SPRAWOZDANIE 2

WPROWADZENIE:

Celem, niniejszego sprawozdania było wykonanie, zbadanie i sporządzenie poszczególnych wykresów, przedstawiające działanie drzew BST i AVL. Drzewa były testowane na danych wejściowych, wygenerowanych przez generator liczb losowych (niepowtarzających się).

```
def random_generator(zakres_poczatek, zakres_koniec, ilosc_liczb):
   random_numbers = []
   while len(random_numbers) < ilosc_liczb:
       num = random.randint(zakres_poczatek, zakres_koniec)
       if num not in random_numbers:
           random_numbers.append(num)
   return random_numbers
```

Sprawozdanie zostało rozłożone na poszczególne sekcje. W pierwszej z nich znajduje się prezentacja całego kodu i poszczególnych struktur. Z względu, że drzewo AVL posiada wiele tych samych funkcji wewnętrznych co BST, lub są one “prawie” podobne, to ich różnica została podkreślona dzięki komentarzom. Sekcja druga dotyczy zależności czasowej od danych wejściowych w poszczególnych operacjach takich jak:
-szukanie najmniejszego elementu
-tworzenie struktury
-wypisanie in-order
W ostatniej sekcji, znajduje się podsumowanie oraz zależność czasu (t) równoważenia drzewa BST od liczby elementów, z podanego zakresu.

LINK DO ARKUSZA EXCEL Z DANYMI:
https://docs.google.com/spreadsheets/d/1huQyXaq7cDkxGXLs6mHQQLNNndvpl147q5ChNAqO96w/edit?usp=sharing

SEKCJA 1

struktura drzewa BST:

W kodzie tym została zaimplementowana funkcja dodaj_z_listy w celu łatwiejszego dodawania z listy wejściowej. Ponadto aby skutecznie realizować niektóre modyfikacje i funkcje, na drzewie konieczne było stworzenie klasy “Wezel” zawierającej własną wartość (dane), informacje na temat lewego, prawego dziecka i współczynnik równowagi, który jest aktualizowany po dodaniu każdego elementu. Wszystkie linie kodu zawierają komentarz.

```
class Wezel:
   def __init__(self, dane=None):
       self.dane = dane
       self.lewe_dziecko = None
       self.prawe_dziecko = None
       self.wspolczynnik_rownowagi = 0
```

```
class BST:
   def __init__(self):
       self.korzen = None

   def dodaj(self, dane):
       if self.korzen is None:              #jeżeli w drzewie nie
ma elementu
           self.korzen = Wezel(dane)        # to pierwszy dodany
będzie korzeniem
       else:
           self.dodaj_do_wierzcholka(dane, self.korzen)
#jeżeli są już jakieś elementy to dodajemy je do istniejącej
struktury

   def dodaj_do_wierzcholka(self, dane, wierzcholek):
#funkcja pomocnocz, pomaga w znalezeniu odpowiedniego miejsca do
wstawienia elementu w drzewie
       if dane < wierzcholek.dane:
#jeżeli dane(input) są mniejsze od danego wierzchołka (patrzymy w
lewo)
           if wierzcholek.lewe_dziecko is None:
#jeżeli lewe dziecko nie istnieje to   lewe = dane
               wierzcholek.lewe_dziecko = Wezel(dane)
           else:
               self.dodaj_do_wierzcholka(dane,
wierzcholek.lewe_dziecko)       #jeżeli lewe dziecko istnieje to
rekurencyjnie wywołujemy funkcje by znalezc odpowiednie miejsce
       elif dane > wierzcholek.dane:
#jeżeli dane(input) są większe od danego wierzchołka (patrzymy w
prawo)
           if wierzcholek.prawe_dziecko is None:
#jeżeli prawe dziecko nie istnieje to    prawe = dane
               wierzcholek.prawe_dziecko = Wezel(dane)
           else:
               self.dodaj_do_wierzcholka(dane,
wierzcholek.prawe_dziecko)          #jeżeli prawe dziecko istnieje
to rekurencyjnie szukamy kolejnego miejsca na nowy element

       # Po dodaniu węzła, aktualizujemy wartość współczynnika
równowagi dla każdego węzła
       wierzcholek.wspolczynnik_rownowagi =
self.oblicz_wspolczynnik_rownowagi(wierzcholek)     #po dodaniu
elementu aktualizujemy wspoczynnik rownowagi (potrzebne do
rownowazenia)


   def dodaj_z_listy(self, lista):                     #funkcja
pomocnicza do dodawania elementów z listy
```

```
        for element in lista:
            self.dodaj(element)
```

struktura drzewa AVL wygląda tak samo, z różnicą w funkcji dodaj_z_listy:

```
def dodaj_z_listy(self, lista):
    if len(lista) == 0:  # Sprawdzamy czy lista jest pusta
        return

    srodkowy_indeks = len(lista) // 2  # Znajdujemy indeks
środkowego elementu
    srodkowy_element = lista[srodkowy_indeks]  # Pobieramy środkowy
element
    self.dodaj(srodkowy_element)  # Dodajemy środkowy element do
drzewa AVL

    lewa = lista[:srodkowy_indeks]  # Tworzymy listę elementów po
lewej stronie
    prawa = lista[srodkowy_indeks + 1:]  # Tworzymy listę elementów
po prawej stronie

    self.dodaj_z_listy(lewa)  # Rekurencyjnie dodajemy lewą część
do drzewa AVL
    self.dodaj_z_listy(prawa)  # Rekurencyjnie dodajemy prawą część
do drzewa AVL
```

dzięki takiej implementacji, po dodaniu elementów do drzewa z listy, drzewo zawsze będzie zbalansowane.

Współczynnik równowagi danego wierzchołka, potrzebny do realizacji balansowania drzewa, zostaje obliczony dzięki wartości bezwzględnej z różnicy wysokości lewego i prawego dziecka.

```
#[FUNKCJE DLA AVL I BST]
def oblicz_wysokosc(self, wezel):
    if wezel is None:              #warunek kończący rekurencje
        return 0
    lewa_wysokosc = self.oblicz_wysokosc(wezel.lewe_dziecko)
    prawa_wysokosc = self.oblicz_wysokosc(wezel.prawe_dziecko)
    return 1 + max(lewa_wysokosc, prawa_wysokosc)
#zwracanie wysokosci wiekszej (lewej lub prawej) +1 bo rodzic

def oblicz_wspolczynnik_rownowagi(self, wezel):
    if wezel is None:             #warunek kończący rekurencje
        return 0
    lewa_wysokosc = self.oblicz_wysokosc(wezel.lewe_dziecko)
    prawa_wysokosc = self.oblicz_wysokosc(wezel.prawe_dziecko)
    return abs(lewa_wysokosc - prawa_wysokosc)
#zwracanie wartosci bezwzględnej wspolczynnika rownowagi
```

Szukanie najmniejszego i największego elementu w drzewie, z właściwości dodawania każdego dodatkowego wierzchołka wiemy, że najmniejszy element znajduje się najbardziej po lewej stronie, natomiast największy najbardziej po prawej (tz. idąc po dzieciach wierzchołka). Dodatkowo funkcja zwraca listę która zawiera ścieżkę do szukanych elementów.

```
#ZNALEZIENIE NAJMNIEJSZEGO I NAJWIEKSZEGO ELEMENTU
   def najmniejszy(self):
       if self.korzen is None:  #sprawdzenie czy drzewo jest puste
           return None, []

       sciezka = []                #zapisanie sciezki
       aktualny = self.korzen
       while aktualny.lewe_dziecko:
           sciezka.append(aktualny.dane)      #zapis do sciezki
           aktualny = aktualny.lewe_dziecko    #zmiana aktualnego
       sciezka.append(aktualny.dane)

       return aktualny.dane, sciezka

   def najwiekszy(self):
       if self.korzen is None: #sprawdzenie czy drzewo jest puste
           return None, []

       sciezka = []                #zapis sciezki
       aktualny = self.korzen
       while aktualny.prawe_dziecko:
           sciezka.append(aktualny.dane)
           aktualny = aktualny.prawe_dziecko
       sciezka.append(aktualny.dane)

       return aktualny.dane, sciezka
```

Usuwanie elementu o wybranym kluczu, zostało specjalnie dostosowane do równoważenia drzewa metodą **usuwania korzenia,** funkcja _usun_wierzcholek najpierw znajduje odpowiedni element, i w zależności od przypadku, jeżeli wierzchołek zawiera:
-*jedno dziecko* lewe lub prawe to usuwany wierzchołek zostaje zastąpiony przez dziecko.
-*dwoje dzieci* to zostaje wybrany odpowiedni następnik, który jest dobierany z dziecka które ma największą wysokość. Zasada doboru następnika z wybranego dziecka jest taka sama jak na prezentacji “ Wykład 4 (drzewa poszukiwań binarnych) - KLIKNIJ“ na stronie 28. Po zamianie elementu na następnik, występuje duplikat następnika, który jest usuwany rekurencyjnie.
Dodatkowo została zaimplementowana funkcja znajdz_wartosc z względu na to, że _nastepnik zwracał element jako klucz a nie obiekt, i należało to zmienić. W drzewie AVL dodana jest linia która ma balansować drzewo po usunięciu elementu.

```
   def usun(self, dane):
       """Metoda usuwająca wierzchołek o podanej wartości."""
       self.korzen = self._usun_wierzcholek(self.korzen, dane)
```

```
    def _usun_wierzcholek(self, wierzcholek, dane):
        """Metoda pomocnicza usuwająca wierzchołek o podanej 
wartości."""
        #dane - które chcemy usunąć
        #wierzchołek - aktualnie sprawdzany wierzchołek

        if wierzcholek is None:
            return None

        if dane < wierzcholek.dane:                        
#sprawdzanie/szukanie wierzchołka do usunięcia, (zasada prawe 
większe lewo mniejsze)
            wierzcholek.lewe_dziecko = 
self._usun_wierzcholek(wierzcholek.lewe_dziecko, dane)
        elif dane > wierzcholek.dane:
            wierzcholek.prawe_dziecko = 
self._usun_wierzcholek(wierzcholek.prawe_dziecko, dane)
        else:
            if wierzcholek.lewe_dziecko is None:        #jeżeli 
drzewo ma tylko prawe dziecko - prawe jest następnikiem
                return wierzcholek.prawe_dziecko
            elif wierzcholek.prawe_dziecko is None:     #jeżeli 
drzewo ma tylko lewe dziecko - lewe jest następnikiem
                return wierzcholek.lewe_dziecko
            else:
                nastepnik, rodzaj = self._nastepnik(wierzcholek)     
#jeżeli drzewo ma dwoje dzieci szukamy następnika za pomocą 
_nastepnik SLAJDY!!!
                wierzcholek.dane = nastepnik.dane                    
#ustawienie znalezionego nastepnika

                if rodzaj == "prawe":         #po ustawieniu 
wierzhołka na nastepnik, mamy "duplikat" więc musimy usunąć 
następnik z drzewa
                    wierzcholek.prawe_dziecko = 
self._usun_wierzcholek(wierzcholek.prawe_dziecko, nastepnik.dane)
                else:
                    wierzcholek.lewe_dziecko = 
self._usun_wierzcholek(wierzcholek.lewe_dziecko, nastepnik.dane)

        #aktualizacja wspoczynnika rownowagi wierzcholka
        wierzcholek.wspolczynnik_rownowagi = 
self.oblicz_wspolczynnik_rownowagi(wierzcholek)

        #rownowazenie drzewa
        self.rownowaz_drzewo()

        return wierzcholek
```

```
  def _nastepnik(self, wierzcholek):
      if wierzcholek is None:
          return None

      klucze_in_order = self.in_order(wierzcholek)
#stworzenie listy kluczy "posortowanych"

      lewa_wysokosc =
self.oblicz_wysokosc(wierzcholek.lewe_dziecko)      #sprawdzenie
lewej i prawej wysokosci by zadecydowac
      prawa_wysokosc =
self.oblicz_wysokosc(wierzcholek.prawe_dziecko)    # z ktorej
strony wziac następnik (z klucze_in_order)  - potrzebne do
balansowania drzewa

      # Znajdowanie indeksu klucza, który ma być usunięty w
liscie
      indeks = klucze_in_order.index(wierzcholek.dane)

      if lewa_wysokosc >= prawa_wysokosc and indeks - 1 >= 0:
#jeżeli lewa wys jest większa od prawej to bierzemy następnik z
lewego dziecka     ROWNE bo avl może mieć równe wysokosci
          rodzaj = "lewe"                              #rodzaj
potrzebny do ifa w funkcji _usun_wierzcholek
          nastepnik_dane = klucze_in_order[indeks - 1]          #
-1 - z lewej strony listy
          nastepnik_wezel = self.znajdz_wartosc(nastepnik_dane)
      elif lewa_wysokosc < prawa_wysokosc and indeks + 1 <
len(klucze_in_order): #jeżeli prawa wys jest większa od lewej to
bierzemy następnik z prawego dziecka
          rodzaj = "prawe"
          nastepnik_dane = klucze_in_order[indeks + 1]     # +1 -
z prawej strony listy
          nastepnik_wezel = self.znajdz_wartosc(nastepnik_dane)
      else:
          return None

      return nastepnik_wezel, rodzaj
#
  def znajdz_wartosc(self, wartosc, wierzcholek=None):
#potrzebne do _nastepnik bo zwracał klucz jako wartość a nie
obiekt
      if wierzcholek is None:
          wierzcholek = self.korzen       #jeżeli nie podamy
żadnego wierzchołka, to automatycznie jest on ustawiany jako
korzeń
```

```
       if wierzcholek is None:             #sprawdzanie czy
wierzchołek jest pusty (czy drzewo jest puste)
           return None

       if wartosc == wierzcholek.dane:     #zwracanie wierzchołka
jeżeli został znaleziony
           return wierzcholek

       if wartosc < wierzcholek.dane:
#szykanie wierzchołka po drzewie kierując się zasadą;
           return self.znajdz_wartosc(wartosc,
wierzcholek.lewe_dziecko)    # większe na prawo, mniejsze na lewo
       else:
           return self.znajdz_wartosc(wartosc,
wierzcholek.prawe_dziecko)
```

Usuwanie w kierunku post_order, najpierw trawersujemy lewe dziecko, potem prawe a na samym końcu korzeń, po każdym elemencie następuje funkcja usun.

```
#dla AVL I BST
def post_order(self, wierzcholek):
   if wierzcholek:                #sprawdzenie czy wierzchołek
istnieje
       self.post_order(wierzcholek.lewe_dziecko)       #LEWE ----
rekurencyjnie do lewego i prawego dziecka
       self.post_order(wierzcholek.prawe_dziecko)      #PRAWE ----
kolejnosc ma znaczeni bo ---> post-order: lewe poddrzewo, prawe
poddrzewo, korzeń
       if wierzcholek.dane:       #KORZEŃ --- jeżeli wierzchołek
istnieje to go usuwamy
           print("usunieto: ", str(wierzcholek.dane))
           self.usun(wierzcholek.dane)
```

Pre order oraz wypisanie poddrzewa o danym kluczu zostało zrealizowane rekurencyjnie, zgodnie z zasadą przeszukiwania: korzeń -> trawersuj lewe poddrzewo -> trawersuj prawe poddrzewo.

```
#dla AVL I BST
def pre_order(self, wierzcholek):
   if wierzcholek:                 #pre-order: korzeń, lewe
poddrzewo, prawe poddrzewo
       if wierzcholek.dane:       #KORZEŃ
           print(str(wierzcholek.dane), end='-')
           self.pre_order(wierzcholek.lewe_dziecko)  #LEWE
           self.pre_order(wierzcholek.prawe_dziecko)  #PRAWE

def pre_order_subtree(self, wierzcholek, klucz):
   if wierzcholek is None:  #warunek kończący rekurencje
       return
```

```
    if wierzcholek.dane == klucz:
        print("Pre-order poddrzewa o korzeniu", klucz, ":")
        self.pre_order(wierzcholek)  # Wywołana metoda pre_order na 
korzeniu poddrzewa o danym kluczu
        return

    # SZUKANIE - Rekurencyjnie wywołanie pre_order_subtree dla 
lewego i prawego poddrzewa
    self.pre_order_subtree(wierzcholek.lewe_dziecko, klucz)
    self.pre_order_subtree(wierzcholek.prawe_dziecko, klucz)
```

DODATKOWO W KODZIE SĄ INNE FUNKCJE POMOCNICZE TAKIE JAK WYŚWIETL, LEVEL TRAVERSAL, WYSWIETL_ WSPOCZYNNIK_ROWNOWAGI.

SEKCJA 2

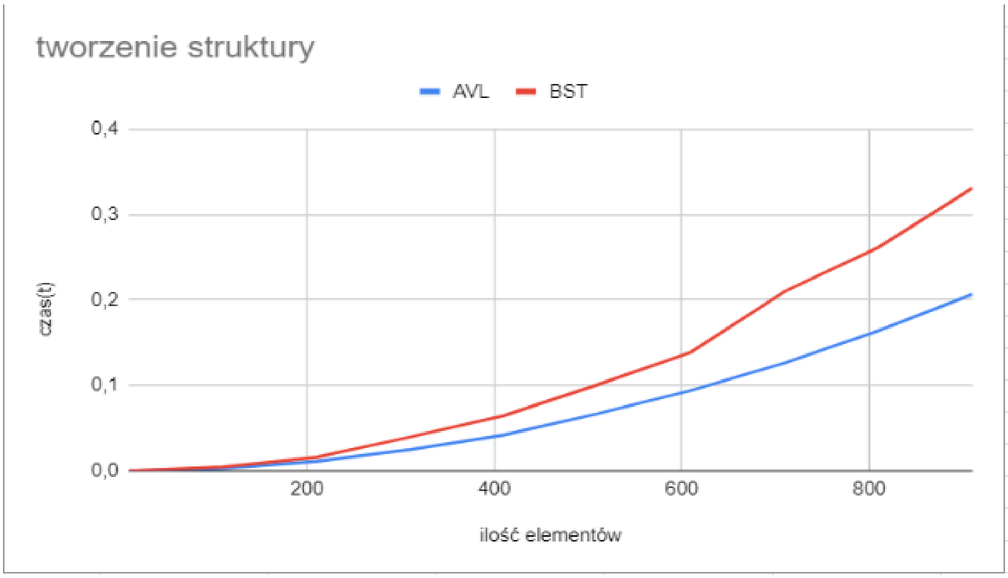
Wykres zależności czasu tworzenia struktury od danych wejściowych:
dla liczby elementów z zakresu 10-910

AVL

10	110	210	310	410	510	610	710	810	910	TEST
4,1E-05	0,003105700016	0,01121050003	0,02477379999	0,04224689997	0,06708130002	0,0940096	0,1262635	0,1646433	0,2075147	1
4,16E-05	0,00307799998	0,01086849999	0,02458339999	0,04042950005	0,07361020002	0,09391380002	0,1247725	0,1634218	0,201327	2
4,19E-05	0,003139399982	0,01123449998	0,02509459999	0,04240800004	0,06662589998	0,09549959999	0,1261729	0,1663494	0,2043734	3
4,34E-05	0,003093200037	0,01102639997	0,0248062	0,04184400005	0,06525680004	0,0914565	0,125713	0,1620159	0,2031041	4
4,17E-05	0,00310989999	0,01094200002	0,0246071	0,0424562	0,0660388	0,09486960003	0,1278605	0,1631295	0,2182178	5
4,22E-05	0,003101999988	0,01117359998	0,02484129998	0,04214209999	0,06438489998	0,09406040004	0,1265521	0,16412	0,2112295	6
4,22E-05	0,003166499955	0,01136359997	0,02524789999	0,04344589997	0,0667114	0,09525880002	0,1286923	0,1660733	0,2077381	7
4,25E-05	0,003114500025	0,0108399	0,02463070001	0,04199229996	0,06576570001	0,09466850001	0,1257058	0,1632006	0,198889	8
0	0,003113649997	0,01108237499	0,02482312499	0,0421206125	0,06693437501	0,09421710001	0,126466575	0,164119225	0,2065492	SREDNIA

BST

10	110	210	310	410	510	610	710	810	910	TEST
4,20E-05	0,003761200001	0,01681659999	0,04199320002	0,05737900001	0,1041096	0,1244722	0,1610304	0,2232065	0,300539	1
4,16E-05	0,004538000037	0,0129116	0,0455217	0,05736529996	0,07748219999	0,1813839	0,1804096	0,2050851	0,2611077	2
5,36E-05	0,004788100021	0,01788449998	0,04154479998	0,04984480003	0,1028228	0,1274864	0,212049	0,3558018	0,2628498	3
4,11E-05	0,006567300006	0,01919329999	0,03988509998	0,1126968	0,1164209	0,1639337	0,2528989	0,3003014	0,4130317	4
4,39E-05	0,00351369998	0,0159615	0,03500480001	0,0512098	0,1104757	0,1392967	0,2244439	0,2560761	0,3091511	5
4,59E-05	0,003928199993	0,0139583	0,04763260001	0,05178060004	0,08305979997	0,1127682	0,2395894	0,2875456	0,2657614	6
4,03E-05	0,006026799965	0,01483499998	0,03149859997	0,08027859998	0,0881314	0,1372571	0,2221368	0,2386576	0,3545175	7
6,10E-05	0,00587459997	0,0144015	0,03420150001	0,0560637	0,1237315	0,1252566	0,1889717	0,2260946	0,479084	8
0	0,004874737497	0,01574528749	0,0396602875	0,06457732501	0,1007792375	0,13898185	0,2101912125	0,2615960875	0,330755275	SREDNIA



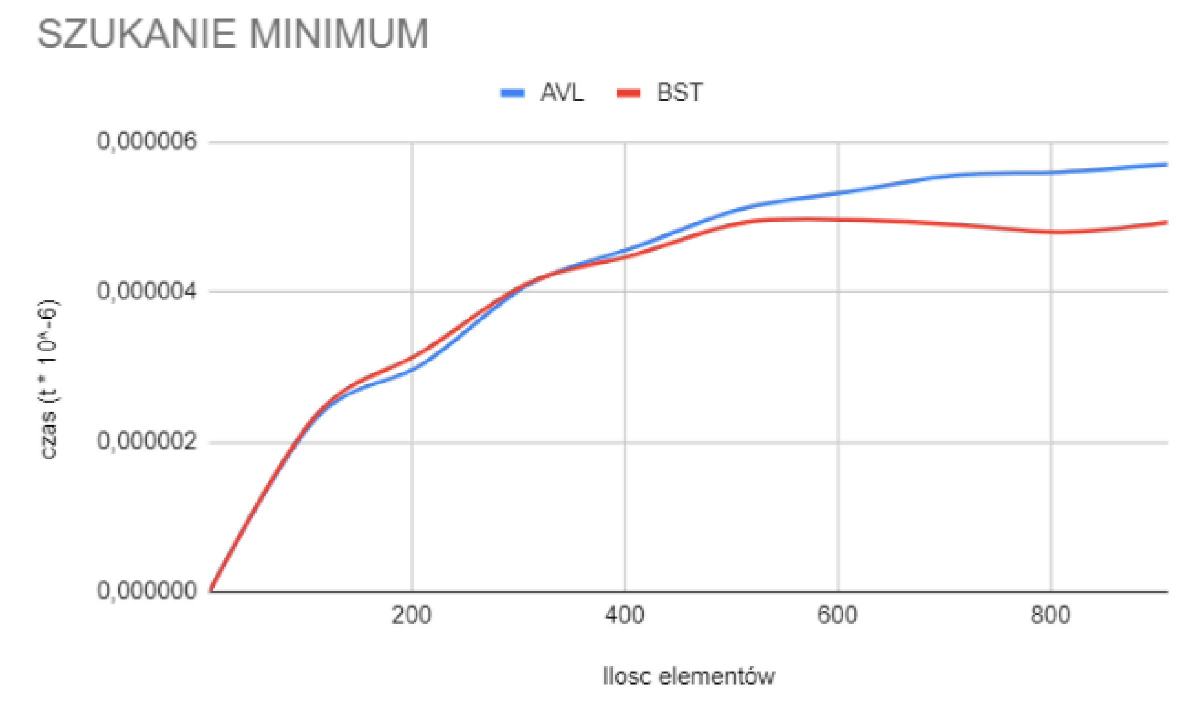
Wykres zależności czasu szukania minimum od danych wejściowych:
dla liczby elementów z zakresu 10-910

AVL

10	110	210	310	410	510	610	710	810	910	TEST
1,30E-06	1,80E-06	2,90E-06	4,00E-06	4,50E-06	4,10E-06	5,90E-06	5,30E-06	4,90E-06	5,60E-06	1
1,40E-06	1,80E-06	4,40E-06	3,60E-06	3,00E-06	5,40E-06	5,40E-06	5,20E-06	4,20E-06	5,60E-06	2
1,40E-06	1,90E-06	2,40E-06	3,90E-06	3,30E-06	4,80E-06	4,80E-06	5,20E-06	4,80E-06	5,70E-06	3
1,40E-06	1,50E-06	1,50E-06	2,90E-06	4,60E-06	4,40E-06	5,10E-06	6,30E-06	5,60E-06	6,40E-06	4
1,90E-06	4,40E-06	4,90E-06	4,10E-06	5,80E-06	5,00E-06	5,00E-06	6,80E-06	5,60E-06	7,10E-06	5
1,50E-06	2,20E-06	3,10E-06	4,30E-06	4,10E-06	4,80E-06	4,60E-06	5,40E-06	4,90E-06	5,30E-06	6
1,30E-06	1,90E-06	2,00E-06	1,96E-05	2,60E-06	5,70E-06	4,60E-06	5,40E-06	5,10E-06	5,20E-06	7
1,40E-06	3,40E-06	3,10E-06	4,20E-06	6,20E-06	5,40E-06	5,10E-06	5,30E-06	5,30E-06	4,90E-06	8
1,60E-06	1,90E-06	3,00E-06	7,50E-06	5,50E-06	6,40E-06	7,50E-06	5,10E-06	5,20E-06	5,50E-06	9
0	2,31E-06	3,03E-06	4,10E-06	4,60E-06	5,11E-06	5,33E-06	5,56E-06	5,60E-06	5,70E-06	SREDNIA

BST

10	110	210	310	410	510	610	710	810	910	TEST
2,10E-06	2,10E-06	2,90E-06	3,50E-06	4,10E-06	3,80E-06	4,10E-06	5,10E-06	4,50E-06	5,90E-06	1
2,00E-06	2,20E-06	2,90E-06	4,80E-06	3,40E-06	4,70E-06	5,10E-06	3,50E-06	4,70E-06	4,20E-06	2
1,80E-06	2,60E-06	3,00E-06	2,90E-06	4,40E-06	4,70E-06	4,10E-06	4,50E-06	4,80E-06	4,10E-06	3
1,80E-06	1,20E-06	1,40E-06	4,00E-06	2,50E-06	5,10E-06	4,20E-06	4,50E-06	5,00E-06	5,10E-06	4
2,20E-06	4,60E-06	4,60E-06	3,60E-06	5,10E-06	4,60E-06	4,80E-06	4,40E-06	5,90E-06	5,00E-06	5
2,10E-06	1,90E-06	2,70E-06	4,30E-06	4,50E-06	4,10E-06	5,10E-06	5,10E-06	4,40E-06	5,30E-06	6
2,20E-06	2,00E-06	2,60E-06	6,00E-06	3,40E-06	5,70E-06	6,10E-06	5,10E-06	3,90E-06	4,20E-06	7
1,70E-06	2,30E-06	3,40E-06	3,60E-06	5,60E-06	4,80E-06	5,20E-06	5,10E-06	5,00E-06	5,00E-06	8
2,10E-06	2,30E-06	3,20E-06	4,40E-06	4,30E-06	6,80E-06	6,00E-06	6,80E-06	5,00E-06	5,60E-06	9
0	2,36E-06	3,20E-06	4,12E-06	4,50E-06	4,92E-06	4,97E-06	4,90E-06	4,80E-06	4,93E-06	SREDNIA



Złożoność obliczeniowa szukania minimum w drzew AVL i BST wynosi O(logn) dla średnich przypadków. W przypadku pesymistycznym złożoność dla AVL nie zostaje zmieniona, jednakże w BST przez rozpatrywanie winorośli której liczba elementów wynosi n, złożoność obliczeniowa wynosi O(n)

Wykres zależności czasu wypisania pre-order od danych wejściowych:
dla liczby elementów z zakresu 10-910

AVL										
10	110	210	310	410	510	610	710	810	910	TEST
2,71E-05	0,000263800029	0,000487099983	0,000921999977	0,000980800017	0,001185300003	0,001412299986	0,001895599999	0,002093500002	0,002580400032	1
2,77E-05	0,000265300041	0,000509899982	0,000747699989	0,001141699962	0,001172900025	0,001392500009	0,00164859998	0,001866000006	0,002006200026	2
2,77E-05	0,000257899984	0,000489400001	0,000862199987	0,000966499967	0,001205699984	0,001404699986	0,001568100008	0,001882200013	0,002622300002	3
2,75E-05	0,000258499989	0,000490899954	0,000714100024	0,000955099996	0,001340500021	0,001407400006	0,001625099976	0,002618699975	0,002725400031	4
2,77E-05	0,000254300015	0,000491899962	0,000720800017	0,000942900020	0,001110600017	0,001426900038	0,001904399993	0,002252599981	0,002033199999	5
2,60E-05	0,000242899986	0,000506000011	0,000718299998	0,000999499985	0,00116429996	0,00133410003	0,001608800027	0,002223700052	0,002106300031	6
2,74E-05	0,000254500017	0,000502699986	0,000823999987	0,000956399948	0,001178200007	0,001531800018	0,002066200017	0,001846199972	0,0020793	7
2,85E-05	0,000254200014	0,000495499989	0,000806999974	0,000947099993	0,001166199974	0,001396300038	0,001922500014	0,002158899966	0,002091999981	8
0	0,000256425009	0,000496674983	0,000789512494	0,000986249986	0,001190462499	0,001531800018	0,001779912502	0,002093500002	0,002280637513	ŚREDNIA

BST

10	110	210	310	410	510	610	710	810	910	TEST
4,68E-05	0,000257100036	0,000577600032	0,0007415	0,000960400037	0,0015059	0,001415800012	0,002662899962	0,001860699966	0,002003600006	1
5,43E-05	0,000270600023	0,000506499956	0,000777499983	0,001510400034	0,001231300004	0,001739800035	0,001660800015	0,002302100009	0,002111400012	2
4,56E-05	0,000265500042	0,000499700021	0,000924399995	0,00173769996	0,00121680001	0,001883899968	0,001676900021	0,001868300023	0,002104099956	3
4,65E-05	0,000282499997	0,000498100009	0,000876799982	0,001199699997	0,001195999968	0,0016818	0,002006999974	0,002060699985	0,002100399986	4
4,60E-05	0,000260800006	0,000484200019	0,000732699991	0,000958499964	0,001198699989	0,00142429996	0,001637300011	0,001863600046	0,002486099994	5
4,59E-05	0,000247400021	0,000494599982	0,000741399999	0,000965700019	0,001194400014	0,001413599995	0,001635000051	0,001874400012	0,00201180001	6
4,77E-05	0,000257299980	0,000492000021	0,000767800025	0,000969400047	0,001154700003	0,001441200031	0,001649700047	0,001896100002	0,002570699959	7
4,76E-05	0,000258499989	0,000491000013	0,000748499995	0,000925300002	0,00118349999	0,00135700003	0,001647200028	0,001895699999	0,002496000011	8
0	0,000262462512	0,000505462507	0,000788824996	0,000986249986	0,001235162497	0,001544675004	0,001822100014	0,002060699985	0,002235512492	ŚREDNIA

Ze względu na to, że AVL i BST ma tyle samo wierzchołków (niezależnie w których dzieciach się znajdują), złożoność wynosi O(n), i jest zależna tylko od liczby wierzchołków.

Wstawianie nowego węzła w drzewie AVL ma złożoność czasową O(logn), gdzie n to liczba wierzchołków w drzewie. Jest to spowodowane koniecznością utrzymywania równowagi drzewa poprzez wykonywanie rotacji/ lub odpowiednich usuwań węzłów. Złożoność pamięciowa operacji wstawiania w drzewie AVL jest również O(logn), ponieważ każde wstawianie wymaga rekurencyjnego wykonania operacji na wysokości drzewa która jest ograniczona przez logn. Wstawianie nowego węzła w drzewie ma złożoność czasową O(h), gdzie h to wysokość. W najgorszym przypadku, gdy drzewo jest winoroślą, wysokość wynosi O(n), wtedy złożoność obliczeniowa wstawiania elementu wynosi O(n). Złożoność pamięciowa operacji wstawiania w drzewie BST jest również O(h), ale w przeciwieństwie do drzewa AVL, wysokość nie jest “uwarunkowana” żadnym wzorem, więc w najgorszym przypadku też jest to O(n).

SEKCJA 2

Równoważenie drzewa zostało zrealizowane przez usuwanie korzenia. Najpierw szukany jest w kolejności level order pierwszy wierzchołek który ma współczynnik równowagi większy od 1 i następuje jego usunięcie. Dzięki modyfikacji w _nastepnik (funkcja usuwania) zawsze wybrany następnik jest z dziecka usuwanego wierzchołka które posiada większą wysokość. W ostatnim etapie balansowania drzewa, usunięty element jest na nowo dodawany do drzewa.

```
def rownowaz_drzewo(self):
   while True:
       # Sprawdzanie czy drzewo jest puste
       if self.korzen is None:
           print("Drzewo jest puste.")
           return

       # zmienna przechowująca pierwszy niezbalansowany węzeł
       niezbalansowany_wezel = self.level_order_traversal()

       # Sprawdzanie czy znaleziono niezbalansowany węzeł
       if niezbalansowany_wezel is None:
           print("Drzewo jest zrównoważone.")
           return

       # Usuwanie niezbalansowanego węzeł
       self.usun(niezbalansowany_wezel)

       # Dodawanie usuniętego węzeła ponownie do drzewa
       self.dodaj(niezbalansowany_wezel)

       # Wyświetlanie współczynnika równowagi po równoważeniu
drzewa
       self.wyswietl_wspolczynnik_rownowagi()
```

Złożoność obliczeniowa tej metody zależy od głębokości drzewa i równomierności rozkładu węzłów. W najlepszym przypadku, gdy drzewo jest zrównoważone złożoność może wynieść O(logn), gdzie n to liczba węzłów w drzewie. Jednak w najgorszym przypadku, gdy drzewo jest niezrównoważone złożoność może być O(n), gdy trzeba przeprowadzić rekurencyjne równoważenie drzewa. Metoda ta jest lepsza dla drzew, które są względnie zrównoważone, lub złożoność czasowa jest mniej istotna niż prostota implementacji. Jednak w bardziej wymagających scenariuszach zaleca się użycie bardziej zaawansowanych technik równoważenia, takich jak rotacje.

Wykres zależności czasu balansowania drzewa BST od danych wejściowych:

liczba elementów						
10	60	110	160	210	260	TESTY
0,000287300034	0,04440869996	0,2165291	0,774458	1,4685539	3,1101145	1
0,000231999962	0,1125077	0,511737	0,8632247	1,8316075	3,0005935	2
0,000293999968	0,03547619999	0,7335148	0,6140044	1,1824951	1,9433688	3
0,000369400018	0,02205829998	0,2103124	0,9381556	1,8396454	3,11445	4
0,000360400008	0,06591480004	0,1934729	1,0762324	2,2266031	1,6503926	5
0,000156900030	0,1199812	0,5561817	0,4027555	1,3325113	2,3199346	6
0	0,06672448334	16,0602497	23,52411866	31,4116309	39,30555057	ŚREDNIA