
def __init__(self, restriction, faster_service_pace):
    """
    :param restriction: how many clients can wait inside, pandemic times :
    :param faster_service_pace: client can be serviced by different cashiers
    True = faster, False = slower cashier!
    """
    self.current_client = None
    self.open_hours = 360
    self.full_queue_inside = False
    self.restricted_amount_of_clients = restriction
    self.faster_service_pace = faster_service_pace

def new_clients(self):
    """
    Function determine how many clients will arrive in specific hour
    :return: True if it's a full hour, if not - False
    """
    if self.open_hours in [60, 120, 180, 240, 300, 360]:
        number = random.randint(8, 15)
        self.clients = number
        return True
    return False

def every_minute_tick(self):
    self.open_hours -= 1

def next_to_the_counter(self, new_client):
    """
    next customer is served
    """
    self.current_client = new_client
```

```
In [138]: class Client:
def __init__(self, spent_time):
    """
    :param spent_time: Time that single client will be served
    """
    self.time_remain = spent_time

def waiting(self):
    self.time_remain -= 1

def has_been_served(self):
    """
    Function determine whether client can leave the queue
    :return:
    if self.time_remain == 0:
        return True
    return False
```

```
In [139]: def single_simulation(restrict, fsp):
"""
:return: amount of people outside
"""
bistro = Bistro(restrict, fsp)
queue = QueueBaB()
next_client = None
clients_outside = 0
client_time_range = [4, 5, 6, 7]
client_time = random.choice(client_time_range)

if not bistro.faster_service_pace:
    client_time *= random.randint(1, 2) # slower cashier makes service time longer

while bistro.open_hours > 0:
    if bistro.new_clients():
        client = Client(client_time)
        queue.enqueue(client)

    if queue.isEmpty(): # nobody in queue, time keep going
        bistro.every_minute_tick()
        continue

    if bistro.current_client is None or next_client.has_been_served():
        next_client = queue.dequeue()
        bistro.next_to_the_counter(next_client)

    if queue.size() >= bistro.restricted_amount_of_clients:
        bistro.full_queue_inside = True
    else:
        bistro.full_queue_inside = False

    next_client.waiting()
    bistro.every_minute_tick()

    if bistro.full_queue_inside:
        clients_outside = queue.size() - restrict

    return clients_outside
```

```
In [140]: def interpret(restrict, n):
"""
In this function I am going to interpret _1 as calculations for faster cashier, similarly _2 for the slower
:param n: how many time to simulate a single day
"""
list_of_clients_outside_1, list_of_clients_outside_2, x = [], [], []
average_amount_1, average_amount_2 = 0, 0

for i in range(n):
    data_1 = single_simulation(restrict, True)
    list_of_clients_outside_1.append(data_1)
    data_2 = single_simulation(restrict, False)
    list_of_clients_outside_2.append(data_2)
    x.append(i)

for num in list_of_clients_outside_1:
    average_amount_1 += num

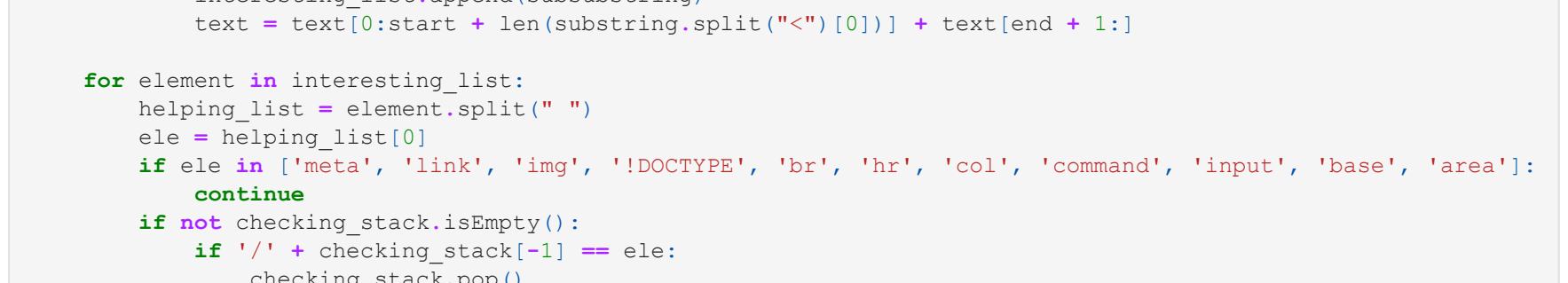
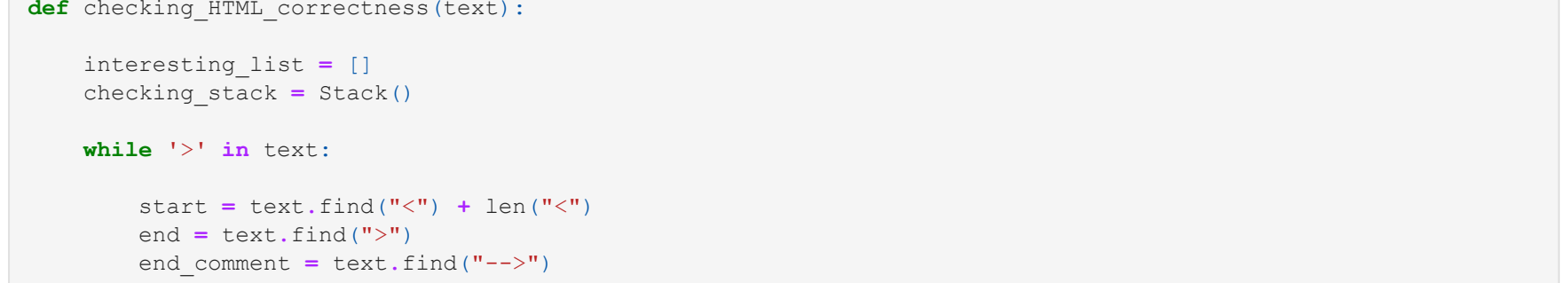
for num in list_of_clients_outside_2:
    average_amount_2 += num

result_1, result_2 = average_amount_1 / len(list_of_clients_outside_1), average_amount_2 / len(list_of_clients_outside_2)

values_1 = list_of_clients_outside_1
values_2 = list_of_clients_outside_2
plt.scatter(x, values_1, cm="blue", label="Faster")
plt.scatter(x, values_2, cm="red", label="Slower")

plt.plot(x, [result_1 for i in range(n)], label="average Faster")
plt.plot(x, [result_2 for i in range(n)], label="average Slower")

plt.title("Simulation")
plt.xlabel("Single days")
plt.ylabel("Customers outside")
plt.legend()
plt.grid()
plt.show()
```



Interpretacja wizualna nie mogła być inna - im więcej osób może przebywać w lokalu, tym więcej ich obsługuje po zamknięciu (średnia ilość osób w kolejce poza bistro). Również tempo kasjerki sprawia, że w ciągu dnia jest mniej lub więcej nieobsłużonych klientów.

Zad. 4 Napisz program, który sprawdzi poprawność składni dokumentu HTML pod kątem brakujących znaczników zamykających.

Program zacznie od zaimplementowania stosu

```
In [34]: class Stack:
def __init__(self):
    self.items = []

def isEmpty(self):
    return self.items == []

def push(self, item):
    self.items.append(item)

def pop(self):
    return self.items.pop()

def peek(self):
    return self.items[len(self.items) - 1]

def size(self):
    return len(self.items)

def getitem(self, item):
    return self.items[item]

def __str__(self):
    return str(self.items)
```

Następnie funkcja właściwa. Wydzieli ona elementy znajdujące się pomiędzy znacznikami "<" oraz ">". Jeśli jest to komentarz - usuwa go w całości (komentarz może być wszystko, tzn. przykładem pozwolone znaczniki). Odpowiednio wysklekowane elementy funkcja kolejno lub wrzuci na stos, lub je ściera.

```
In [35]: def checking_HTML_correctness(text):
interesting_list = []
checking_stack = Stack()

while '<' in text:
    start = text.find("<") + len("<")
    end = text.find(">")
    end_comment = text.find("-->")
    substr = text[start:end]

    if substr[0:3] == "!--":
        text = text[0:start - 1] + text[end_comment + len("-->"):]
        continue

    if len(substr.split("<") > 1:
        if len(interesting_list) > 0:
            text = text[0:start - 1] + text[end + 1:]
        else:
            start2 = substr.find("<") + len("<")
            substr2 = substr[start2:end]
            interesting_list.append(substr2)
            text = text[0:start + len(substr.split("<"))(0)] + text[end + 1:]

    for element in interesting_list:
        helping_list = element.split(" ")
        if helping_list[0]:
            if ele in ["a", "b", "i", "img", "input", "base", "area"]:
```

Funkcja ta dla potrzeb działania w Jupyterze została zmodyfikowana tak, aby przyjmowała string nie plik jako argument. W pliku .py jest zadana poprawnie i generuje następujące wyniki dla zadanych kodów HTML.

```
>>> if __name__ == '__main__':
print(checking_HTML_correctness('1. J&A; sample HTML 1.txt'))
print(checking_HTML_correctness('1. J&A; sample HTML 2.txt'))
print(checking_HTML_correctness('1. J&A; sample HTML 3.txt'))

checking_HTML_correctness()

C:\Users\A\luka\PycharmProjects\Lista4_alg\venv\Scripts\python.exe C:\Users\A\luka\PycharmProjects\Lista4_alg\1. J&A;4.py
True
False
False
Process finished with exit code 0
```

Krótki przykład działania programu:

Pierwszy przykład jest poprawny

```
In [36]: sample_1 = "<head> <p> <meta> <!-- --> </head>"
checking_HTML_correctness(sample_1)
```

W drugim natomiast brakuje jednego domknięcia "</div>". Dlatego otrzymujemy False

```
In [37]: sample_2 = "<foote> <div> <div> <div> <ul> <!-- Facebook ===== --></ul> <p> </p> </div> </div>"
checking_HTML_correctness(sample_2)
```

Dodajmy do przykładu brakujący zamykający "</div>". Teraz wszystko jest w porządku.

```
In [38]: sample_2_repaired = "<foote> <div> <div> <div> <ul> <!-- Facebook ===== --></ul> <p> </p> </div> </div>"
checking_HTML_correctness(sample_2_repaired)
```

Ostatni przykład, w którym widać że zawartość komentarzy nie ma znaczenia.

```
In [39]: sample_3 = "<head> <a> <a> <!-- <title> <p> --> </head>"
checking_HTML_correctness(sample_3)
```

Wszystko jest poprawne.

Zad. 5 Dodaj brakujące metody do klasy UnorderedList prezentowanej na wykładzie.

```
In [40]: class Node:
def __init__(self, init_data):
    self.data = init_data
    self.next = None

def getData(self):
    return self.data

def getNext(self):
    return self.next

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

def setNext(self, new_next):
    self.next = new_next
```



```
[41]: class UnorderedList(object):
def __init__(self):
self.head = None

def isEmpty(self):
return self.head is None

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

def size(self):
current = self.head
count = 0
while current is not None:
count += 1
current = current.getNext()

return count

def search(self, item):
current = self.head
found = False
while current is not None and not found:
if current.getData() == item:
found = True
else:
current = current.getNext()

return found

def remove(self, item):
current = self.head
previous = None
found = False
while not found:
if current.getData() == item:
found = True
else:
previous = current
current = current.getNext()

if previous is None: # jeśli usuwamy pierwszy element
self.head = current.getNext()
else:
previous.setNext(current.getNext())

# ----- SELF-ADDED METHODS -----

def append(self, item):
"""
Method to add an item to the end of the list.
:param item: object to add
"""
current = self.head
temp = Node(item)

if self.isEmpty():
self.add(item)
else:
while current.getNext() is not None:
current = current.getNext()
current.setNext(temp)

def index(self, item):
"""
The method gives a place on the list, which has a specific element -
the element underneath self.head has an index 0.
:param item: element, whose position is to be determined
:return: item position on the list or None in the case of,
when the chosen item is not on the list
"""
index_pos = 0
current = self.head

while current is not None:
if current.getData() == item:
return index_pos
else:
current = current.getNext()
index_pos += 1

return None

def insert(self, pos, item):
"""
The method places a given element on the given position.
Takes the position as arguments,
on which to place the element and this element.
:param pos: position on which to place the element
:param item: element to place
"""
temp = Node(item)
current, current_pos, size = self.head, 0, self.size()

if -size <= pos < 0: # odczytanie indeksu ujemnego
pos += size + 1

if pos == 0:
self.add(item)
elif pos == size:
self.append(item)
elif not 0 <= pos < size:
raise IndexError("Incorrect index")
else:
while current_pos != (pos - 1):
current = current.getNext()
current_pos += 1

next_for_temp = current.getNext()
temp.setNext(next_for_temp)
current.setNext(temp)

def pop(self, pos=-1):
"""
The method removes an item from the list from the specific place.
:param pos: optional: position .
:return: deleted element
"""
current, current_pos, size = self.head, 0, self.size()
previous = None
limes = size

if self.isEmpty():
raise IndexError("empty list, nothing to pop")
if not 0 <= pos <= limes:
raise IndexError("Incorrect index")

while current is not None:
if current_pos == 0:
if pos == -1 and size == 1:
self.head = previous
return current.getData()
elif pos == 0:
self.head = current.getNext()
return current.getData()
if current_pos is [pos, pos + size]:
previous.setNext(current.getNext())
return current.getData()

previous = current
current = current.getNext()
current_pos += 1

# -----

def __str__(self):
current = self.head
li = []
while current is not None:
li.append(current.getData())
current = current.getNext()
s = "Elements in the list are [" + ', '.join('{}' * len(li)) + "]"
return s.format(*li)

In [154]: mylist = UnorderedList()
mylist.add(3)
mylist.add(44)
print(mylist)
mylist.insert(1, 15)
print(mylist)
print(mylist.pop())
print(mylist.search(15))

elements in the list are [4, 3]
elements in the list are [4, 15, 3]
3
True
```

Zad. 6 Zaimplementuj stos przy pomocy listy jednokierunkowej.

```
In [43]: class StackUsingUL(object):
def __init__(self):
self.items = UnorderedList()

def isEmpty(self):
"""
A method to check if the stack is empty.
"""
return self.items.isEmpty()

def push(self, item):
"""
The method places a new item on the stack.
:param item: item to place
"""
self.items.append(item)

def pop(self):
"""
The method pops the item off the stack.
:return: popped element
"""
if self.isEmpty():
raise IndexError("Stack is empty")
return self.items.pop()

def peek(self):
"""
The method gives the value of the item on top of the stack without taking it off.
:return: the top element of the stack
"""
if self.isEmpty():
raise IndexError("Stack is empty")

current = self.items.head
while current.getNext() is not None:
current = current.getNext()
value = current.getData()
return value

def size(self):
"""
:return: the number of items on the stack
"""
return self.items.size()

def __str__(self):
return str(self.items)

In [44]: if __name__ == '__main__':
stack = StackUsingUL()
stack.push(4)
stack.push(54)
stack.push(44)
print(stack)
print(stack.peek(), stack.size())
stack.push(6)
print(stack)
print(stack.pop())
print(stack)
stack.pop()
stack.pop()
print(stack)
stack.pop()

elements in the list are [4, 54, 44]
44 3
elements in the list are [4, 54, 44, 6]
6
elements in the list are [4, 54, 44]
elements in the list are []

IndexError                                Traceback (most recent call last)
<ipython-input-44-32ac40f2f525> in <module>
13     stack.pop()
15     print(stack)
----> 16     stack.pop()
21
<ipython-input-43-a6d825b713c2> in pop(self)
22
23     if self.is_empty():
----> 24         raise IndexError("Stack is empty")
25
26     return self.items.pop()
IndexError: Stack is empty
```

Zad. 7 Zaimplementuj kolejkę dwustronną przy pomocy listy jednokierunkowej.

```
In [45]: class DequeueUsingUL(object):
def __init__(self):
self.items = UnorderedList()

def isEmpty(self):
"""
A method to check if the queue is empty.
"""
return self.items.isEmpty()

def add_left(self, item):
"""
The method adds an item to the queue on the left.
:param item: the item to be added
"""
self.items.add(item)

def add_right(self, item):
"""
The method adds an item to the queue on the right.
:param item: the item to be added
"""
self.items.append(item)

def remove_left(self):
"""
The method removes the element from the queue on the left.
:return: removed item
"""
if self.isEmpty():
raise IndexError("Queue is empty")

return self.items.pop(0)

def remove_right(self):
"""
The method removes the element from the queue on the right.
:return: removed item
"""
if self.isEmpty():
raise IndexError("Queue is empty")

return self.items.pop()

def size(self):
"""
:return: the number of items in the queue
"""
return self.items.size()

def __str__(self):
return str(self.items)

In [46]: if __name__ == '__main__':
queue = DequeueUsingUL()
queue.add_left(4)
queue.add_left(54)
queue.add_right(44)
print(queue)
print(queue.size())
queue.remove_left()
print(queue)
queue.remove_right()
print(queue)
queue.remove_left()
print(queue)
queue.remove_right()

elements in the list are [54, 4, 44]
3
elements in the list are [54, 4]
elements in the list are [4]
elements in the list are []

IndexError                                Traceback (most recent call last)
<ipython-input-46-780e9ba543d> in <module>
12     queue.remove_left()
13     print(queue)
----> 14     queue.remove_right()
21
<ipython-input-45-f63234760e2d> in remove_right(self)
40     """
41     if self.is_empty():
----> 42         raise IndexError("Queue is empty")
43
44     return self.items.pop()
IndexError: Queue is empty
```

Zad. 8 Zaprojektuj i przeprowadź eksperyment porównujący wydajność listy jednokierunkowej i listy wbudowanej w Pythona.

W tym zadaniu będzie porównywać czas wykonania operacji append, insert oraz pop pomiędzy listą jednokierunkową i wbudowaną.

```
In [47]: def pop_times(n):
python_list = []
un_list = UnorderedList()

start_python = time.time()
for i in range(0, n):
python_list.append(n)
end_python = time.time()
time_python = end_python - start_python

start_un = time.time()
for i in range(0, n):
un_list.append(i)
end_un = time.time()
time_un = end_un - start_un

return [time_python, time_un]

In [48]: for i in range(5):
check = append_times(2000)
print("Time for python list: {}".format(check[0])+"n"+"Time for unordered list: {}".format(check[1])+"n")

Time for python list: 0.00193975503349609375
Time for unordered list: 1.3154803864793418

Time for python list: 0.0
Time for unordered list: 1.3462212085723877

Time for python list: 0.0
Time for unordered list: 1.4015278816223145

Time for python list: 0.0
Time for unordered list: 1.3925375938415527

Time for python list: 0.0
Time for unordered list: 1.2997558116912842

In [52]: def insert_times(n):
python_list = []
un_list = UnorderedList()

start_python = time.time()
for i in range(0, n):
python_list.insert(0, i)
end_python = time.time()
time_python = end_python - start_python

start_un = time.time()
for i in range(0, i)
un_list.insert(0, i)
end_un = time.time()
time_un = end_un - start_un

return [time_python, time_un]

In [53]: for i in range(5):
check = insert_times(2000)
print("Time for python list: {}".format(check[0])+"n"+"Time for unordered list: {}".format(check[1])+"n")

Time for python list: 0.0019936561584472656
Time for unordered list: 0.833401848020312

Time for python list: 0.00499010860595703
Time for unordered list: 0.8193018436431885

Time for python list: 0.003987550735473633
Time for unordered list: 1.0464789867401123

Time for python list: 0.00193931793212890625
Time for unordered list: 0.9790396690368552

Time for python list: 0.003265192779541016
Time for unordered list: 0.8619673252105713

In [57]: def pop_times(n):
python_list = []
un_list = UnorderedList()
for i in range(0, n):
un_list.add(i)

start_python = time.time()
for i in range(n):
python_list.pop()
end_python = time.time()
time_python = end_python - start_python

start_un = time.time()
for i in range(n):
un_list.pop()
end_un = time.time()
time_un = end_un - start_un

return [time_python, time_un]

In [58]: for i in range(5):
check = pop_times(2000)
print("Time for python list: {}".format(check[0])+"n"+"Time for unordered list: {}".format(check[1])+"n")

Time for python list: 0.000997334916381836
Time for unordered list: 2.442253828048706

Time for python list: 0.0
Time for unordered list: 2.7217960357666016

Time for python list: 0.0
Time for unordered list: 2.795039653778076

Time for python list: 0.0
Time for unordered list: 2.5997231006622314

Time for python list: 0.000997334916381836
Time for unordered list: 2.0144569873809814
```

Z powyższych symulacji dla aż 2000 wywołań, widać że lista wbudowana jest dużo wydajniejsza. Co do listy jednokierunkowej kolejnej jednak spójnie się metoda insert - czas schodzi poniżej sekundy. Najślabiej za to wypadła metoda pop, której wydolanie dla n=2000 dochodzi do 3s.

Wizualizacja:

```
In [75]: def compare(n, name):
x = [i for i in range(n)]
values_python, values_un = [], []

for timer in range(n):

if name == 'pop':
data = pop_times(n)
elif name == 'insert':
ttl = 'Comparison of pop method'
data = insert_times(n)
elif name == 'append':
ttl = 'Comparison of insert method'
data = append_times(n)
elif name == 'experiment': # na później :)
data = experiment(n)
ttl = 'Comparison of the three methods'

values_python.append(data[0])
values_un.append(data[1])
plt.scatter(x, values_python, c="blue", label="python")
plt.scatter(x, values_un, c="red", label="unordered")
plt.title(ttl)
plt.xlabel("Number of elements")
plt.ylabel("Time")
plt.legend()
plt.grid()
plt.show()
```

Ponieważ na wykonanie wykresu dla n=2000 czas oczekiwania jest dość duży i generuje duże obciążenie, wykres wykonamy dla mniejszych wartości n:

```
In [76]: compare(100, 'append')
compare(100, 'insert')
compare(100, 'pop')
```



Z wykresów łatwo odczytać, że lista wbudowana jest zdecydowanie szybsza.

Przykład wykonania wszystkich operacji

```
In [78]: def experiment(n):
python_list = []
un_list = UnorderedList()

start_python = time.time()
for i in range(0, n):
python_list.append(n)
python_list.insert(0, i)
python_list.pop()
end_python = time.time()
time_python = end_python - start_python

start_un = time.time()
for i in range(0, n):
un_list.append(i)
un_list.insert(0, i)
un_list.pop()
end_un = time.time()
time_un = end_un - start_un

return [time_python, time_un]

In [79]: compare(100, 'experiment')
```

