SPRAWOZDANIE - LISTA 4

Małgorzata Kowalczyk

Kamil Kowalski

07.12.2021

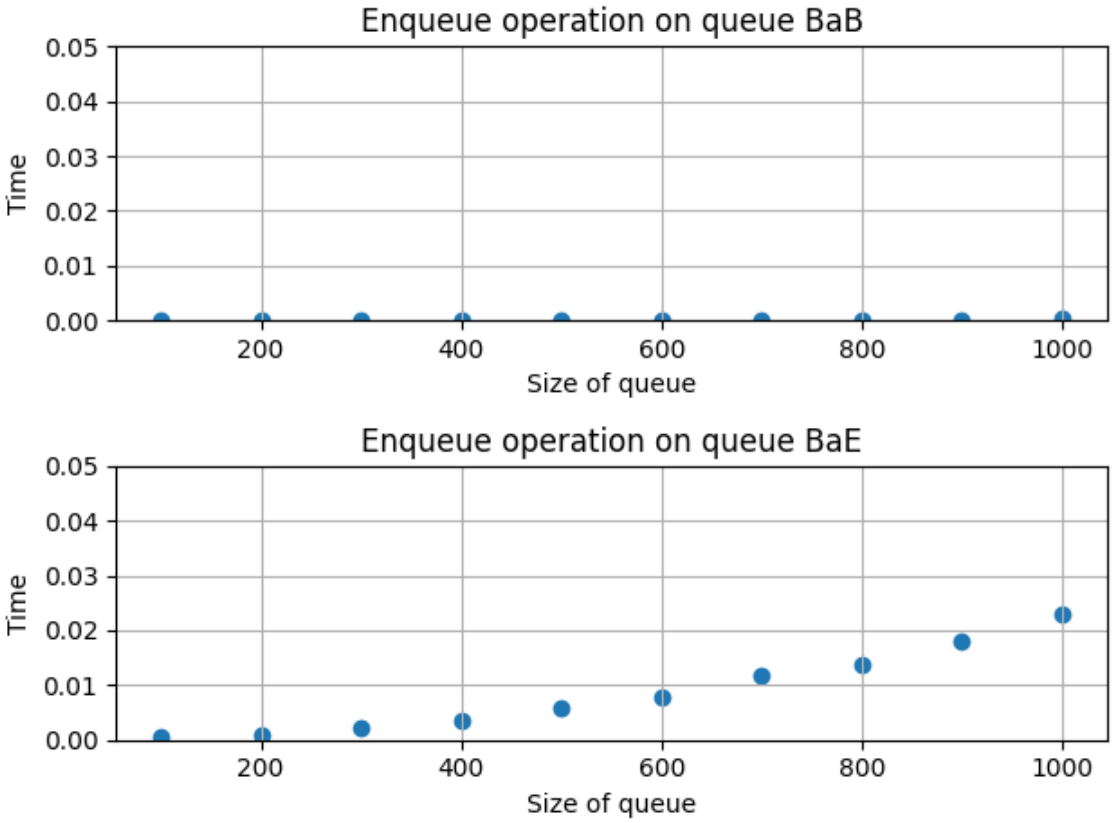
Zadanie 1

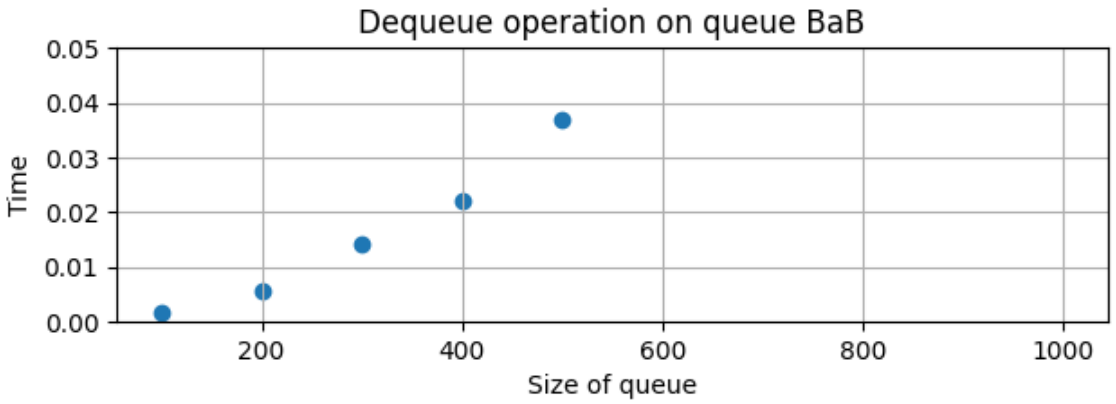
W zadaniu 1, naszym celem było zaimplementowanie kolejki na dwa różne sposoby. W pierwszym, koniec kolejki znajdował się na końcu listy, a w drugim na początku.

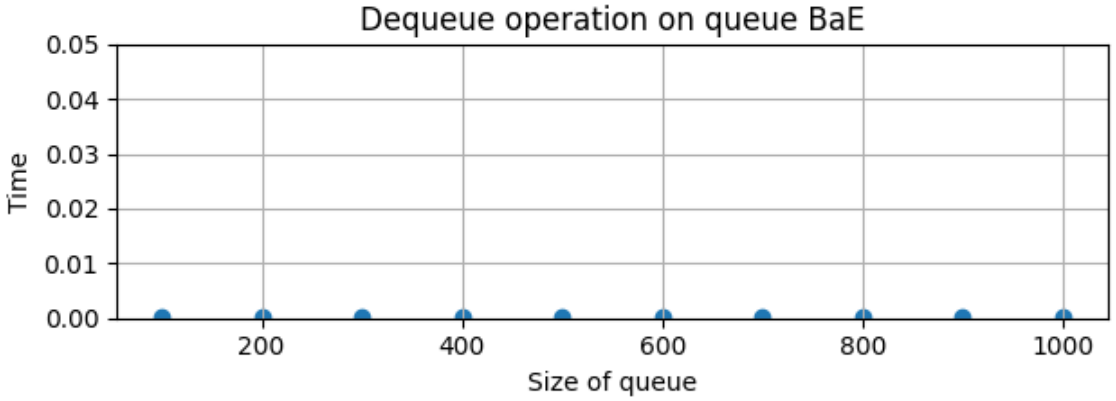
Zadanie 2

W tym zadaniu mieliśmy zaprojektować eksperyment, dzięki któremu dowiemy się, która z naszych implementacji kolejek jest wydajniejsza. Mierzyliśmy czasy dla funkcji enqueue(), jak i dequeue() dla kolejki zawierającej 1000 elementów. Wyniki są następujące:

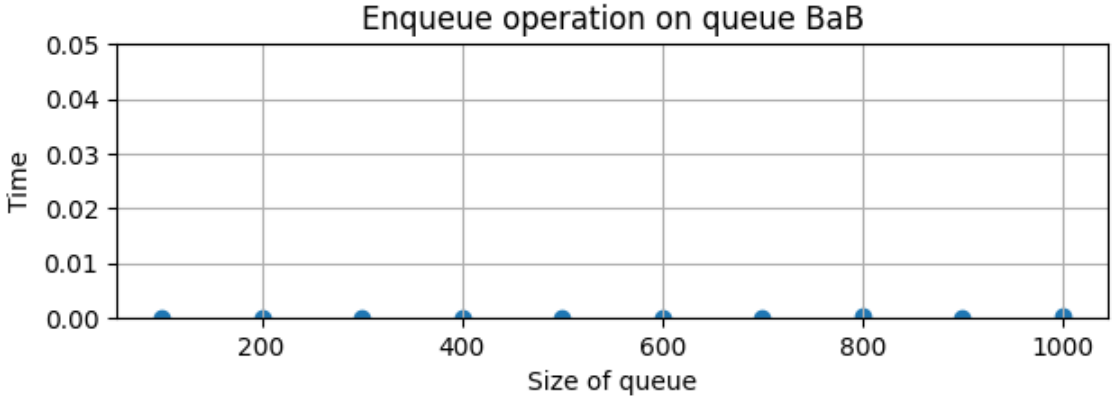
```
Size of queue: 1000
Time of enqueue operation for QueueBaB: 0.000368s
Time of enqueue operation for QueueBaE: 0.022896s
Time of dequeue operation for QueueBaB: 0.139594s
Time of dequeue operation for QueueBaE: 0.000378s
```

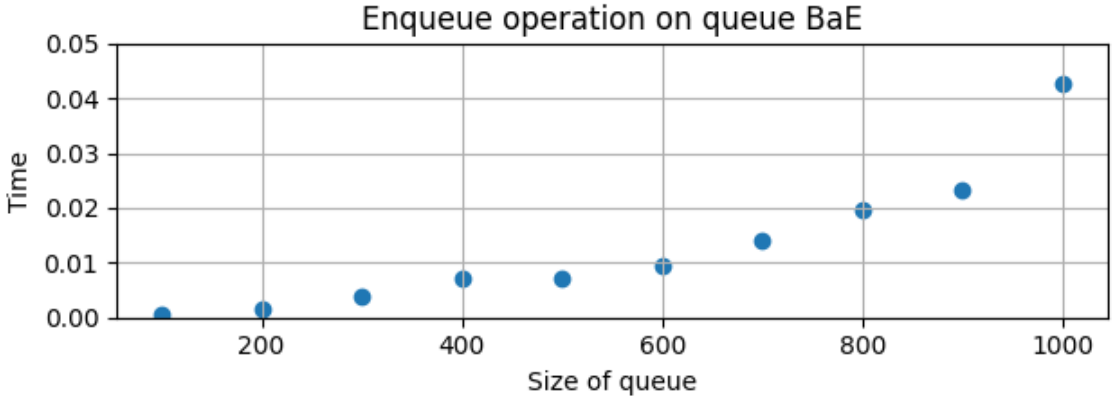


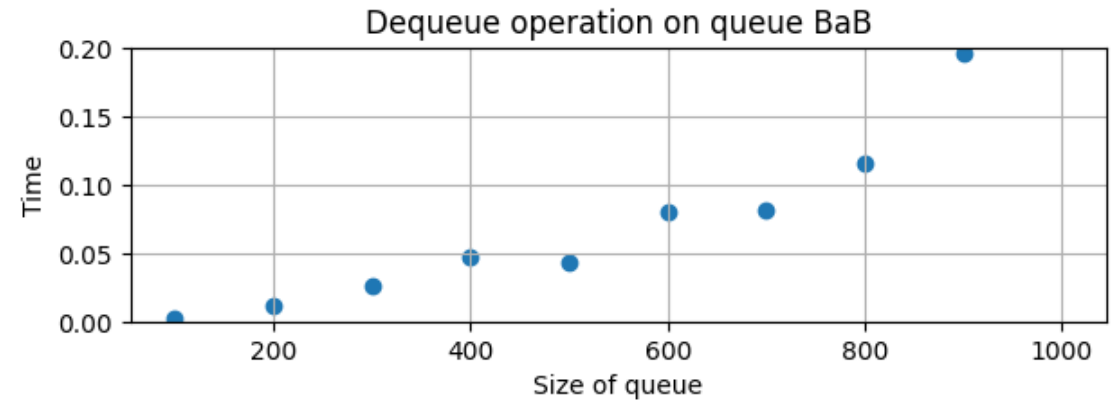


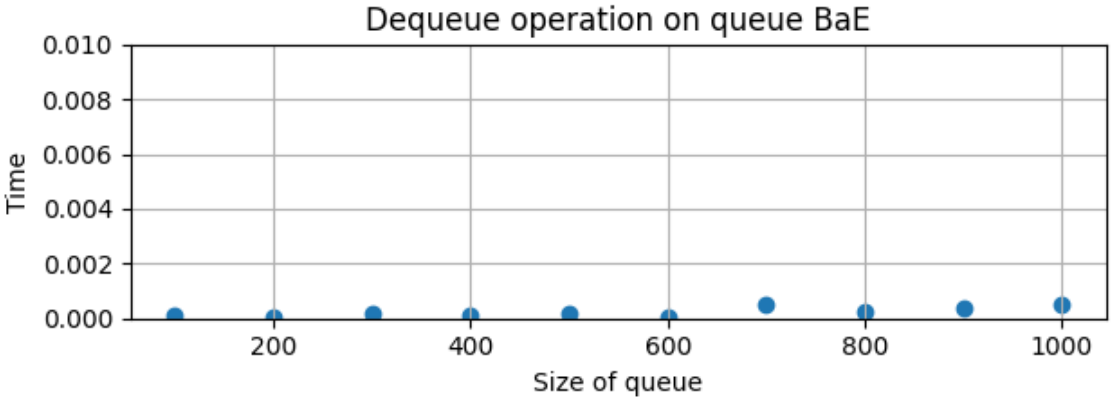



```
Size of queue: 1000
Time of enqueue operation for QueueBaB: 0.000249s
Time of enqueue operation for QueueBaE: 0.027502s
Time of dequeue operation for QueueBaB: 0.250077s
Time of dequeue operation for QueueBaE: 0.000497s
```



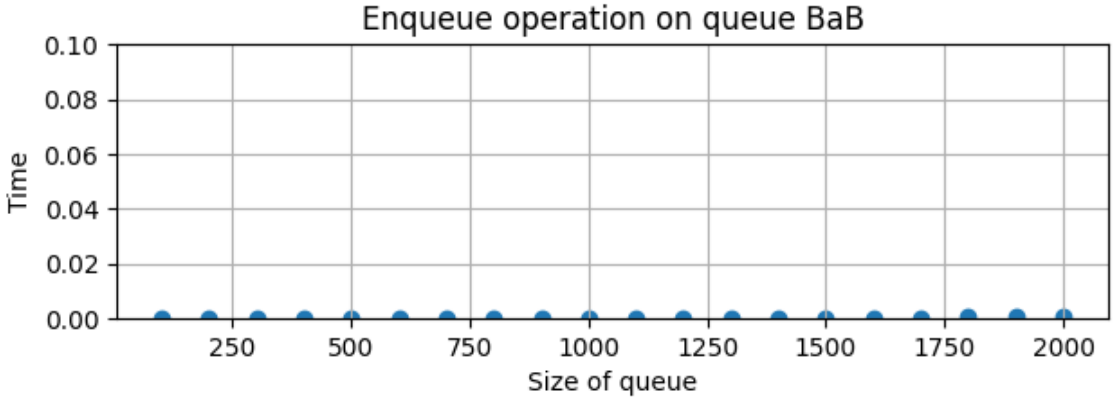


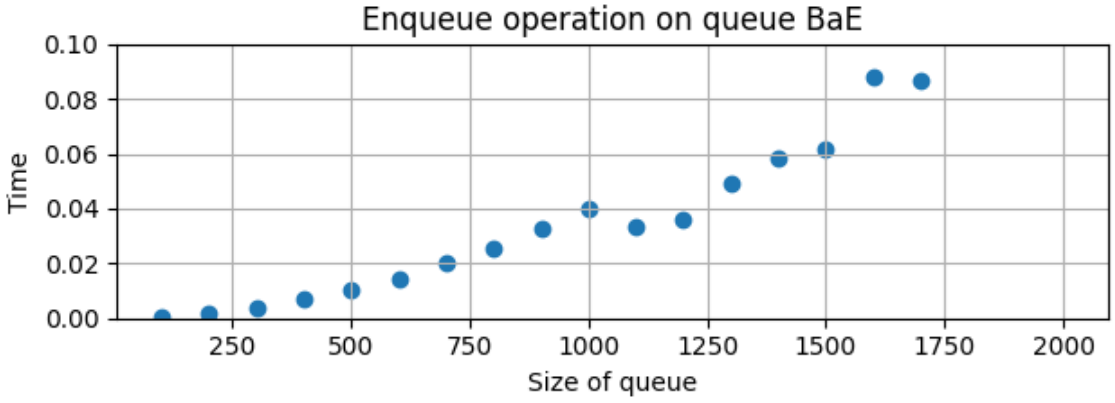


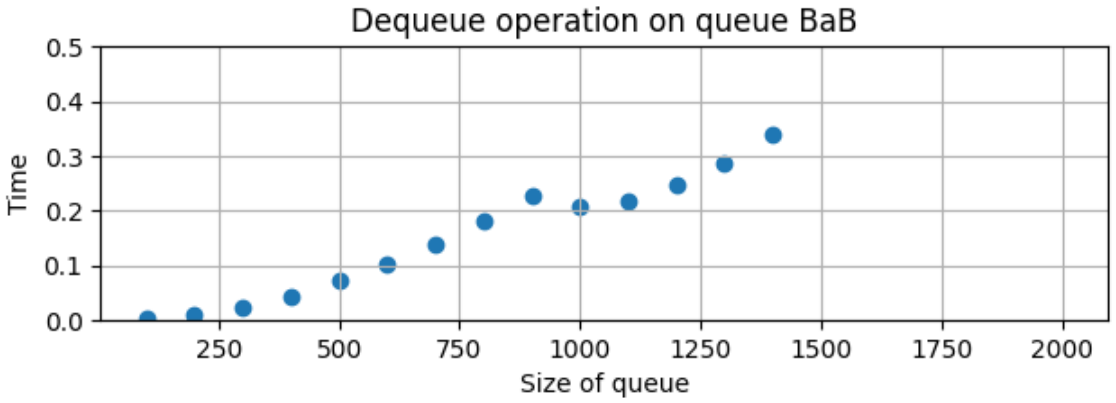


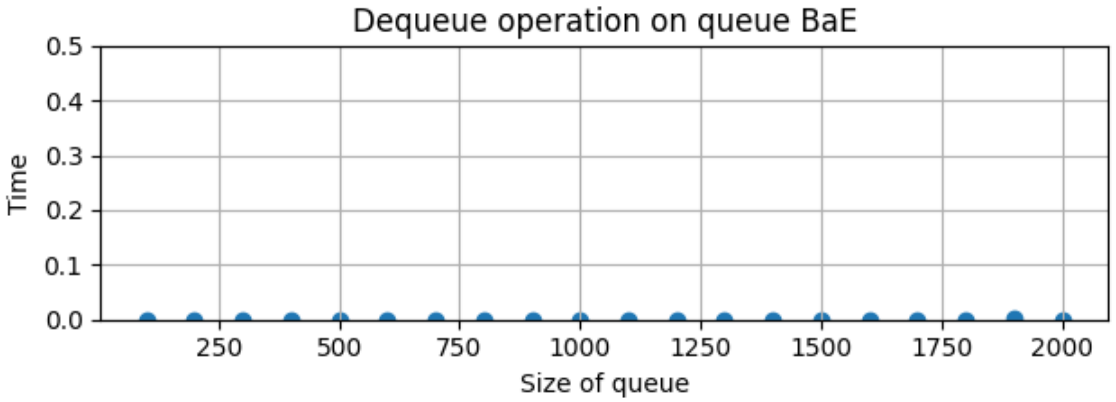
Dla coraz to dłuższych kolejek, czasy różnią się coraz bardziej. Tutaj przykład kolejki o rozmiarze 2000 elementów.

```
Size of queue: 2000
Time of enqueue operation for QueueBaB: 0.000395s
Time of enqueue operation for QueueBaE: 0.125936s
Time of dequeue operation for QueueBaB: 1.020931s
Time of dequeue operation for QueueBaE: 0.000886s
```









Widzimy, że dodawanie elementów do kolejki, wykona się szybciej w przypadku, gdy początek listy znajduje sie na początku. Analizując dokładniej, najpierw w kolejce znajduję się [1]. Potem dodajemy z jego prawej strony i mamy [1, 2]. Gdy mamy do czynienia z kolejką QueueBaE, to dodawanie wygląda inaczej. Najpierw w kolejce znajduję się [1]. Następnie po dodaniu 2, otrzymujemy [2, 1]. Dlatego też, wykonuje się ono dłużej. Zatem, operacja wstawiania elementów w przypadku QueueBaB będzie rzędu O(1), a w przypadku QueueBaE rzędu O(n).

Natomiast ściąganie elementu, wykona się sprawniej, gdy początek kolejki jest przechowywany na końcu - czyli dla QueueBaE. Ma to swoje uzasadnienie, gdyż w kolejkach, kto pierwszy się pojawił, pierwszy ją opuszcza. W przypadku QueueBaB, ściągając z niej pierwszy element - czyli początek listy - wpływa to na pozostałe, gdyż drugi staje się pierwszym, trzeci staje się drugim itd. Podsumowując, operacja usuwania elementów w przypadku QueueBaB będzie rzędu O(n), a w przypadku QueueBaE rzędu O(1).

Zadanie 3

W zadaniu 3 musieliśmy przeprowadzić symulację sytuacji z życia wziętej.

Opis naszego doświadczenia: Jesteśmy właścicielami spływu kajakowego. Mamy 10 jednoosobowych kajaków. Czas spływu znajduje się w przedziale 30-60 minut. Z przetransportowaniem kajaków z powrotem w górę rzeki, czekamy, aż spłyną wszystkie kajaki. Pomijamy czas potrzebny na wwiezienie kajaków na górę. Przychodzi 4 klientów na godzinę.

Pytanie badawcze: Jaki będzie średni czas oczekiwania na kajak oraz ilu klientów pozostanie nieobsłużonych po 8 godzinach pracy?

Opis kodu

W naszym programie zastosowaliśmy dwa złożone typy danych - kolejkę oraz stos. Ich implementacja była standardowa. Kolejkę będziemy wykorzystywać do opisu kolejki klientów, natomiast stos do opisu kajaków znajdujących się przy starcie oraz na mecie.

Klasa Kayak

Atrybuty:

- in_use - informuje, czy kajak jest aktualnie w użyciu
- down - informuje, czy kajak znajduje się na dole (mecie) trasy
- time_remaining - przechowuje informację przez ile sekund będzie jeszcze spływał kajak

Dodatkowo metoda tick() służy do zmniejszania atrybutu time_remaining wraz z biegiem czasu.

Klasa Client

Atrybuty:

- time_stamp - określa czas (liczony od początku symulacji), w którym klient został dodany do kolejki
- ride_time - losowa liczba z przedziału 1800-3600 (30-60 minut) określająca jak długo będzie spływał klient

Dodatkowo metoda wait_time() służy do obliczenia jak długo klient stał w kolejce.

Funkcja new_client()

Zakładamy, że przychodzi średnio 4 klientów na godzinę. Daje to jednego klienta na 900 sekund. Funkcja losuje liczbę z przedziału 1-900 i w przypadku wylosowania liczby 900, pozwala stworzyć nowego klienta.

Funkcja simulation(number_of_kayaks, time)

Funkcja, w której dzieje się cała symulacja.

- number_of_kayaks - liczba posiadanych kajaków, u nas jest to 10
- time - czas trwania symulacji, u nas 8 godzin czyli 28 800 sekund
- kayaks_up - stos z kajakami znajdującymi się na górze spływu, określany dalej jako stos górny
- kayaks_down - stos z kajakami znajdującymi się na dole spływu, określany dalej jako stos dolny
- clients_queue - kolejka z klientami oczekującymi na kajak
- kayaks_on_river - lista zawierająca kajaki, które w danym momencie są na rzece
- waiting_times - lista z czasem oczekiwania klientów na kajak

Na początku dodajemy wszystkie nasze kajaki do stosu kayaks_up. Następnie rozpoczynamy symulację iterując sekunda po sekundzie.

- sprawdzamy czy przyszedł nowy klient dzięki funkcji new_client()
- jeśli klient czeka w kolejce oraz na stosie górnym znajduje się dostępny kajak to klient rozpoczyna podróż, a czas oczekiwania oraz kajak dodajemy odpowiednio do list waiting_times oraz kayaks_on_river
- z każdą sekundą zmniejszamy czas podróży kajaków znajdujących się na rzece, a jeśli dopłyną one do końca to usuwamy je z listy kayaks_on_river oraz dodajemy na stos dolny
- jeśli na stosie dolnym znajdują się wszystkie nasze kajaki to transportujemy je wszystkie na górę i dodajemy do stosu górnego

Po zakończonej symulacji, dodajemy do listy waiting_times czas oczekiwania nieobsłużonych klientów. Ostatnim krokiem jest wyliczenie średniego czasu oczekiwania na kajak, korzystając z listy waiting_times.

Po obliczeniu średniej z 1000 symulacji otrzymaliśmy, że średni czas czekania na kajak to 5,88 minut.

Poniżej kilka zasymulowanych wyników.

In [2]:
```
for i in range(5):
    stimulation(10, 28800)
```

```
Average Wait   7.94 min,   0 clients remaining.
Average Wait   6.85 min,   0 clients remaining.
Average Wait   5.25 min,   0 clients remaining.
```

```
Average Wait   5.74 min,   0 clients remaining.
Average Wait   8.06 min,   0 clients remaining.
```

Zadanie 4

W zadaniu 4 musieliśmy wykorzystać stos do sprawdzenia, czy dokument HTML posiada wszystkie znaczniki zamykające. Stworzone funkcje:

- find_openers_closers(text) - funkcja znajduje znaczniki otwierające i zamykające w przekazanym stringu text. Sprawdzamy po kolei każdy znak w zmiennej text. W zależności od tego na jaki ciąg znaków trafimy, wstawiamy do listy openers_closers następujące elementy (... oznacza dowolny ciąg znaków): '</...>', '<! --', '<...' lub '<...>', '/>', '>', '-->'. Funkcja wywołuje funkcję add_endings(openers_closers).
- add_endings(tags_list) - funkcja ma za zadanie dodać brakujące zamknięcia do znaczników, które znajdują się na liście tags_list i nie zostały zamknięte np. '<..'. Iterujemy po kolejnych elementach listy i dodajemy zamknięcia '>' i '/>'. Zwracamy listę z zamkniętymi pełnymi znacznikami.
- checking_HTML_correctness(filename) - główna funkcja. Tworzy string z pliku, który zostaje przekazany jako filename. Stworzyliśmy listę singletons, która zawiera wszystkie znaczniki, które nie wymagają dodatkowego znacznika zamykającego i mogą się kończyć na '>' lub '/>'. Gdy mamy już listę openers_closers, wygenerowaną dzięki funkcji find_openers_closers, tworzymy stos.
  - Jeśli nasz znacznik należy do funkcji singletons to go pomijamy, ponieważ nie posiada on znacznika zamykającego.
  - Jeśli nie jest to znacznik zamykający to dodajemy go na stos.

Następnie w zależności od pojawiających sie znaczników zamykających, zdejmujemy elementy z naszego stosu. Jeśli wszystkie znaczniki były prawidłowo zamykane oraz stos na końcu został pusty to plik posiada prawidłowe znaczniki.

Zadanie 5

W tym zadaniu musieliśmy dodać brakujące metody do klasy UnorderedList.

- append(self, item) - metoda dołącza element na koniec naszej listy. Jeśli jest ona pusta, to dołączony element staje się pierwszym.
- index(self, item) - zwraca miejsce na liście, na którym znajduje się dany element. Jeśli elementu nie ma na liście, zwraca wartość None.
- insert(self, pos, item) - metoda wstawia element w dane miejsce. Przechodzimy po kolejnych elementach listy i dołączamy go we wskazane miejsce. Jeśli pozycja jest źle określona, zwracamy IndexError.
- pop(self, pos=-1) - metoda usuwa element z podanej pozycji. Na początku sprawdzamy szczególne wartości pos.
  - Jeśli wynosi ono size-1 to wtedy wiemy, że chodzi nam o ostatni element.
  - Jeśli długość listy to 0 to zwracamy IndexError.
  - Jeśli długość listy to 1 to pos może być równe tylko 1 albo -1.

Następnie obługujemy wartości pos dla -1 i 0. Dla innych wartości przechodzimy po liście i odłączamy niechciany element.

- Dodatkowo dopisaliśmy reprezentację teskstową _ *str* _ oraz metodę peek().

In [16]:
```
list = UnorderedList()
list.append("a")
list.append("b")
list.append("c")
list.append(10)
list.append(20)
print(list)
print("Is empty: "+str(list.is_empty()))
print("Size: "+str(list.size()))
print("Search item - a: "+str(list.search("a")))
```

```
print("Search item - d: "+str(list.search("d")))
print("Index number 10: "+str(list.index(10)))
print("Index number 15: "+str(list.index(15)))
print("Insert method: ")
list.insert(2,1000)
print(list)
print("Remove the last number: "+str(list.pop()))
print("Remove the first number: "+str(list.pop(0)))
print(list)
print("Is empty: "+str(list.is_empty()))
print("Size: "+str(list.size()))
```

```
Elements in the list are [a, b, c, 10, 20]
Is empty: False
Size: 5
Search item - a: True
Search item - d: False
Index number 10: 3
Index number 15: None
Insert method:
Elements in the list are [a, b, 1000, c, 10, 20]
Remove the last number: 20
Remove the first number: a
Elements in the list are [b, 1000, c, 10]
Is empty: False
Size: 4
```

In [5]:
```
list.insert(10, 2000)
print(list)
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-5-16e501d08806> in <module>
----> 1 list.insert(10, 2000)
      2 print(list)

<ipython-input-1-e248f91f0ddb> in insert(self, pos, item)
    110                     current = current.get_next()
    111                     if current == None and index < pos:
--> 112                         raise IndexError
    113                 index += 1
    114         else:

IndexError:
```

W naszej liście, nie mamy nic na 10 indeksie. W związku z tym, program zwraca błąd, gdy chcemy coś tam dodać.

In [6]:
```
for i in range(list.size()):
    print('-----')
    print("Remove item: "+str(list.pop()))
    print("Size: "+str(list.size()))
print("Is empty: "+str(list.is_empty()))
```

```
-----
Remove item: 10
Size: 3
-----
Remove item: c
Size: 2
-----
Remove item: 1000
Size: 1
-----
Remove item: b
Size: 0
Is empty: True
```

In [7]:
```
print("Remove item: "+str(list.pop()))
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-7-7fef3fc4a8ab> in <module>
----> 1 print("Remove item: "+str(list.pop()))
```

```
<ipython-input-1-e248f91f0ddb> in pop(self, pos)
    121         size = self.size()
    122         if size == 0:
--> 123             raise IndexError
    124         elif size == pos+1:
    125             pos = -1

IndexError:
```

Mamy pustą listę, zatem program zwraca błąd, gdy chcemy coś z niej usunąć.

Zadanie 6, 7

W zadaniach 6 i 7, musieliśmy zaimplementować stos oraz kolejkę dwukierunkową przy pomocy listy jednokierunkowej. Korzystając z zadania 5 stworzyliśmy klasę StackUsingUL() i DequeueUsingUL(). Oprócz obowiązkowych funkcji, dodaliśmy także reprezentację tekstową _ *str* _, która umożliwi nam podgląd, na to, co dokładnie znajduje się w naszych listach.

In [10]:
```
stack1 = StackUsingUL()
print(stack1)
print("Is empty: "+str(stack1.is_empty()))
print("Stack size: "+str(stack1.size()))
print("---Writing to the stack---")
stack1.push("pies")
stack1.push("kot")
stack1.push(1)
stack1.push(10)
stack1.push(0)
print(stack1)
print("Is empty: "+str(stack1.is_empty()))
print("The peek of the stack: "+str(stack1.peek()))
print("Stack size: "+str(stack1.size()))
print("---Writing out the stack---")
print("Remove item: " + str(stack1.pop()))
print("Remove item: " + str(stack1.pop()))
print(stack1)
print("Stack size: " + str(stack1.size()))
```

```
Elements in the list are []
Is empty: True
Stack size: 0
---Writing to the stack---
Elements in the list are [pies, kot, 1, 10, 0]
Is empty: False
The peek of the stack: 0
Stack size: 5
---Writing out the stack---
Remove item: 0
Remove item: 10
Elements in the list are [pies, kot, 1]
Stack size: 3
```

Nasza lista zawiera elementy [pies, kot, 1, 10, 0]. Jej rozmiar wynosi 5. Korzystając z peek(), widzimy, że ostatni element to 0. Ściągamy go przy pomocy pop(). Kolejny element to 10 - również go usuwamy. Nasz stos zmienia się. Rozmiar listy wynosi teraz 3.

In [12]:
```
stack2 = StackUsingUL()
stack2.push("wiosna")
stack2.push("lato")
stack2.push("algorytmy")
stack2.push(10)
stack2.push(20)
print(stack2)
print("Is empty: " +str(stack2.is_empty()))
print("Stack size: " +str(stack2.size()))
print(stack2.size())
for i in range(stack2.size()):
    print('-----')
    print("The peek of the stack: "+str(stack2.peek()))
    print("Remove item: "+str(stack2.pop()))
    print("Stack size: "+str(stack2.size()))
```

```
print('-----')
print("Is empty: " +str(stack2.is_empty()))
```

```
Elements in the list are [wiosna, lato, algorytmy, 10, 20]
Is empty: False
Stack size: 5
5
-----
The peek of the stack: 20
Remove item: 20
Stack size: 4
-----
The peek of the stack: 10
Remove item: 10
Stack size: 3
-----
The peek of the stack: algorytmy
Remove item: algorytmy
Stack size: 2
-----
The peek of the stack: lato
Remove item: lato
Stack size: 1
-----
The peek of the stack: wiosna
Remove item: wiosna
Stack size: 0
-----
Is empty: True
```

Nasza lista zawiera elementy [wiosna, lato, algorytmy, 10, 20]. Nie jest pusta, dlatego otrzymujemy False. Jej rozmiar wynosi 5. Ściągamy po kolei wszystkie elementy przy pomocy pop. Po ich usunięciu rozmiar wynosi 0 oraz lista jest pusta, stąd mamy True na końcu.

In [13]:
```
print("Remove number: "+str(stack2.pop()))
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-13-525f787d4270> in <module>
----> 1 print("Remove number: "+str(stack2.pop()))

<ipython-input-2-a2645611f126> in pop(self)
     14     def pop(self):
     15         if self.is_empty():
---> 16             raise IndexError('The stack is empty')
     17         else:
     18             return self.items.pop()

IndexError: The stack is empty
```

In [14]:
```
print("The peek of the stack: "+str(stack2.peek()))
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-14-dc8c9d5d5bba> in <module>
----> 1 print("The peek of the stack: "+str(stack2.peek()))

<ipython-input-2-a2645611f126> in peek(self)
     20     def peek(self):
     21         if self.is_empty():
---> 22             raise IndexError("The stack is empty")
     23         else:
     24             return self.items.peek()

IndexError: The stack is empty
```

Nasz stos jest pusty, dlatego program zwraca błąd - nie usuniemy nic z pustego stosu oraz nie znajdziemy wartości na wierzchu.

In [15]:
```
deque = DequeueUsingUL()
print("Is empty: " +str(deque.is_empty()))
print("Size: " +str(deque.size()))
print(deque)
print("---Writing to the deque---")
deque.add_left("a")
```

```
deque.add_left("b")
deque.add_right("algorytmy")
deque.add_right("struktury")
deque.add_left("c")
deque.add_right(1)
deque.add_right(2)
deque.add_left("d")
print(deque)
print("Is empty: " +str(deque.is_empty()))
print("Size: " +str(deque.size()))
```

```
Is empty: True
Size: 0
Elements in the list are []
---Writing to the deque---
Elements in the list are [d, c, b, a, algorytmy, struktury, 1, 2]
Is empty: False
Size: 8
```

Na początku kolejka jest pusta. Dodajemy do niej elementy. Teraz nasza kolejka dwustronna wygląda następująco [d, c, b, a, algorytmy, struktury, 1, 2]. Jej długość to 8 i nie jest pusta.

In [19]:
```
print("Remove item: "+str(deque.remove_left()))
print("Size: " +str(deque.size()))
```

```
Remove item: d
Size: 7
```

Ściągamy pierwszy element z lewej. Jest to d. Długość się zmniejsza.

In [20]:
```
for i in range(deque.size()):
    print("Remove item: "+str(deque.remove_right()))
print("Is empty: " +str(deque.is_empty()))
print("Size: " +str(deque.size()))
```

```
Remove item: 2
Remove item: 1
Remove item: struktury
Remove item: algorytmy
Remove item: a
Remove item: b
Remove item: c
Is empty: True
Size: 0
```

Usuwamy teraz wszystko po kolei, kończąc na pustej liście.

In [21]:
```
print(deque.remove_left())
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-21-777eaf42cd26> in <module>
----> 1 print(deque.remove_left())

<ipython-input-16-5421c4a4b176> in remove_left(self)
     38         """
     39         if self.is_empty():
---> 40             raise IndexError("The queue is empty")
     41         else:
     42             return self.items.pop(0)

IndexError: The queue is empty
```

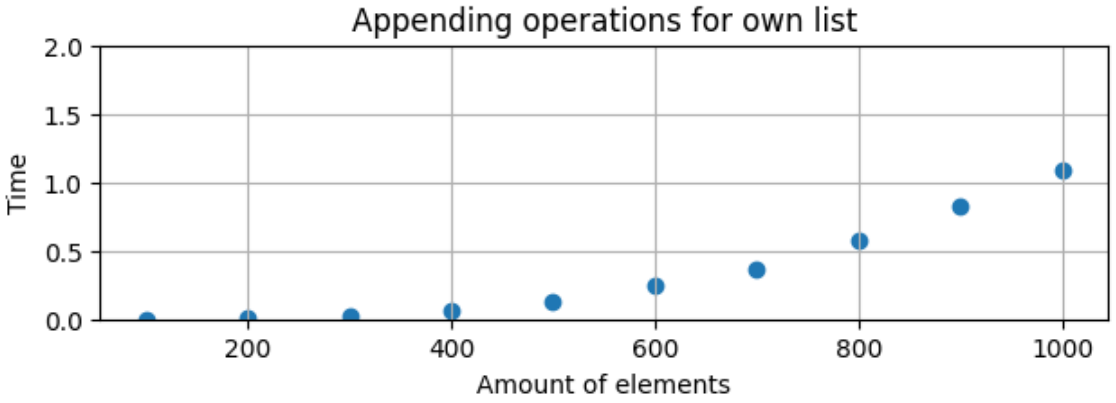
Program zwraca błąd. Nie da się usunąć nic z pustej listy.

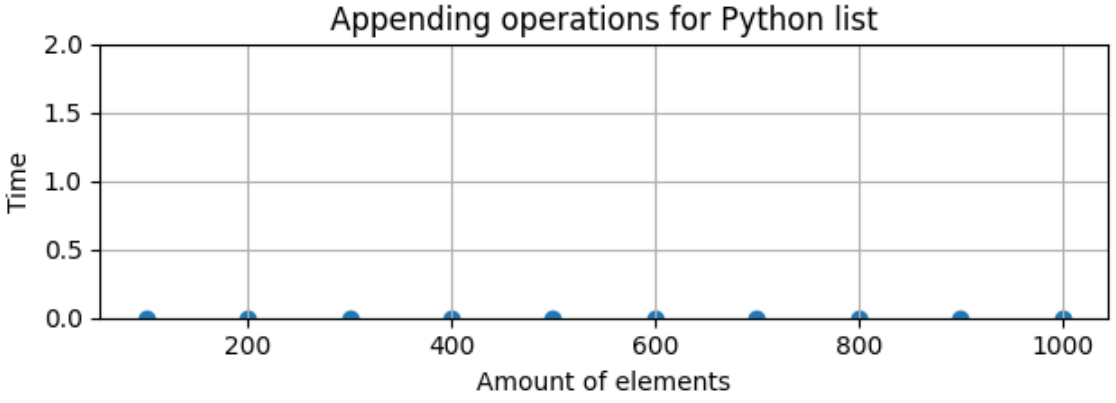
Zadanie 8

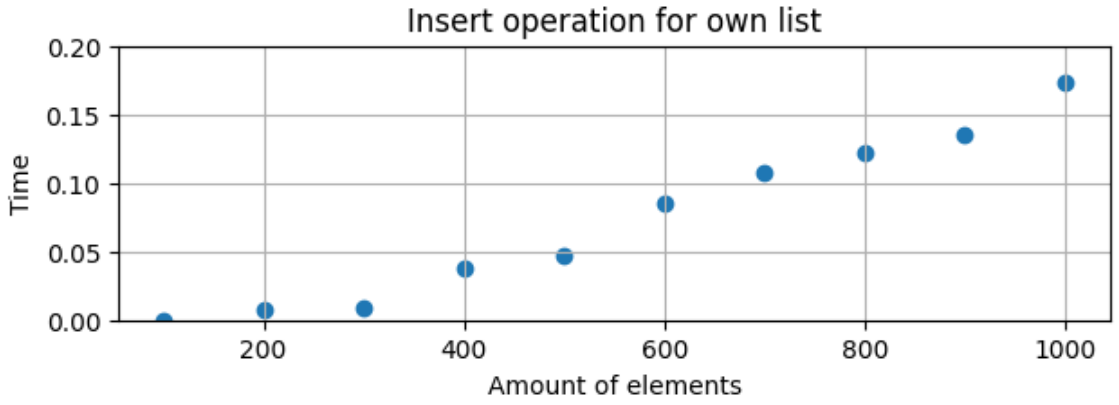
W ostatnim zadaniu, naszym celem było zaprojektowanie eksperymentu porównującego wydajność listy jednokierunkowej i listy wbudowanej w Pythona. Stworzyliśmy więc funkcję compare_lists(), która jest odpowiedzialna za liczenie czasów dla kolejnych operacji i metod, takich jak: tworzenie listy, dodawanie elementów, dodawanie elementów w określone miejsca, znajdowanie indeksów, mierzenie rozmiaru oraz usuwanie elementów.

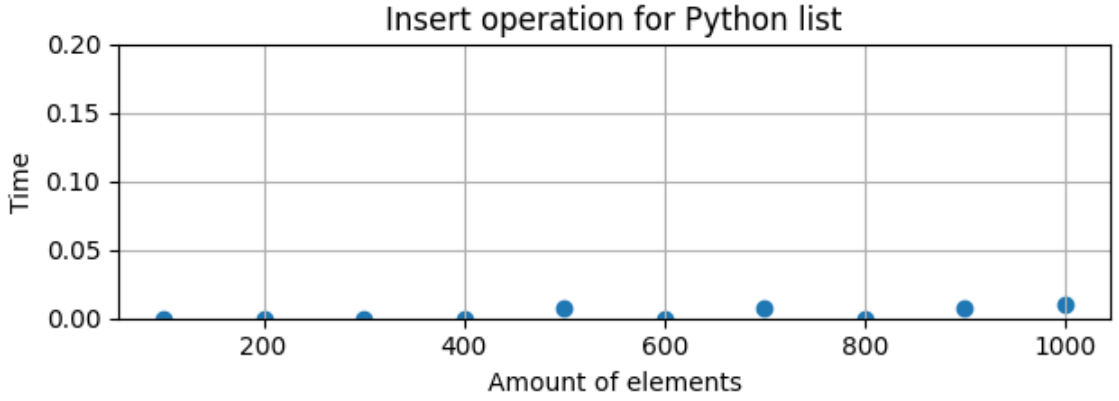
```
Amount of elements: 1000
Time of creating own list: 0.000000s
Time of creating Python list: 0.000000s
----
Time of appending data for own list: 1.101968s
Time of appending data for Python list: 0.000000s
----
Time of insert operation  for own list: 0.174715s
Time of insert operation for Python list: 0.009919s
----
Search time for the own index method: 0.000000s
Search time for the Python index method: 0.000000s
----
Search time for the own size method: 0.000000s
Search time for the Python size method: 0.000000s
----
Time of erase of data from own list: 3.652925s
Time of erase of data from Python list: 0.000000s
```

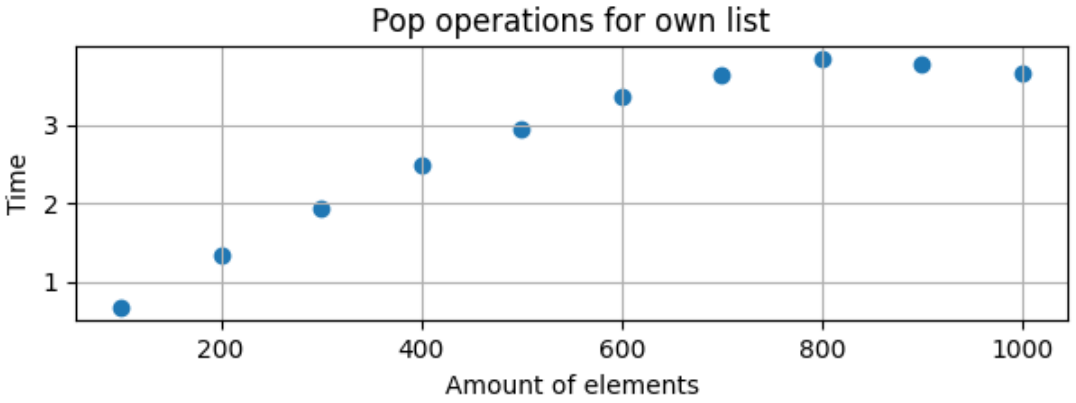
Dodatkowo stworzyliśmy również wykresy porównujące szybkość działania dla append(), insert() oraz pop().

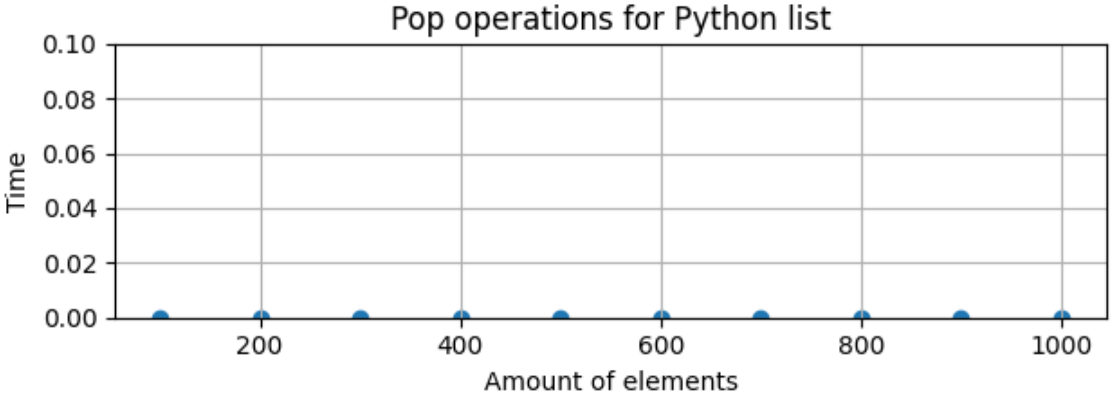












Po eksperymentach łatwo zauważyć, że listy w Pythonie są zdecydowanie szybsze we wszystkich operacjach na nich wykonywanych, niż nasza własna lista.

Linki

https://github.com/github-kamilk/AiSD/tree/main/Lista_4