Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Krystian Gaj

Nr albumu: 1147878

Implementacja list z przeskokami w języku Python

Praca licencjacka na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Informatyki Stosowanej

Kraków 2020

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Kraków, dnia Podpis autora pracy

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Kraków, dnia Podpis kierującego pracą

Składam podziękowania wszystkim,
którzy udzielili mi wsparcia
w trakcie tworzenia tej pracy,
a w szczególności
dr hab. Andrzejowi Kapanowskiemu
za nieocenioną pomoc w postaci
rad, wskazówek oraz cierpliwości i wyrozumiałości.

Streszczenie

Celem pracy było przedstawienie oraz implementacja w języku Python list z przeskokami. Praca została podzielona na trzy części. Pierwszą z nich jest wprowadzenie, gdzie przedstawiono motywację stworzenia opisywanej struktury danych oraz podstawowy teorii z nią związanej. Druga część została poświęcona implementacji list z przeskokami wraz z omówieniem wykorzystanych algorytmów. Część ta zawiera opis zarówno dla użytkowników, jak i dla osób zainteresowanych sposobem działania poszczególnych metod. Trzecią część stanowi tematyka mediany kroczącej w kontekście praktycznego zastosowania list z przeskokami. Składa się ona z wprowadzenia w postaci podstawowych definicji oraz omówienia algorytmów. Podobnie jak w przypadku części drugiej, tutaj także opis metod podzielony został na instrukcję użytkownika oraz dokładne zapoznanie z algorytmami. W dodatkach przedstawiony został kod źródłowy poszczególnych modułów powstałych na potrzeby pracy, a także praktyczny przykład ich zastosowania.

Słowa kluczowe: lista z przeskokami, lista powiązana, Python, mediana krocząca

English title: Python implementation of skip lists

Abstract

The purpose of the bachelor thesis is to present skip lists and implement them in Python language. The thesis has been divided into three parts. The first part is an introduction, where motivation of creating skip lists has been depicted and basic theoretical knowledge has been elaborated. The second part is about skip lists implementation and interface. That part contains end user's guide and algorithms description for people looking for better understanding. The third part presents running median calculations as a practical use of skip lists. It consists of theoretical introduction, user's guide and algorithms' deep characterisation. Appendices contain source code of modules created for this thesis and an example of their practical use.

Keywords: skip list, linked list, Python, running median

Spis treści

Spis tabel

Spis rysunków

1. Wstęp

Niniejsza praca ma za zadanie przybliżenie tematyki list z przeskokami osobom, które nie są zaznajomione z tą strukturą danych, a które poszukują alternatywy prostszej w implementacji od drzew zrównoważonych oraz zapewniającej lepszą oczekiwaną złożoność obliczeniową niż klasyczne listy powiązane.

1.1. Cel pracy

Celem niniejszej pracy jest stworzenie implementacji list z przeskokami w języku Python oraz przedstawienie ich praktycznego zastosowania na przykładzie obliczania mediany kroczącej. W kolejnych rozdziałach przedstawione zostaną założenia teoretyczne w celu sformułowania zagadnienia związanego z tematyką pracy, przykładowa sesja interaktywna list z przeskokami wraz z instrukcją użytkownika, oraz jako przykład zastosowania praktycznego sesja interaktywna modułu mediany kroczącej, by przedstawić czytelnikowi sposób pracy z interfejsami tych modułów i pozwolić mu naocznie ujrzeć efekty działania, dokładne omówienie zastosowanych algorytmów w celu przedstawienia mechanizmów skrywających się za interfejsem, a także porównanie wydajności list z przeskokami oraz innych struktur danych, by wskazać wady i zalety poszczególnych rozwiązań. Pozwoli to na wskazanie miejsc, w których listy z przeskokami mają przewagę.

1.2. Struktura

Praca podzielona została na trzy części.

Część pierwszą stanowi wprowadzenie w tematykę list z przeskokami. Przedstawione zostaną klasyczne listy powiązane, ich podstawowe algorytmy, a także zalety i wady. Następnie opisane zostaną listy z przeskokami, ich podstawowe założenia oraz mocne i słabe strony tej struktury danych.

Drugą część rozpoczyna instrukcja użytkownika. Przedstawione zostają tam sposoby wykorzystania interfejsu stworzonej implementacji list z przeskokami. Zawarta jest tam jedynie wiedza podstawowa, niezbędna do używania tej struktury danych. Następnie przedstawiona zostanie przykładowa sesja interaktywna Pythona, której celem jest lepsze zobrazowanie wcześniej omówionych zastosowań.

Część drugą kończy szczegółowe przedstawienie każdej z metod interfejsu. Opisane zostaną kryjące się za nimi algorytmy, co stanowi rozszerzenie informacji z części pierwszej.

W ostatniej części przedstawiona zostanie problematyka wyznaczania mediany kroczącej, a także interfejs modułu do tego przeznaczonego. Podobnie, jak w części poprzedniej, omówienie implementacji podzielone zostało na instrukcję użytkownika wraz z przykładową sesją interaktywną Pythona oraz szczegółowy opis algorytmów.

W dodatkach zamieszczone zostały wyniki testów algorytmów list z przeskokami oraz mediany kroczącej. Zamieszczony został także praktyczny przykład wykorzystania mediany kroczącej opartej o listy z przeskokami - wyznaczanie mediany hurtowych cen benzyny dla ciągle zmieniających się danych. Umieszczony został także kod opisywanych w pracy modułów.

2. Listy powiązane

W tym rozdziale przedstawione zostaną trzy rodzaje list powiązanych. Począwszy od list jednokierunkowych, przez listy dwukierunkowe, aż po listy z przeskokami przedstawione zostaną podstawowe twierdzenia związane z każdą z tych struktur danych, motywacja ich powstania. Omówione zostaną także podstawowe operacje każdej z nich wraz ze złożonością obliczeniową.

2.1. Definicja

Lista jest to struktura danych umożliwiająca przechowywanie zawartości w porządku liniowym [5]. Składa się ona z węzłów przechowujących wartość oraz odnośnik (lub odnośniki) do innych elementów listy [6].

2.2. Listy jednokierunkowe

Najprostszą wersją listy jest lista pojedynczo powiązana, zwana także jednokierunkową. Węzeł oprócz wartości przechowuje także odnośnik do swojego następnika. Poniżej przedstawione zostały podstawowe algorytmy.

2.2.1. Wstawianie

Dodawanie nowego elementu przed wartością znajdującą się już na liście odbywa się następująco [8]:

1. Począwszy od głowy listy, przejdź do elementu poprzedzającego miejsce umieszczenia nowego węzła.
2. Jeśli taki węzeł nie został odnaleziony, zwróć błąd. W przeciwnym wypadku przejdź do kroku następnego.
3. Utwórz nowy węzeł, przypisz mu żądaną wartość oraz jako następnik ustaw następny element po znalezionym w kroku 1.
4. Ustaw nowy węzeł jako następny po elemencie z kroku 1.

Złożoność tej operacji wynosi $O(n)$. W przypadku, gdy pozycja nowego elementu jest już znana, złożoność również wynosi $O(n)$.

2.2.2. Usuwanie

Usuwanie elementu o podanej wartości odbywa się w następujący sposób[8]:

1. Przejdź do elementu poprzedzającego usuwany.
2. Jeśli taki węzeł nie został odnaleziony, zwróć błąd. W przeciwnym wypadku przejdź do kroku następnego.
3. Jako następnik tego elementu ustaw węzeł następny po usuwanym.
4. Zwolnij pamięć po usuwanym węźle.

Złożoność tej operacji jest rzędu $O(n)$, niezależnie od tego, czy pozycja usuwanego elementu jest już znana, czy nie.

2.2.3. Wyszukiwanie

Wyszukiwanie elementu o żądanej wartości można opisać następującym algorytmem [8]:
1. Utwórz odnośnik do głowy listy.
2. Dopóki wskazywany element jest różny od różny od NIL oraz jego wartość jest inna niż poszukiwana, przejdź do następnego węzła.
3. Jeśli węzeł o żądanej wartości został znaleziony, zwróć jego wartość. W przeciwnym wypadku zwróć błąd.

Złożoność tego algorytmu jest rzędu $O(n)$.

2.3. Listy dwukierunkowe

Zasadniczą różnicą między listami podwójnie powiązanymi a listami jednokierunkowymi jest obecność dodatkowego odnośnika wewnątrz każdego z węzłów. Wskazuje on na element poprzedni.

Dzięki wprowadzeniu dodatkowego łącza poprawie uległa złożoność obliczeniowa operacji dodawania oraz usuwania w przypadku, gdy znana jest pozycja węzła, na którym wykonywana jest operacja. Jest ona rzędu $O(1)$.

2.4. Zalety oraz wady powyższych wersji

Poniżej przedstawione zostaną główne argumenty przemawiające za oraz przeciwko listom powiązanym pojedynczo oraz podwójnie.

Zalety:
- Łatwa implementacja
- Złożoność pamięciowa rzędu $O(n)$ [9][10]

Wady:
- Oczekiwana złożoność obliczeniowa operacji wstawiania (listy posortowane oraz indeksowane), usuwania oraz wyszukiwania rzędu $O(n)$ [9][10]
- Utrzymanie uporządkowania zawartości wymaga sortowania po każdym wstawieniu nowego elementu, modyfikacji algorytmu dodawania nowych elementów lub specjalnej obsługi tego algorytmu z zewnątrz

2.5. Listy z przeskokami

Z powodu wad wymienionych w poprzednim podrozdziale listy nie są pierwszym wyborem, gdy zachodzi konieczność pracy z dużymi zbiorami danych, gdy zachodzi potrzeba szybkiego, wielokrotnego dostępu do nich oraz gdy muszą być posortowane. W takich zastosowaniach znacznie lepiej sprawdzają się drzewa zrównoważone, takie jak drzewa AVL czy drzewa

czerwono-czarne. Ich wadą jednak jest duży stopień komplikacji, co utrudnia prawidłową implementację.

W 1989 roku William Pugh przedstawił strukturę danych, nazywaną listą z przeskokami (ang. *skip list*) [2], która łączy zalety list oraz drzew zbalansowanych. Jest zarówno prosta w implementacji, jak i posiada operacje dodawania, usuwania oraz wyszukiwania o oczekiwanej złożoności rzędu $O(\log_2 n)$.

Poniżej przedstawione zostały podstawowe definicje oraz założenia dotyczące list z przeskokami. Szczegółowe omówienie algorytmów znajduje się w rozdziale 4.

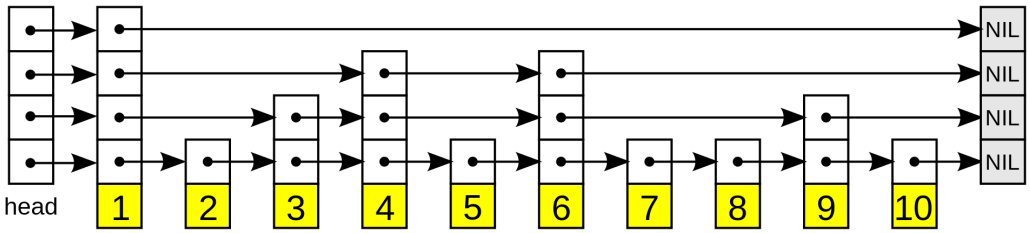
2.5.1. Definicje

Lista z przeskokami jest to probabilistyczna struktura danych oparta na listach [4]. Powstała jako prostsza w implementacji alternatywa dla drzew zrównoważonych. Utrzymanie zbalansowania odbywa się z pomocą generatora liczb pseudolosowych [3]. Główną różnicą w stosunku do klasycznych list jest możliwość posiadania przez pojedynczy węzeł więcej niż jednego odnośnika do elementów następnych. Oznacza to, że operacje na listach z przeskokami nie wymagają sekwencyjnego przechodzenia przez kolejne węzły.

Sytuacje, by lista z przeskokami nie była zrównoważona, są bardzo rzadkie. "Przykładowo, dla słownika zawierającego więcej niż 250 elementów szansa, by wyszukiwanie trwało dłużej niż 3-krotność oczekiwanego czasu jest mniejsza niż jeden na milion" [3].

Poziom węzła - jest to liczba odnośników węźle do następnych elementów listy.

Poziom listy - jest to poziom węzła, który spośród wszystkich elementów listy posiada największą liczbę odnośników do swoich następników. W implementacji przedstawionej w tej pracy pojęcie to równoznaczne jest liczbie o jeden mniejszej niż liczba odnośników. Przyczyną tego jest zliczanie poziomów od zera, a nie jak w przypadku Pugha od 1.



Rysunek 2.1: Lista z przeskokami o 4 poziomach, z wartownikiem. Przykład pochodzi ze strony [2].

2.5.2. Przydzielanie poziomów

Poziomy przydzielane są według następującego wzorca: 50% węzłów posiada poziom pierwszy, 25% drugi, 12.5% trzeci, itd [3]. Węzły o poziomie i stanowią więc $1/2^i$ całości.

Rozmieszczenie węzłów w taki sposób, by każdy węzeł o indeksie 2^i posiadał odnośnik do węzła oddalonego o 2^i zapewniałoby bardzo szybkie wyszukiwanie, jednak znacznie skomplikowałoby operacje dodawania oraz usuwania elementów listy. Stąd też nowe poziomy nowych węzłów przydzielane są z użyciem generatora liczb pseudolosowych, lecz z zachowaniem schematu przedstawionego w poprzednim akapicie. Odnośnik na i-tym poziomie nie wskazuje na węzeł oddalony o ściśle określoną liczbę skoków, lecz na węzeł o tym samym lub wyższym poziomie. Dzięki temu podczas dodawania czy usuwania korekcja poziomu pozostałych elementów listy jest zbędna [3].

2.5.3. Maksymalny poziom listy

Maksymalny poziom listy determinowany jest przez liczbę elementów w nim zawartych. Wynosi ona $\log_2 n$, gdzie n u Pugha [3] oznacza maksymalną dozwoloną liczbę węzłów, natomiast u Hettingera [7] ta liczba jest wielkością przewidywaną.

W implementacji przedstawionej w niniejszej pracy maksymalny poziom listy uzależniony jest od aktualnej liczby węzłów i obliczany jest po każdym dodaniu nowego elementu.

2.5.4. Zalety i wady

Zalety:

- Łatwa implementacja - prostsza niż w przypadku drzew zrównoważonych
  - Utrzymanie zrównoważenia z wykorzystaniem probabilistyki jest łatwiejsze niż definiowanie wprost metod wykonujących to w określony sposób [3]
- Oczekiwana złożoność obliczeniowa operacji wstawiania, usuwania oraz wyszukiwania rzędu $O(\log_2 n)$ [10]

Wady:

- Pesymistyczna złożoność obliczeniowa operacji wstawiania, usuwania oraz wyszukiwania rzędu $O(n)$ [10]
- Złożoność pamięciowa rzędu $O(n \log_2 n)$ [10]

2.6. Indeksowanie

Wszystkie struktury przedstawione w tym rozdziale mogą zostać przystosowane do wykonywania operacji na podstawie nie wartości, lecz pozycji elementu w liście. Modyfikacja głównie sprowadza się do zastąpienia w algorytmach wartości indeksami oraz dodania zliczania wykonywanych przejść między węzłami. Jako przykład posłużyć mogą algorytmy przedstawione w rozdziale 4.

3. Implementacja

Ten podrozdział poświęcony został interfejsowi modułu SkipList z punktu widzenia użytkownika. W przystępny sposób przedstawione zostaną poszczególne operacje dostępne dla tej implementacji oraz zasady ich użytkowania.

3.1. Importowanie

Import klasy SkipList możliwy jest na kilka sposobów. Zalecanym ze względu na prostotę późniejszego użycia jest:

```
>>> from SkipList import SkipList
```

lub krócej:

```
>>> from SkipList import *
```

3.2. Tworzenie nowej listy

```
>>> nazwa_listy = SkipList()
```

3.3. Dodawanie nowego elementu do listy

Do wstawiania nowego elementu do listy służy metoda SkipList.insert(). Przyjmuje ona jeden argument, którym jest dodawana do listy wartość. Nowy element musi być porównywalny z dotychczasowymi elementami listy, tj. mieć operatory porównania dla typów reprezentowanych przez zawartość listy. Przykładowe użycie:

```
>>> nazwa_listy.insert(5)
```

3.4. Usuwanie elementu z listy

Usunięcie elementu z listy możliwe jest na dwa sposoby. Pierwszym z nich jest podanie wartości usuwanego węzła, natomiast drugim podanie jego miejsca na liście (indeksu).

Usuwanie węzła poprzez wartość odbywa się z użyciem metody SkipList.remove(). Jej argumentem jest wartość elementu, który ma zostać usunięty. Przykładowe użycie:

```
>>> nazwa_listy.remove(20)
```

Jeśli dana wartość nie występuje w liście, zwrócony zostanie wyjątek ValueError. Usuwaniu elementu listy poprzez indeks służy słowo kluczowe **del**. Przykładowe użycie:

```
>>> del nazwa_listy[13]
```

Możliwe jest także używanie indeksów ujemnych. Przykładowo, usunięcie ostatniego elementu listy wygląda następująco:

```
>>> del nazwa_listy[-1]
```

natomiast czwartego od końca:

```
>>> del nazwa_listy[-4]
```

W przypadku, gdy podany indeks jest większy niż rozmiar listy pomniejszony o jeden lub mniejszy niż liczba przeciwna do wielkości listy, zwrócony zostanie wyjątek IndexError.

3.5. Wyszukiwanie najmniejszego oraz największego elementu listy

Interfejs klasy SkipList udostępnia narzędzia pozwalające w prosty, czytelny sposób odnaleźć najmniejszą oraz największą wartość w liście. Użytkownik może uzyskać najmniejszą wartość listy poprzez metodę SkipList.find_min(). Nie przyjmuje ona żadnych argumentów. Przykładowe użycie:

```
>>> L = SkipList()
>>> for item in range(10, 25):
...     L.insert(item)
...
>>> L.find_min()
10
```

Metoda zwraca najmniejszą wartość przechowywaną w liście. W przypadku, gdy lista jest pusta, zwrócony zostanie wyjątek IndexError.

Analogicznie do poprzedniego przykładu, użytkownik może znaleźć największy element listy poprzez metodę SkipList.find_max(). Nie przyjmuje ona żadnych argumentów. Przykładowe użycie:

```
>>> L = SkipList()
>>> for item in range(10, 25):
...     L.insert(item)
...
>>> L.find_max()
24
```

3.6. Długość listy

Do sprawdzenia długości listy, tj. liczby zawartej w niej elementów służy słowo kluczowe **len**. Przykład użycia:

```
>>> L = SkipList()
>>> for item in range(10, 25):
...     L.insert(item)
...
>>> len(L)
15
```

3.7. Sprawdzenie, czy lista jest pusta

Aby sprawdzić, czy lista jest pusta, wystarczy użyć funkcji **bool**, jak na poniższym przykładzie:

```
>>> L = SkipList()
>>> bool(L)
False
>>> L.insert(7)
>>> bool(L)
True
```

Metoda ta zwraca True, gdy lista zawiera przynajmniej jeden element lub False, jeśli lista jest pusta. Można zastosować tę metodę w instrukcji **if** lub **while**:

```
>>> if(L):
...     print(2+2)
...
4
>>> while(L):
...     del L[0]
...
>>> bool(L)
False
>>> len(L)
0
>>>
```

3.8. Indeks elementu o podanej wartości

Klasa SkipList umożliwia także sprawdzenie, w którym miejscu na liście znajduje się podana wartość. W tym celu należy użyć metody SkipList.index(), która jako argument przyjmuje wartość dla której indeks jest poszukiwany. Przykład użycia:

```
>>> L = SkipList()
>>> for item in range(25):
...     L.insert(item)
...
```

```
>>> L.index(7)
7
>>> L.remove(2)
>>> L.index(7)
6
```

Metoda zwraca indeks, pod którym znajduje się pierwsze wystąpienie podanej wartości. W przypadku, gdy dana wartość nie należy do żadnego elementu listy, zwracany jest wyjątek ValueError.

3.9. Generatory

Obiekty klasy SkipList mogą zwracać generatory, które przechodzą listę po jej najniższym poziomie. Dzięki temu możliwe jest korzystanie z iteratorów.

```
>>> for item in L:
...     print(item)
...
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
>>> print([item for item in L])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
>>> L_iter = iter(L)
>>> print(next(L_iter))
0
>>> print(next(L_iter))
1
>>> print(next(L_iter))
2
>>> print(next(L_iter))
3
```

3.10. Przynależność elementu do listy

Możliwe jest proste sprawdzenie, czy element należy do listy. Służy do tego słowo kluczowe **in**. Jego działanie obrazuje poniższy przykład:

```
>>> L = SkipList()
>>> for item in range(5):
...     L.insert(item)
```

```
...
>>> 1 in L
True
>>> 8 in L
False
```

Jako wynik zwracana jest wartość logiczna True, jeśli element należy do listy, False w przeciwnym wypadku.

3.11. Wyszukiwanie elementu na podstawie jego indeksu

W przypadku potrzeby odnalezienia elementu o określonej pozycji w liście pomocny jest operator [ ]. Jako argument przyjmuje, podobnie jak w przypadku słowa kluczowego **del** zarówno argumenty w postaci liczb dodatnich, jak i ujemnych.

```
>>> L = SkipList()
>>> for item in range(5):
...     L.insert(item)
...
>>> L[2]
2
>>> L[-1]
4
>>> L[4]
4
```

3.12. Aktualna struktura listy

Użytkownik może uzyskać informację o aktualnej strukturze listy w formie łańcucha znakowego dzięki funkcji **str**(). Może ona zostać wypisana lub użyta w innym celu. Przykładowe zastosowanie:

```
>>> print(L)
Level 3: [8]
Level 2: [4, 8, 13]
Level 1: [4, 6, 8, 9, 12, 13]
Level 0: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

>>> L_as_str = str(L)
>>> print(L_as_str)
Level 3: [8]
Level 2: [4, 8, 13]
Level 1: [4, 6, 8, 9, 12, 13]
Level 0: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

3.13. Przykładowa sesja interaktywna

```
>>> from SkipList import *
>>> import random
>>>
>>> L = [x for x in range(20)]
>>> random.shuffle(L)
>>> print(L)
[17, 3, 11, 16, 0, 2, 10, 18, 1, 14, 7, 15, 9, 6, 4, 5, 12, 8, 19, 13]
>>> SL = SkipList()
>>> for item in L:
...     SL.insert(item)
...
>>> print(SL)
Level 4: [19]
Level 3: [7, 19]
Level 2: [7, 8, 9, 19]
Level 1: [6, 7, 8, 9, 10, 14, 15, 19]
Level 0: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> for elem in SL:
...     print(elem)
...
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
>>>
>>> SL.remove(5)
>>> SL.remove(5)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "D:\Dokumenty\Studia_FAIS\Licencjat\lic-gaj\src\median\SkipList.py", line 195
raise ValueError("Value not found")
ValueError: Value not found
>>> del SL[2]
>>> print(SL)
Level 4: [19]
Level 3: [7, 19]
Level 2: [7, 8, 9, 19]
Level 1: [6, 7, 8, 9, 10, 14, 15, 19]
Level 0: [0, 1, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> SL.insert(20)
>>> print(SL)
```

```
Level 4: [19]
Level 3: [7, 19]
Level 2: [7, 8, 9, 19]
Level 1: [6, 7, 8, 9, 10, 14, 15, 19, 20]
Level 0: [0, 1, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
>>> bool(SL)
True
>>> print([item for item in SL])
[0, 1, 3, 4, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
>>> print(len(SL))
19
>>> for i in range(1, len(SL) + 1):
...     print(SL[-i])
...
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
4
3
1
0
>>> print(SL.index(20))
18
>>> print(SL.find_min())
0
>>> print(SL.find_max())
20
>>> del SL[-3]
>>> del SL[7]
>>> SL.remove(1)
>>> print(SL)
Level 4: [19]
Level 3: [7, 19]
Level 2: [7, 8, 19]
Level 1: [6, 7, 8, 10, 14, 15, 19, 20]
Level 0: [0, 3, 4, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 19, 20]
>>> del SL[-2]
>>> print(SL)
Level 3: [7]
Level 2: [7, 8]
Level 1: [6, 7, 8, 10, 14, 15, 20]
Level 0: [0, 3, 4, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 20]
>>> while bool(SL):
...     del SL[0]
...
```

```
>>> print(SL)
Level 0:]
>>> bool(SL)
False
>>> SL.insert(1)
>>> SL.insert(2)
>>> SL.insert(2)
>>> SL.insert(2)
>>> print(SL)
Level 2: [1, 2]
Level 1: [1, 2]
Level 0: [1, 2, 2, 2]
>>> SL.insert(0)
>>> print(SL)
Level 2: [1, 2]
Level 1: [1, 2]
Level 0: [0, 1, 2, 2, 2]
>>> SL.insert(27)
>>> print(SL)
Level 2: [1, 2]
Level 1: [1, 2, 27]
Level 0: [0, 1, 2, 2, 2, 27]
>>> for i in range(len(SL)):
...     print(SL[i])
...
0
1
2
2
2
27
>>> SL[3] = 5
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: __setitem__
>>> print(SL[-(len(SL) + 1)])
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "D:\Dokumenty\Studia_FAIS\Licencjat\lic-gaj\src\median\SkipList.py", line 76,
self.__test_index(index)
File "D:\Dokumenty\Studia_FAIS\Licencjat\lic-gaj\src\median\SkipList.py", line 121
raise IndexError("Index out of range")
IndexError: Index out of range
>>> print(SL[len(SL) + 1])
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "D:\Dokumenty\Studia_FAIS\Licencjat\lic-gaj\src\median\SkipList.py", line 76,
self.__test_index(index)
File "D:\Dokumenty\Studia_FAIS\Licencjat\lic-gaj\src\median\SkipList.py", line 121
raise IndexError("Index out of range")
IndexError: Index out of range
>>>
```

4. Algorytmy

Niniejszy rozdział jest rozwinięciem poprzedniego. W rozdziale 3 przedstawiono informacje o użyciu interfejsu modułu SkipList, natomiast poniżej przedstawione zostaną szczegóły działania poszczególnych metod. Pełny kod źródłowy modułu SkipList znajduje się w dodatku E.

4.1. Klasa SkipList.__Node

Klasa __Node reprezentuje węzeł listy. Znajduje się ona wewnątrz klasy SkipList i jest prywatna. Z punktu widzenia użytkownika jest całkowicie niewidoczna - operuje on na wartościach przechowywanych wewnątrz węzłów.

Klasa posiada konstruktor __init__ oraz typową metodę __repr__ do tworzenia reprezentacji napisowej.

4.1.1. Konstruktor __init__()

Dane wejściowe:
value - wartość przechowywana wewnątrz węzła,
level - poziom węzła.

Problem: Utworzenie nowej instancji klasy SkipList.__Node.

Opis algorytmu: Konstruktor na podstawie otrzymanych danych tworzy atrybuty przechowujące wartość oraz informacje o poziomie węzła. Następnie tworzone są listy Pythona, które będą przechowywały informacje o następnych węzłach na poszczególnych poziomach oraz długościach skoków do nich.

Złożoność: Utworzenie węzła zajmuje stały czas $O(1)$.

Uwagi: Ze względu na to, że klasa __Node nie jest wykorzystywana poza klasą SkipList, konstruktor nie sprawdza poprawności danych.

4.1.2. Metoda __repr__()

Dane wejściowe: Brak.

Problem: Przekazanie jednoznacznej wartości w formie łańcucha znakowego.

Opis algorytmu: Metoda zwraca łańcuch znakowy w formacie "__Node(value, level)".

Złożoność: $O(1)$.

4.2. Szczegółowe omówienie metod klasy SkipList

Klasa SkipList posiada interfejs częściowo zgodny z interfejsem list Pythona.

4.2.1. Konstruktor __init__()

Dane wejściowe: Brak.

Problem: Utworzenie nowej instancji klasy SkipList.

Opis algorytmu: Konstruktor tworzy atrybuty, które przechowują informacje o prawdopodobieństwie, z jakim losowane są poziomy nowych węzłów, maksymalnym dozwolonym poziomie, aktualnej liczbie poziomów oraz długości listy. Tworzona jest także głowa listy.

Złożoność: Utworzenie pustej listy z przeskokami zajmuje stały czas $O(1)$.

4.2.2. Metoda insert()

Dane wejściowe: value - dowolna wartość posiadająca operatory porównania z pozostałymi elementami listy.

Problem: Umieszczenie nowego elementu na liście.

Opis algorytmu: Algorytm rozpoczyna się utworzeniem dwóch list Pythona służących kolejno do przechowywania informacji o ostatnich odwiedzonych węzłach oraz długościach skoków na poszczególnych poziomach. Następnie dla każdego poziomu listy w kolejności od najwyższego dopuszczalnego do najniższego przechodzi przez kolejne węzły oraz sumuje kolejne długości skoków, dopóki nie napotka elementu o wartości mniejszej lub równej tej dodawanej albo obiektu None. W takim przypadku zapamiętany zostaje ostatni węzeł spełniający powyższy warunek, a przeglądanie kontynuowane jest poziom niżej.

Po zakończeniu tej pętli algorytm tworzy nowy węzeł o wcześniej wylosowanym poziomie. Następnym krokiem jest powiązanie nowego węzła z już istniejącymi. W tym celu na każdym poziomie nowego wezła uprzednio zapamiętane węzły stają się jego poprzednikami, natomiast elementy, które dotychczas były następne po nich, następnikami wcześniej wspomnianego. W międzyczasie obliczane oraz przypisywane zostają długości skoków pomiędzy poszczególnymi węzłami.

W przypadku, gdy poziom węzła jest wyższy niż poziom nowego elementu, długości skoków "ponad" nim zostają zwiększone o jeden. W końcowym etapie działania algorytmu zostają zaktualizowane informacje o liczbie poziomów, liczbie elementów oraz najwyższym dopuszczalnym poziomie.

Złożoność: n to liczba elementów na liście z przeskokami.
Oczekiwana: $O(\log_2 n)$.
Pesymistyczna: $O(n)$.

Uwagi: Możliwe jest duplikowanie elementów, tj. na liście może występować więcej niż jeden węzeł o danej wartości.

4.2.3. Metoda remove()

Dane wejściowe: value - dowolna wartość posiadająca operatory porównania z pozostałymi elementami listy.

Problem: Usunięcie z listy elementu o podanej wartości.

Opis algorytmu: Pierwszym krokiem jest sprawdzenie, czy lista zawiera jakiekolwiek elementy. Jeśli nie, zwracany jest wyjątek IndexError. W przeciwnym wypadku działanie algorytmu jest kontynuowane.

Następnym krokiem jest utworzenie listy Pythona przeznaczonej na informacje o ostatnich odwiedzonych węzłach na kolejnych poziomach oraz łącza, który w tym pomoże.

Potem w kolejności od najwyższego do najniższego poziomu algorytm przechodzi przez kolejne węzły do momentu napotkania węzła o wartości wyższej lub równej usuwanej lub obiektu None. Wtedy poszukiwanie kontynuowane jest poziom niżej oraz zapamiętany zostanie ostatni element spełniający powyższy warunek.

Jeśli podana wartość nie została odnaleziona, zwracany jest wyjątek ValueError. W przeciwnym wypadku algorytm na wszystkich poziomach usuwanego elementu łączy jego poprzedników i następników oraz aktualizuje długości skoków pomiędzy nimi.

Następnym krokiem jest sprawdzenie liczby pustych górnych poziomów, tj. takich, na których głowa nie ma następnika. Na końcu zmniejszana jest wartość pola przechowującego informację o liczbie elementów.

Złożoność: n to liczba elementów na liście z przeskokami.
Oczekiwana: $O(\log_2 n)$.
Pesymistyczna: $O(n)$.

Uwagi: Usunięty zostanie tylko pierwszy odnaleziony element o podanej wartości.

4.2.4. Metoda find_min()

Dane wejściowe: Brak.

Problem: Znalezienie elementu o najmniejszej wartości.

Opis algorytmu: Algorytm zwraca wartość elementu znajdującego się na początku listy z przeskokami lub IndexError, jeśli lista jest pusta.

Złożoność: $O(1)$.

4.2.5. Metoda find_max()

Dane wejściowe: Brak.

Problem: Znalezienie elementu o największej wartości.

Opis algorytmu: Algorytm zwraca wartość elementu znajdującego się na końcu listy z przeskokami lub IndexError, jeśli lista jest pusta. Wykorzystuje się algorytm znajdowania elementu o indeksie równym długości listy minus jeden, ponieważ nie ma bezpośredniego dostępu do ostatniego elementu.

Złożoność: n to liczba elementów na liście z przeskokami.
Oczekiwana: $O(\log_2 n)$.
Pesymistyczna: $O(n)$.

Uwagi: Jeżeli w danym zastosowaniu list z przeskokami często szuka się największego elementu na liście, to można rozważyć inną implementację listy, opartą na liście podwójnie powiązanej. Wtedy istnieje bezpośredni dostęp do początku i końca listy, kosztem dodatkowej pamięci na łącza do poprzedników.

4.2.6. Metoda index()

Dane wejściowe: value - dowolna wartość posiadająca operatory porównania z pozostałymi elementami listy.

Problem: Znalezienie pierwszego wystąpienia na liście elementu o podanej wartości.

Opis algorytmu: Na początku algorytm tworzy zmienną przechowującą sumę długości kolejnych skoków oraz łącze, który w tym pomoże.

Następnie dla kolejnych poziomów począwszy na najwyższym, a na najniższym skończywszy, algorytm przechodzi przez kolejne węzły dopóki nie napotka na obiekt None lub wartość wyższą niż poszukiwana. W takim wypadku poszukiwanie kontynuowane jest poziom niżej. W trakcie działania pętli sumowane są kolejne długości skoków. Ponieważ długość pierwszego skoku na danym poziomie liczona jest od głowy, konieczne jest zmniejszenie uzyskanego wyniku o jeden.

Jeśli element o podanej wartości został odnaleziony, zwracana jest uzyskana w poprzednich krokach suma. W przeciwnym wypadku zwracany jest wyjątek ValueError.

Złożoność: n to liczba elementów na liście z przeskokami.
Oczekiwana: $O(\log_2 n)$.
Pesymistyczna: $O(n)$.

4.2.7. Metoda __len__()

Metoda specjalna umożliwiająca korzystanie z funkcji **len**().

Dane wejściowe: Brak.

Problem: Uzyskanie długości listy, tj. liczby elementów w niej zawartych.

Opis algorytmu: Metoda zwraca wartość atrybutu przechowującego informację o liczbie elementów.

Złożoność: $O(1)$.

4.2.8. Metoda __bool__()

Metoda specjalna umożliwiająca korzystanie z funkcji **bool**(), która oblicza wartość logiczną obiektu.

Dane wejściowe: Brak.

Problem: Obliczenie wartości logicznej reprezentowanej przez listę.

Opis algorytmu: Pusta lista ma wartość logiczną False, lista niepusta ma wartość logiczną True. Jest to spójne ze sposobem w jaki Python oblicza wartość logiczną dla klas kontenerowych, wyposażonych w metodę specjalną __len__.

Złożoność: $O(1)$.

4.2.9. Metoda __str__()

Metoda specjalna umożliwiająca korzystanie z funkcji **str**().

Dane wejściowe: Brak.

Problem: Wyznaczenie tekstowej reprezentacji listy.

Opis algorytmu: Algorytm przechodząc od najwyższego do najniższego poziomu listy, odwiedza wszystkie węzły i dodaje ich wartości do łańcucha znakowego. Gdy węzeł nie ma następcy, metoda przechodzi poziom niżej, a do wyniku dodawany jest znak przejścia do nowej linii. Na końcu zwracany jest uzyskany łańcuch znakowy.

Złożoność: $O(n)$, n to długość listy.

4.2.10. Metoda __delitem__()

Metoda specjalna umożliwiająca korzystanie ze słowa kluczowego **del**.

Dane wejściowe: index - dowolna liczba całkowita.

Problem: Usunięcie z listy elementu na pozycji index.

Opis algorytmu: Metoda rozpoczyna swoje działanie od sprawdzenia poprawności wprowadzonego indeksu. Jeśli indeks nie jest liczbą całkowitą, to zwracany jest wyjątek ValueError. W przypadku przekroczenia dozwolonego zakresu $[-s, s-1]$, gdzie s to długość listy, metoda zwraca IndexError. W przeciwnym wypadku algorytm kontynuuje działanie.

Do ujemnego indeksu dodawana jest długość listy, aby dostać wartość dodatnią. Niezależnie od znaku początkowej wartości indeksu, jest ona zwiększana o jeden. Metoda tworzy łącze służące do poruszania się po liście oraz listę Pythona do przechowywania informacji o ostatnich węzłach odwiedzonych na poszczególnych poziomach.

Następnym etapem jest przejście listy w kolejności od najwyższego do najniższego poziomu. Na każdym z nich algorytm odwiedza kolejne węzły do momentu natrafienia na element będący obiektem None lub którego długość skoku jest mniejsza lub równa indeksowi. W takiej sytuacji przeglądanie listy kontynuowane jest poziom niżej oraz zapamiętany zostaje ostatni węzeł spełniający powyższy warunek. Każdemu przejściu do następnego spełniającego warunek węzła towarzyszy odjęcie od indeksu długości skoku. Po zakończeniu pętli następuje przejście do następnego elementu na zerowym poziomie.

Algorytm na każdym poziomie usuwanego elementu łączy jego poprzednik z następnikiem oraz aktualizuje długości skoków. Na wyższych poziomach długość skoków "ponad" kasowanym węzłem zmniejsza się o jeden.

Końcowym etapem działania algorytmu jest zmniejszenie aktualnej liczby poziomów dla każdego poziomu, na którym następnikiem głowy jest obiekt None oraz aktualizacja długości listy.

Złożoność: n to liczba elementów na liście z przeskokami.
Oczekiwana: $O(\log_2 n)$.
Pesymistyczna: $O(n)$.

4.2.11. Metoda __iter__()

Metoda specjalna umożliwiająca korzystanie z iteratorów.

Dane wejściowe: Brak.

Problem: Iteracja po kolejnych elementach listy na jej najniższym poziomie.

Opis algorytmu: Algorytm tworzy generator, który iteruje po kolejnych elementach najniższego poziomu listy.

Złożoność: $O(n)$.

4.2.12. Metoda __getitem__()

Metoda specjalna umożliwiająca korzystanie z operatora [ ].

Dane wejściowe: index - dowolna liczba całkowita.

Problem: Dostęp do elementu o podanym indeksie.

Opis algorytmu: Algorytm w pierwszym kroku sprawdza poprawność wprowadzonego indeksu. Jeśli ten nie jest liczbą całkowitą, działanie metody przerywane jest przez wyjątek ValueError. Jeśli indeks jest spoza zakresu $[-s, s-1]$, gdzie s to długość listy, zwracany jest IndexError.

Do ujemnego indeksu dodawana jest długość listy, aby dostać wartość dodatnią. Niezależnie od znaku początkowej wartości indeksu, jest on zwiększany o jeden.

Kolejnym krokiem jest utworzenie łącza pomocniczego, służącego do poruszania się po liście. Algorytm rozpoczyna poszukiwanie wartości o podanym indeksie. W kolejności od najwyższego poziomu do najniższego odwiedzane są kolejne węzły niebędące obiektem None oraz o skoku krótszym lub równym wartości indeksu. Jeśli warunek nie zostanie spełniony, poszukiwanie kontynuowane jest poziom niżej. Każde przejście do następnego węzła spełniającego powyższy warunek wiąże się z zmniejszeniem indeksu o długość skoku. Metoda kończy działanie zwracając wartość odnalezionego węzła.

Złożoność: n to liczba elementów na liście z przeskokami.
Oczekiwana: $O(\log_2 n)$.
Pesymistyczna: $O(n)$.

4.2.13. Metoda __contains__()

Metoda specjalna umożliwiająca korzystanie ze słowa kluczowego **in**.

Dane wejściowe: value - dowolna wartość posiadająca operatory porównania z pozostałymi elementami listy.

Problem: Uzyskanie informacji o przynależności do listy elementu o podanej wartości.

Opis algorytmu: Pierwszym krokiem jest stworzenie łącza pomocniczego. Następnie metoda na kolejnych poziomach od najwyższego do najniższego przechodzi przez kolejne węzły niebędące obiektem None oraz o wartości niższej lub równej poszukiwanej. Niespełnienie tego warunku skutkuje kontynuowaniem poszukiwania na niższym poziomie. Jeśli odnaleziony węzeł nie jest obiektem None oraz ma oczekiwaną wartość, zwracanym wynikiem jest True. W przeciwnym wypadku metoda zwraca False.

Złożoność: n to liczba elementów na liście z przeskokami.
Oczekiwana: $O(\log_2 n)$.
Pesymistyczna: $O(n)$.

5. Mediana krocząca jako przykład zastosowania list z przeskokami

Niniejszy rozdział podzielony został na cztery części. Pierwsza z nich zawiera definicję mediany kroczącej. W drugiej części w formie instrukcji użytkownika przedstawiony zostanie interfejs klasy RunningMedian. Część trzecia poświęcona została szczegółowemu omówieniu algorytmów. Na końcu przedstawiono przykładową sesję interaktywną.

Testy porównujące szybkość wyznaczania mediany kroczącej z wykorzystaniem listy z przeskokami oraz listy posortowanej znajduje się w dodatku B. W dodatku C przedstawiono przykład praktycznego zastosowania interfejsu RunningMedian. Kod modułu RunningMedian znajduje się w dodatku F.

5.1. Podstawowe definicje

Mediana krocząca (ang. *moving median*, *running median*) jest to mediana wyznaczana w określonym fragmencie (oknie) szeregu wartości [13]. Dane wewnątrz okna zmieniają się w czasie, np. najstarsze zostają usunięte, po czym wolne miejsce uzupełniane jest najbliższą nieprzetworzoną wartością szeregu. Do wydajnego wyznaczania mediany kroczącej można wykorzystać indeksowane listy z przeskokami [7][14].

5.2. Interfejs klasy RunningMedian

5.2.1. Importowanie

Ze względu na prostotę użycia zalecanym sposobem importu klasy RunningMedian jest polecenie:

```
>>> from RunningMedian import RunningMedian
```

lub krócej:

```
>>> from RunningMedian import *
```

5.2.2. Tworzenie instancji klasy

Nowy obiekt tworzony jest w następujący sposób:

```
>>> nazwa_obiektu = RunningMedian(window, iterable)
```

Argumentami konstruktora są kolejno: rozmiar okna dla mediany (np. 100) oraz dowolny zawierający dane w postaci powiązanych ze sobą par (data

oraz powiązana z nią wartość) obiekt iterowalny, tj. taki, dla którego można utworzyć iterator.

5.2.3. Obliczanie mediany dla przesuwającego się okna

Obliczanie mediany dla zmieniających się danych możliwe jest dzięki generatorom. Przykładowo, możliwe jest wygenerowanie listy median dla zmieniających się danych:

```
>>> medians = RunningMedian(window, data)
>>> L = [item for item in medians]
```

lub manualne przesuwanie okna o jedną wartość:

```
>>> medians = iter(RunningMedian(window, data))
>>> item = next(medians)
```

Typowym przeznaczeniem klasy RunningMedian jest obliczanie mediany kroczącej dla par (data, dane), w związku z tym wartość zwracana przez iterator ma postać pary (data ostatnio wprowadzonego rekordu, mediana).

5.3. Algorytmy

W poprzednim podrozdziale przedstawiona została instrukcja korzystania z interfejsu klasy RunningMedian. Ten podrozdział stanowi jego rozwinięcie. Skupia się on wokół wykorzystanych algorytmów.

5.3.1. Konstruktor __init__()

Dane wejściowe:
window - dowolna dodatnia liczba całkowita.
iterable - iterowalny zbiór par danych.

Problem: Utworzenie nowej instancji klasy.

Opis algorytmu: W pierwszym kroku tworzony jest iterator dla obiektu z danymi przekazanego jako drugi argument. Następnie algorytm umieszcza pierwszy fragment o szerokości window danych z obiektu iterable w kolejce. Następnym krokiem jest umieszczenie danych z kolejki wewnątrz listy z przeskokami oraz zapamiętanie daty dla ostatniej zamieszczonej pary.

Złożoność: w to szerokość okna.
Oczekiwana: $O(w \log_2 w)$.
Pesymistyczna: $O(w^2)$.

5.3.2. Metoda __iter__()

Dane wejściowe: Brak.

Problem: Iteracja po kolejnych elementach strumienia danych.

Opis algorytmu: W pierwszym kroku algorytm ustala indeks elementu środkowego okna.

Dla pierwszego przejścia, tj. gdy wewnątrz okna są wszystkie wartości umieszczone tam przez konstruktor, zwracana jest zapamiętana w trakcie tworzenia instancji klasy data oraz wartość przechowywana na środku listy z przeskokami.

Dla kolejnych przejść iteratora mediany kroczącej algorytm pobiera dane (datę oraz wartość) wskazywane przez iterator danych wejściowych, usuwa z kolejki skrajną lewą wartość, wstawia do niej wartość pobraną oraz powtarza te dwa kroki dla listy z przeskokami.

W następnym kroku zwracana jest para (data ostatnio pobranego elementu, mediana wartości przechowywanych na liście z przeskokami).

Złożoność: w to szerokość okna, natomiast n to szerokość strumienia.
Oczekiwana: $O((n-w)\log_2 w)$.
Pesymistyczna: $O((n-w)*w)$.

5.4. Przykładowa sesja interaktywna

```
>>> from RunningMedian import RunningMedian
>>> import datetime
>>> import random
>>> M, N, window = 100, 20, 5
>>> day1 = datetime.date(2000, 1, 1)
>>> dt = datetime.timedelta(1)
>>> data = ((day1 + i * dt, random.randrange(M)) for i in range(N))
>>> medians = RunningMedian(window, data)
>>> for median in medians:
...     print(median)
...
(datetime.date(2000, 1, 5), 70.5)
(datetime.date(2000, 1, 6), 63.0)
(datetime.date(2000, 1, 7), 42.5)
(datetime.date(2000, 1, 8), 35.0)
(datetime.date(2000, 1, 9), 35.5)
(datetime.date(2000, 1, 10), 35.5)
(datetime.date(2000, 1, 11), 48.5)
(datetime.date(2000, 1, 12), 48.5)
(datetime.date(2000, 1, 13), 47.5)
(datetime.date(2000, 1, 14), 64.0)
(datetime.date(2000, 1, 15), 61.5)
(datetime.date(2000, 1, 16), 44.0)
(datetime.date(2000, 1, 17), 44.0)
(datetime.date(2000, 1, 18), 38.5)
(datetime.date(2000, 1, 19), 38.5)
```

```
(datetime.date(2000, 1, 20), 45.5)
>>>
>>>
>>> data = ((day1 + i * dt, random.randrange(M)) for i in range(N))
>>> medians = iter(RunningMedian(window, data))
>>> for i in range(5):
...     print(next(medians))
...
(datetime.date(2000, 1, 10), 51.5)
(datetime.date(2000, 1, 11), 51.5)
(datetime.date(2000, 1, 12), 51.5)
(datetime.date(2000, 1, 13), 37.5)
(datetime.date(2000, 1, 14), 35.0)
>>>
```

6. Podsumowanie

Celem pracy było stworzenie oraz przedstawienie implementacji listy z przeskokami w języku Python, z wykorzystaniem narzędzi z biblioteki standardowej. Problem stworzenia struktury danych łączącej wydajność oraz prostotę jest nieodłączną częścią pracy osób zajmujących się tematyką przechowywania oraz przetwarzania danych. Przez lata powstało wiele rozwiązań, które były albo proste (np. listy), albo wydajne (drzewa zrównoważone). Listy z przeskokami łączą cechy obu.

W pierwszej części przedstawiono genezę oraz motywację powstania list z przeskokami, a także założenia, zgodnie z którymi działają.

Część druga, poświęcona implementacji, przygotowana została w ten sposób, by użytkownik mógł zapoznać się ze sposobem korzystania z interfejsu opracowanego modułu. Zostało zrealizowane to poprzez przedstawienie celu korzystania z każdej z metod, przykładu jej wywołania, a także przedstawienia przykładu całej sesji interaktywnej z wykorzystaniem modułu. Dla osób bardziej dociekliwych przewidziany został podrozdział poświęcony szczegółowemu omówieniu każdej z metod, wraz z opisem działania kryjących się w nich algorytmów.

W ostatniej części czytelnik mógł zapoznać się z pojęciem mediany kroczącej, której wyznaczanie jest jednym z praktycznych zastosowań list z przeskokami. Zaczyna się ona przedstawieniem definicji. Podobnie jak w części poprzedniej, interfejs omówiony został w formie instrukcji użytkownika z przykładową sesją interaktywną oraz szczegółowego opisu algorytmów.

W formie dodatku przedstawione zostało wyznaczanie mediany kroczącej dla cen hurtowych benzyny 95-oktanowej. Jest to przykład praktycznego zastosowania przedstawionych w pracy modułów list z przeskokami oraz mediany kroczącej.

W efekcie czytelnik, który zapoznał się z tą pracą, posiada podstawy niezbędne do praktycznego korzystania z list z przeskokami.

A. Testy algorytmów

Poniżej przedstawione zostały wyniki testów poszczególnych algorytmów list z przeskokami. Są one średnią z 100 pomiarów kolejno dla 1000, 10000, 100000 oraz 1000000 różnych elementów z przedziałów odpowiednio: $[0, 1000]$, $[0, 10000]$, $[0, 100000]$, $[0, 1000000]$, umieszczonych na nieuporządkowanych listach Pythona.

Dla porównania identyczne testy wykonano dla drzew z biblioteki *bintrees*: AVLTree oraz RBTree.

Testy składały się z wstawienia nowego elementu o wartości wyższej niż dotychczasowa zawartość, usunięcia elementu o najwyższej wartości oraz odnalezienia elementu znajdującego się w połowie struktury. Wszystkie czasy podano w nanosekundach.

	1 000	10 000	100 000	1 000 000
SkipList	9735	12541	15367	18962
AVLTree	9966	12034	13852	14645
RBTree	6788	9811	11524	13952

Tabela A.1: Dodawanie elementu na koniec.

	1 000	10 000	100 000	1 000 000
SkipList	8009	11060	14370	16771
AVLTree	10512	12708	13864	15808
RBTree	17486	23930	27572	36346

Tabela A.2: Usuwanie elementu z końca.

	1 000	10 000	100 000	1 000 000
SkipList	7695	9003	12720	27462
AVLTree	2846	4073	4962	10725
RBTree	1793	1900	2838	7054

Tabela A.3: Wyszukiwanie elementu o środkowym indeksie.

Wnioski Na podstawie powyższych wyników można stwierdzić, że implementacja będąca tematem tej pracy dla mniejszych zestawów danych wykonuje operacje wstawiania w czasie porównywalnym do drzew AVL. Różnica większa niż $0.5\mu s$ widoczna jest dopiero dla zestawu danych o rozmiarze 100000. W każdym z testów wstawiania najszybsze były drzewa czerwono-czarne.

Różnice pomiędzy strukturami w tej tabeli są najmniejsze - żadna z nich nie jest więcej niż 1.5 razy szybsza od innej.

W przypadku usuwania ponownie widoczna jest przewaga list z przeskokami nad drzewami AVL dla zestawów danych wielkości 1000 oraz 10000. Dla wszystkich zestawów danych usuwanie ostatniego elementu najwięcej czasu zajmowało drzewom czerwono-czarnym. Pod tym względem były ok. 1.9-2.3 razy wolniejsze od najszybszej w danym teście struktury.

Największą różnicę zauważyć można w trzeciej tabeli, gdzie listy z przeskokami wypadły najgorzej. Dla poszczególnych zestawów danych były one między 3.9, a 4.7 razy wolniejsze od drzew czerwono-czarnych.

Każda z powyższych implementacji posiada swoje mocne i słabe strony. Wybór powinien zależeć od tego, jakie operacje będą dominować w danym zastosowaniu, a także jakiej wielkości dane będą przechowywane. Jeśli najczęściej wykonywanymi operacjami mają być dodawanie oraz usuwanie elementów, a liczba przechowywanych danych jest rzędu tysięcy lub dziesiątek tysięcy, najlepszym rozwiązaniem będą listy z przeskokami. Dla większych zestawów danych warto rozważyć drzewa AVL (bintrees.AVLTree). Jeśli natomiast struktura danych ma służyć głównie do wyszukiwania przechowywanych w niej elementów, najlepszym wyborem będą drzewa czerwono-czarne (bintrees.RBTree). Kompromisem pomiędzy szybkością wstawiania, usuwania oraz wyszukiwania, niezależnie od rozmiaru danych, będą drzewa AVL z biblioteki (bintrees).

B. Testy szybkości wyznaczania mediany kroczącej

Poniżej przedstawiono wyniki testów szybkości dla modułu RunningMedian w wersji opartej na liście z przeskokami oraz wersji wykorzystującej listę posortowaną (kod w dodatku G) dla danych w postaci par (kolejny dzień liczony od 1 stycznia 1700 roku, losowa wartość) w liczbie 1000, 10000, 100000 oraz 1000000 i wielkości okna 100, 1000, 2500, 5000 oraz 10000. Czas w tabelach jest średnią z 100 pomiarów i podany został w sekundach z zaokrągleniem do trzech miejsc po przecinku.

	1 000	10 000	100 000	1 000 000
RunningMedian	0.016	0.171	1.697	18.277
RunningMedianSlow	0.002	0.025	0.246	2.582

Tabela B.1: Porównanie implementacji algorytmu wyznaczania mediany kroczącej dla okna wielkości 100 elementów.

	1 000	10 000	100 000	1 000 000
RunningMedian	0.008	0.225	2.414	22.785
RunningMedianSlow	0.001	0.068	0.713	7.359

Tabela B.2: Porównanie implementacji algorytmu wyznaczania mediany kroczącej dla okna wielkości 1 000 elementów.

	1 000	10 000	100 000	1 000 000
RunningMedian	-	0.206	2.494	24.304
RunningMedianSlow	-	0.126	1.580	16.191

Tabela B.3: Porównanie implementacji algorytmu wyznaczania mediany kroczącej dla okna wielkości 2 500 elementów.

	1 000	10 000	100 000	1 000 000
RunningMedian	-	0.173	2.477	26.004
RunningMedianSlow	-	0.173	3.199	33.348

Tabela B.4: Porównanie implementacji algorytmu wyznaczania mediany kroczącej dla okna wielkości 5 000 elementów.

	1 000	10 000	100 000	1 000 000
RunningMedian	-	0.109	2.759	27.911
RunningMedianSlow	-	0.020	6.138	69.382

Tabela B.5: Porównanie implementacji algorytmu wyznaczania mediany kroczącej dla okna wielkości 10 000 elementów.

Wnioski

Na podstawie powyższych danych można stwierdzić, że dla dużych danych wejściowych oraz dużej szerokości okna implementacja oparta o listę z przeskokami jest lepszym wyborem niż wersja korzystająca z listy posortowanej.

C. Praktyczny przykład wyznaczania mediany kroczącej

Praktycznym przykładem zastosowania klasy RunningMedian może być wyznaczanie mediany z pewnej liczby rekordów w strumieniu (np. dziesięciu), którym są hurtowe ceny benzyny 95-oktanowej w latach 2004-2020. Dane ze strony [15] zostały umieszczone w pliku o rozszerzeniu .txt tak, by w jednej linii znalazły się data oraz cena.

Następnie metoda IterableData.reverse_and_make_data_iterable odwraca je, by były w kolejności od najstarszej do najnowszej daty, tworzy na ich podstawie pary obiektów (datetime.date, **float**), po czym umieszcza je w generatorze, który przekazywany jest jako argument konstruktora klasy RunningMedian.

Końcowym etapem jest iteracja po zawartości utworzonego obiektu klasy RunningMedian oraz zapisywanie otrzymanych par (data ostatnio wprowadzonego do okna rekordu, mediana wartości w oknie) w pliku o nazwie pb95_medians.txt.

Poniżej przedstawiono kod modułu IterableData oraz pliku petrol.py. Kod modułu RunningMedian znajduje się w dodatku F. Kod jest kompatybilny zarówno z Pythonem 2.7, jak i Pythonem 3.

Listing C.1. Moduł IterableData.

```
import datetime


class IterableData:

    @staticmethod
    def reverse_and_make_data_iterable(input_file):
        with open(input_file) as input_data:
            lines = input_data.readlines()

            for line in reversed(lines):    # parse file content
                line = line.rstrip("\n")
                data = line.split("\t")
                date_tmp = list(map(int, reversed(data[0].split("-"))))
                price_tmp = data[1].split(" ")
                price = int("".join(price_tmp[:-1]))
                date = datetime.date(date_tmp[0], date_tmp[1], date_tmp[2])

                yield date, price
```

Listing C.2. Plik petrol.py.

```
from RunningMedian import RunningMedian
from IterableData import IterableData
```

```
window = 10
data = IterableData.reverse_and_make_data_iterable("benzyna1.txt")
output_file = open("pb95_medians.txt", 'w')
output_file.truncate(0)

medians = iter(RunningMedian(window, data))
try:
    line = next(medians)
    while True:
        output_file.write(str(line))
        line = next(medians)
        output_file.write('\n')
except StopIteration:
    pass
```

D. Problem odwracania danych wejściowych

W poprzednim dodatku przedstawiony został przykład użycia modułu RunningMedian do obliczania mediany dla rzeczywistych danych. W tym celu konieczne było odwrócenie kolejności danych wejściowych. Poniżej przedstawione zostały dwa sposoby, na które można to zrobić. Ponieważ w tym dodatku omawiany jest przypadek ogólny, sprowadzanie danych wejściowych do par wymaganych przez moduł RunningMedian zostało pominięte.

Sposób 1
Pierwszy sposób polega na wczytaniu całej zawartości pliku do pamięci (przy użyciu funkcji readlines), by potem iterować po niej w kolejności od ostatniej do pierwszej linii. Zaletą takiego rozwiązania jest prostota - cała procedura zajmuje kilkanaście linii kodu. Wadą jest konieczność przechowywania w pamięci całej zawartości pliku wejściowego.

Listing D.1. Odwracanie kolejności danych - sposób 1.

```
#!/usr/bin/python

input_file = open("benzyna1.txt")
lines = input_file.readlines()

print("Input file num of lines: {}".format(len(lines)))

output = open("benzyna1_rev.txt", 'w')
output.truncate(0)
# in case that file exists and is not empty

for line in reversed(lines):
    output.write(line)
    if line[-1] != '\n':
        output.write('\n')

input_file.close()
output.close()
```

Sposób 2
Ten sposób pochodzi z [16]. Funkcja tworzy generator, który począwszy od końca pliku czyta kolejne fragmenty danych i składa w linie, po czym zwraca każdą z nich. Zaletą jest wczytywanie danych w sposób leniwy, tj. dany fragment jest wczytywany dopiero w momencie, gdy jest potrzebny. Dzięki temu nie jest konieczne przechowywanie w pamięci zawartości całego pliku.

Rozwiązanie to ma jednak dwie wady. Pierwszą z nich jest możliwość błędnego odczytania znaków Unicode w Pythonie 3. Wynika to stąd, że funkcja seek operuje na bajtach, natomiast read na poszczególnych znakach. Drugim problemem jest pomijanie pustych linii. Występuje ono zarówno w Pythonie 2, jak i 3.

Listing D.2. Odwracanie kolejności danych - sposób 2.

```
#!/usr/bin/python
#
# https://stackoverflow.com/questions/2301789/how-to-read-a-file-in-reverse-order
#
# Problem 1: in Python 3 wrong Unicode calculation.
# [seek() counts in bytes, read() in characters].
# Problem 2: empty rows are ignored.

import os


def reverse_readline(filename, buf_size=32):  # works with Python 2
    """A generator that returns the lines of a file in reverse order."""
    with open(filename) as fh:
        segment = None
        offset = 0
        fh.seek(0, os.SEEK_END)
        file_size = remaining_size = fh.tell()
        print("file_size {} buf_size {}".format(file_size, buf_size))
        while remaining_size > 0:
            offset = min(file_size, offset + buf_size)  # distance from end
            fh.seek(file_size - offset)
            print("Position from beginning {}".format(file_size - offset))
            buf = fh.read(min(remaining_size, buf_size))
            remaining_size -= buf_size
            lines = buf.split('\n')
            # The first line of the buffer is probably not a complete line so
            # we'll save it and append it to the last line of the next buffer
            # we read
            #
            # If line does not contain '\n', the last item in
            # split() result will be ''
            if segment is not None:
                # If the previous chunk starts right from the beginning of line
                # do not concat the segment to the last line of new chunk.
                # Instead, yield the segment first
                if buf[-1] != '\n':
                    lines[-1] += segment
                else:
                    yield segment
            segment = lines[0]
            for index in range(len(lines) - 1, 0, -1):
                if lines[index]:
                    yield lines[index]
        # Don't yield None if the file was empty
        if segment is not None:
            yield segment
```

```
if __name__ == '__main__':

    output = open("benzyna2_rev.txt", 'w')
    output.truncate(0)
    # in case that file exists and is not empty

    for line in reverse_readline("benzyna1.txt"):
        output.write(line + '\n')

    output.close()

# EOF
```

E. Kod źródłowy modułu SkipList

Przy tworzeniu poniżej przedstawionego kodu posiłkowano się następującymi źródłami: [3], [7], [11] oraz [12]. Kod jest kompatybilny zarówno z Pythonem 2.7, jak i Pythonem 3.

Listing E.1. Moduł SkipList.

```
from math import floor, ceil, log
from random import random


class SkipList(object):
    __slots__ = ('__possibility', '__maxLevel', '__header',
                 '__elements', '__numOfLevels', '__dict__')

    class __Node:
        __slots__ = ('value', 'level', 'next', 'skip_length', '__dict__')

        def __init__(self, value, level):
            self.value, self.level = value, level
            self.next = [None] * (level + 1)  # next nodes
            self.skip_length = [0] * (level + 1)    # length of skip to next eleme

        def __repr__(self):
            return "__Node({}, {})".format(self.value, self.level)

    def __init__(self):
        self.__possibility = 0.5
        # self.__oppositePossibility = 1 - self.__possibility
        self.__maxLevel = 0
        self.__header = self.__Node(None, 0)
        self.__elements = 0  # number of nodes
        self.__numOfLevels = 1
        # number of levels, can be different than self.__maxLevel + 1

    def __len__(self):  # length
        return self.__elements

    def __bool__(self):  # false when empty
        # Python interprets zero as False
        return bool(self.__elements)

    def __str__(self):  # all levels as string
        result = ""
        for idx in reversed(range(self.__numOfLevels)):
            result += ("Level " + str(idx) + ": [")
            current = self.__header.next[idx]
            while current is not None:
                result += str(current.value) + ", "
                current = current.next[idx]
```

```
            if result[-2:] != " [":
                result = result[:-2]  # remove space and comma
            result += "]\n"

        return result

    def __delitem__(self, index):
        self.__test_index(index)
        if index < 0:
            index += self.__elements
        index += 1
        current = self.__header
        previous = [None] * self.__numOfLevels

        # search for value
        for level in reversed(range(self.__numOfLevels)):
            while (current.next[level] is not None and
                   current.skip_length[level] < index):
                index -= current.skip_length[level]
                current = current.next[level]
            previous[level] = current
        current = current.next[0]

        self.__remove_helper(current, previous)

    def __iter__(self):  # returns generator to elements on level 0
        current = self.__header.next[0]
        while current is not None:
            yield current.value
            current = current.next[0]

    def __getitem__(self, index):  # [] operator
        self.__test_index(index)
        if index < 0:
            index = self.__elements + index
        index += 1
        current = self.__header

        for level in reversed(range(self.__maxLevel + 1)):
            while (current.next[level] is not None and
                   current.skip_length[level] <= index):
                index -= current.skip_length[level]
                current = current.next[level]

        return current.value

    def __contains__(self, value):
        current = self.__header

        for level in reversed(range(self.__maxLevel + 1)):
            while (current.next[level] is not None and
                   current.next[level].value <= value):
                current = current.next[level]

        return current is not None and current.value == value
```

```
def __random_level(self):
    level = 0
    while random() < self.__possibility and level < self.__maxLevel:
        level += 1

    # level = min(self.__maxLevel, -int(log(1 - random(), 2)))

    return level

def __calc_max_level(self):
    new_max_level = int(floor(log(self.__elements, 2)))
    if new_max_level > self.__maxLevel:
        self.__header.next.append(None)
        self.__header.skip_length.append(0)
        self.__maxLevel = new_max_level

def __test_index(self, index):
    if not isinstance(index, int):  # wrong type
        raise ValueError("Index is not int")

    if self.__elements - ceil(abs(index + 0.5)) < 0:
        raise IndexError("Index out of range")

def __remove_helper(self, current, previous):
    # set next nodes of removed node as next node of previous nodes
    for level in range(current.level + 1):
        tmp_previous = previous[level]
        tmp_previous.skip_length[level] += \
            tmp_previous.next[level].skip_length[level]
        tmp_previous.skip_length[level] -= 1
        tmp_previous.next[level] = current.next[level]

    for level in range(current.level + 1, self.__numOfLevels):
        previous[level].skip_length[level] -= 1

    # remove empty levels
    while (self.__numOfLevels > 1 and
           self.__header.next[self.__numOfLevels - 1] is None):
        self.__numOfLevels -= 1

    self.__elements -= 1

def insert(self, value):
    # last elements in level when comparison returns true
    previous = [None] * (self.__maxLevel + 1)
    skip_len_tmp = [0] * (self.__maxLevel + 1)

    current = self.__header  # currently visited node
    for level in reversed(range(self.__maxLevel + 1)):
        # search for "good place" for new node
        while (current.next[level] is not None and
               current.next[level].value < value):
            skip_len_tmp[level] += current.skip_length[level]
            current = current.next[level]  # go to next node
        previous[level] = current  # go 1 level down

    rand_lvl = self.__random_level()
```

```
        current = self.__Node(value, rand_lvl)

        hops = 0
        for level in range(current.level + 1):
            # "connect" new node with the rest of the list
            current.next[level] = previous[level].next[level]
            previous[level].next[level] = current
            current.skip_length[level] = \
                previous[level].skip_length[level] - hops
            previous[level].skip_length[level] = hops + 1
            hops += skip_len_tmp[level]

        for level in range(current.level + 1, self.__maxLevel + 1):
            previous[level].skip_length[level] += 1

        self.__numOfLevels = max(self.__numOfLevels, rand_lvl + 1)
        self.__elements += 1
        self.__calc_max_level()

    def remove(self, value):
        if self.__elements == 0:
            raise IndexError("Empty list")
        previous = [None] * self.__numOfLevels
        current = self.__header

        for level in reversed(range(self.__numOfLevels)):
            # search for value
            while (current.next[level] is not None and
                   current.next[level].value < value):
                current = current.next[level]
            previous[level] = current
        current = current.next[0]

        if current is not None and current.value == value:
            # value was found
            self.__remove_helper(current, previous)

        else:
            raise ValueError("Value not found")

    def find_min(self):
        result = self.__header.next[0]

        if result is None:
            raise IndexError("Empty list")
        else:
            return result.value

    def find_max(self):
        if self.__header.next[0] is None:
            raise IndexError("Empty list")
        else:
            return self[self.__elements - 1]

    def index(self, value):
        path_width = 0
        current = self.__header
```

```
        for level in reversed(range(self.__numOfLevels)):
            while (current.next[level] is not None and
                   current.next[level].value <= value):
                path_width += current.skip_length[level]
                current = current.next[level]

        if current is not None and current.value == value:
            return path_width - 1
        else:
            raise ValueError("Value not found")
```

F. Kod źródłowy modułu RunningMedian

Przy tworzeniu poniżej przedstawionego kodu posiłkowano się programem Raymonda Hettingera [7]. Kod jest kompatybilny zarówno z Pythonem 2.7, jak i Pythonem 3.

Listing F.1. Moduł RunningMedian.

```
#!/usr/bin/python

from collections import deque
from itertools import islice
# Make an iterator that returns selected elements from the iterable.
from math import floor, ceil
from SkipList import SkipList
import sys, os
sys.path.append(os.path.abspath(os.path.join('../main/final')))


class RunningMedian:
    """Running-median using deque and SkipList."""

    def __init__(self, window, iterable):
        # iterable may be an generator
        # iterable contains pairs (date, value)
        self.__it = iter(iterable)  # iterable may be large
        self.__queue = deque(islice(self.__it, window))  # collect window elements
        self.__skip_list = SkipList()
        self.__last_date = None
        for item in self.__queue:
            self.__last_date, value = item
            self.__skip_list.insert(value)

    def __iter__(self):
        middle = len(self.__queue) / 2
        median = (self.__skip_list[int(ceil(middle))]
                  + self.__skip_list[int(floor(middle))]) / 2
        yield self.__last_date, median
        for new_item in self.__it:  # collecting next elements
            self.__last_date, new_value = new_item
            old_item = self.__queue.popleft()  # O(1) time
            old_date, old_value = old_item
            self.__queue.append(new_item)  # O(1) time
            self.__skip_list.remove(old_value)  # O(log w) time
            self.__skip_list.insert(new_value)
            median = (self.__skip_list[int(ceil(middle))]
                      + self.__skip_list[int(floor(middle))]) / 2
            yield self.__last_date, median
```

G. Kod źródłowy modułu RunningMedianSlow

Różnice w stosunku do RunningMedian Główną różnicą jest zastąpienie listy z przeskokami listą posortowaną. W celu uniknięcia konieczności sortowania listy po każdym wstawieniu nowego elementu, odbywa się ono z użyciem metody bisect.insort(). Oczekiwana złożoność obliczeniowa tej metody jest rzędu $O(w)$, gdzie w oznacza wielkość okna. Podobnie jest w przypadku usuwania elementu z listy, stąd też nazwa tego modułu - RunningMedianSlow.

Przy tworzeniu poniżej przedstawionego kodu posiłkowano się programem Raymonda Hettingera [7]. Kod jest kompatybilny zarówno z Pythonem 2.7, jak i Pythonem 3.

Listing G.1. Moduł RunningMedianSlow.

```
#!/usr/bin/python

import bisect  # insort
from collections import deque
from itertools import islice
# Make an iterator that returns selected elements from the iterable.
from math import ceil, floor


class RunningMedianSlow:
    """Slow running-median using deque and a Python list."""

    def __init__(self, window, iterable):
        # iterable can be an iterator
        self.it = iter(iterable)  # iterable may be large
        # collect window elements
        self.queue = deque(islice(self.it, window))
        # often seen code: list(islice(it, n))
        self.last_date = None
        self.sorted_list = []
        for item in self.queue:
            self.last_date, value = item
            bisect.insort(self.sorted_list, value)

    def __iter__(self):
        middle = len(self.queue) / 2
        median = (self.sorted_list[int(floor(middle))]
                  + self.sorted_list[int(ceil(middle))]) / 2
        yield self.last_date, median
        for new_item in self.it:  # collecting next elements
            self.last_date, new_value = new_item
            old_item = self.queue.popleft()  # O(1) time
            old_date, old_value = old_item
            self.queue.append(new_item)  # O(1) time
```

```
            self.sorted_list.remove(old_value)   # O(w) time
            bisect.insort(self.sorted_list, new_value)   # O(w) time
            median = (self.sorted_list[int(floor(middle))]
                      + self.sorted_list[int(ceil(middle))]) / 2
            yield self.last_date, median
```

Bibliografia

[1] Python Programming Language - Official Website, https://www.python.org/.

[2] Wikipedia, Skip list, 2020, https://en.wikipedia.org/wiki/Skip_list.

[3] William Pugh, Skip Lists: A Probabilistic Alternative to Balanced Trees, Communications of the ACM 33 (6), 668-676 (1990).

[4] William Pugh, A Skip List Cookbook, Technical Report UMIACS-TR-89-72.1, University of Maryland at College Park College Park, MD, USA, 1990.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Wprowadzenie do algorytmów*, Wydawnictwo Naukowe PWN, Warszawa 2012.

[6] Adam Drozdek, *C++. Algorytmy i struktury danych*, Helion 2004.

[7] Raymond Hettinger, *Efficient running median using an indexable skiplist (Python recipe)*, 2009. http://code.activestate.com/recipes/576930/

[8] *Linked Lists* by Victor S. Adamchik, Carnegie Mellon University, 2009, https://www.cs.cmu.edu/~adamchik/15-121/lectures/Linked%20Lists/linked%20lists.html

[9] Złożoność list powiązanych, https://en.wikipedia.org/wiki/Linked_list#Linked_lists_vs._dynamic_arrays

[10] Big-O Datasheet, https://www.bigocheatsheet.com/

[11] GeeksforGeeks - Skip List Set 2 (Insertion), https://www.geeksforgeeks.org/skip-list-set-2-insertion/

[12] GeeksforGeeks - Skip List Set 3 (Searching and Deletion), https://www.geeksforgeeks.org/skip-list-set-3-searching-deletion/

[13] IBM Knowledge Center, Moving Median, https://www.ibm.com/support/knowledgecenter/pl/SSLVMB_subs/statistics_mainhelp_ddita/spss/base/time_series_functions.html

[14] Wikipedia, Moving Median, https://en.wikipedia.org/wiki/Moving_average#Moving_median

[15] Orlen, ceny paliw, https://www.orlen.pl/PL/DlaBiznesu/HurtoweCenyPaliw/Strony/Archiwum-Cen.aspx

[16] How to read a file in reverse order?, stackoverflow.com, 2010, https://stackoverflow.com/questions/2301789/how-to-read-a-file-in-reverse-order