Gdańsk, 25.01.2022 r.

POLITECHNIKA GDAŃSKA

WYDZIAŁ FIZYKI TECHNICZNEJ I MATEMATYKI STOSOWANEJ

Sprawozdanie nr 3

Algorytmy i Struktury Danych
Laboratorium Komputerowe

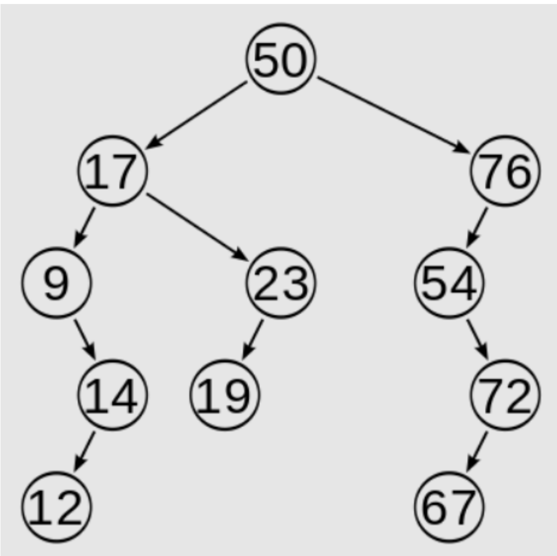
Algorytm: Szukanie w binarnym drzewie wyszukiwawczym (z rekursją)
Student: Aleksander Kruczkowski, 185390
Semestr: III
Stopień studiów: I
Kierunek studiów: Fizyka Techniczna
Specjalność: Informatyka Stosowana
Rok akademicki: 2021/2022

1. Wstęp teoretyczny

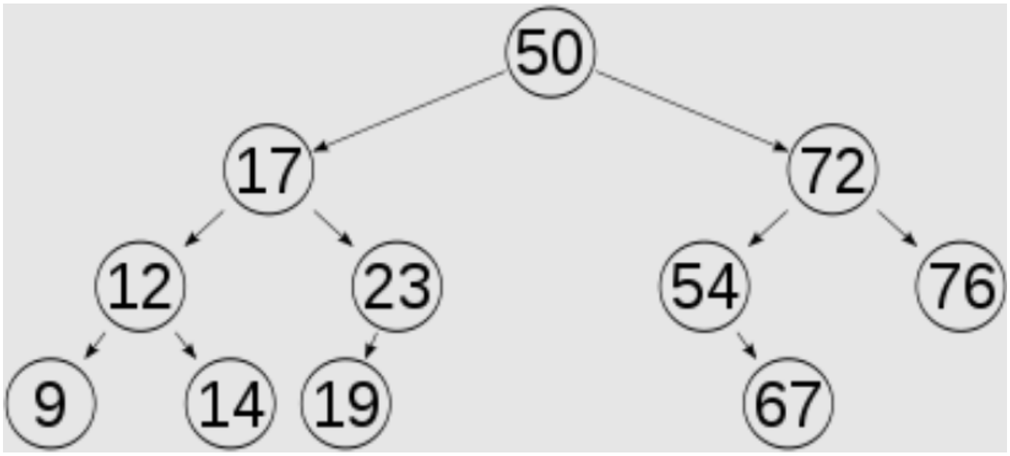
1.1. Informacje ogólne

Opracowywany algorytm odnajduje wybrane wartości znajdujące się w binarnym drzewie wyszukiwawczym.
Binarne drzewo poszukiwań (z ang. BST - Binary Search Tree) to struktura danych będąca drzewem binarnym, w którym lewe poddrzewo każdego z węzłów zawiera wyłącznie elementy o kluczach mniejszych niż klucz tego węzła, natomiast prawe poddrzewo zawiera wyłącznie elementy o kluczach nie mniejszych niż klucz węzła.

Przykładowe binarne drzewo wyszukiwawcze:



Złożoność obliczeniowa wyszukiwania wartości w BST zależy od wysokości drzewa (tj. długości najdłuższej ścieżki od korzenia do liści). Wysokość jest powiązana ze stopniem zrównoważenia drzewa, czyli różnicą w ilości węzłów w lewych i prawych poddrzewach BST. Powyżej zaprezentowano przykład niezrównoważonego drzewa. Jeśli byłoby zrównoważone, wyglądałoby tak:



Stopień zrównoważenia (a zarazem wysokość) drzewa binarnego zależy od kolejności umieszczania w nim elementów. W najlepszym wypadku wysokość drzewa wynosi **lg(N)**, gdzie **N** to liczba węzłów (elementów) w drzewie. W najgorszym przypadku, czyli gdy kolejne wartości umieszczane w drzewie mają tylko prawe (lub tylko lewe) poddrzewa, wysokość BST wynosi **N**.

Złożoność obliczeniowa operacji na binarnym drzewie wyszukiwań jest opisywana taką samą zależnością, jak jego wysokość, czyli w pesymistycznym przypadku wyszukiwanie w BST zajęłoby **O(N)**. Średnia złożoność jest opisywana taką samą funkcją jak złożoność optymistyczna, czyli **O(lg(N))**.

1.2. Sposób działania, pseudokod

Algorytm korzysta z konstrukcji BST - jeśli poszukiwany element jest mniejszy od klucza bieżącego węzła, trzeba go szukać w lewym poddrzewie tego węzła, a jeśli jest większy, to w prawym.

```
function BST_Search(node, key)
    if node = NIL or key = node.key then    (Ad. 1)
        return node                         (Ad. 2)
    if key < node.key then           (klucz mniejszy - szukaj na lewo)
        return BST_Search(node.left, key)
    else                             (klucz większy - szukaj na prawo)
        return BST_Search(node.right, key)

(Ad. 1) - (jeśli węzeł pusty, to w drzewie nie ma klucza)
(Ad. 2) - (jeśli węzeł zawiera klucz, znaleziono klucz)
```

2. Opis implementacji

2.1. Algorytm zaimplementowany z użyciem Python

Żeby można było opisać algorytm, zaimplementowano drzewo binarne z wykorzystaniem klas.

```
class Node:
    def __init__(self, data = None):
        self.data = data
        self.left = None
        self.right = None

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

Poniżej przedstawiono kod szukania w binarnym drzewie poszukiwań. Skorzystano z funkcji `find` inicjującej właściwą funkcję wyszukiwania `_find` w celu polepszenia wygody pracy nad badaniem algorytmu.

```
def find(self, data):
      if self.root:
          is_found = self._find(data, self.root)
          if is_found:
              return is_found
          return False
      else:
          return None

def _find(self, data, cur_node):
      if data > cur_node.data and cur_node.right:
          return self._find(data, cur_node.right)    (Ad. 1)

      elif data < cur_node.data and cur_node.left:  (Ad. 2)
          return self._find(data, cur_node.left)

      if data == cur_node.data:                     (Ad. 3)
          return cur_node
```

```
(Ad. 1) - (jeśli szukana wartość jest większa, niż znajdująca się
          w obecnie porównywanym węźle, wywołaj _find od prawego węzła)
(Ad. 1) - (analogicznie do Ad. 1)
(Ad. 1) - (jeśli obecnie porównywany węzeł zawiera poszukiwany klucz,
          zwróć adres tego węzła)
```

Funkcja wyszukująca `_find` jest praktycznie identycznym odtworzeniem wyżej podanego pseudokodu. Przekazuje otrzymany adres węzła do funkcji inicjującej `find` w przypadku odnalezienia poszukiwanego klucza. Jeśli klucz nie zawiera się w przeszukiwanym drzewie, zmienna `is_found` będzie pusta, co spowoduje zwrócenie wyniku `False` oznaczającego brak danej wartości w drzewie.

2.2. Funkcja weryfikująca skuteczność algorytmu

W celu sprawdzenia, czy opracowywana funkcja wyszukiwania zwraca poprawną wartość, w podobny sposób do badanego algorytmu zaimplementowano funkcję przechodzącą po każdym węźle w drzewie.

```
def verify(self, data):
    if self.root:
        self._verify(data, self.root)
        try:
            if self.tmp:
                return self.tmp
            return False
        except AttributeError:
            return False
    else:
        return None
def _verify(self, data, cur_node):
    if cur_node:
        self._verify(data, cur_node.left)
        self._verify(data, cur_node.right)

        if cur_node.data == data:
            print('verifier printout', cur_node.data)
            self.tmp = cur_node
            return self.tmp
        else:
            return False
```

Tak jak w implementacji badanego algorytmu, tutaj również skorzystano z funkcji inicjującej właściwe operacje. Porównanie klucza poszukiwanego z tym znajdującym się w obecnie odwiedzonym węźle przebiega dla każdego z węzłów. W momencie znalezienia właściwego, jego adres jest przekazywany do inicjatora. Dodatkowo dla komfortu pracy wyświetlano klucz w przypadku udanego sondowania jako potwierdzenie skuteczności dla użytkownika.

3. Opis testów

3.1. Rodzaje testów

- przypadek pesymistyczny - praca na danych rozmieszczonych w binarnym drzewie wyszukiwawczym w najmniej korzystny sposób, czyli korzeń ma węzeł tylko z jednej strony (drzewo zdegenerowane do listy),
- przypadek średni/losowy - praca na danych generowanych pseudolosowo, czyli BST nie będzie doskonale zrównoważone,
- przypadek optymistyczny - drzewo doskonale zrównoważone.

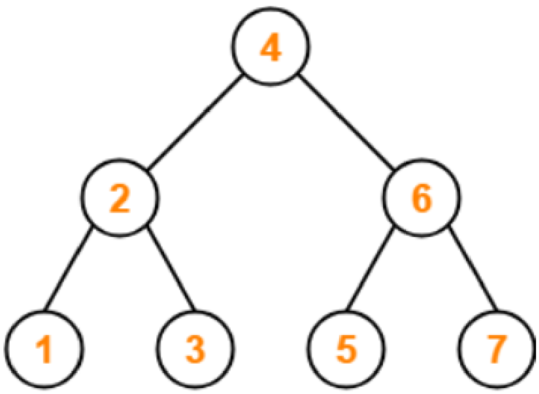
Dla tych przypadków wykonano po 12 prób na BST o 10 różnych rozmiarach. Odpowiednio **N** = [10, 50, 100, 250, 500, 750, 1000, 1500, 2000, 3000]. Wszystkie testy przeprowadzono w możliwie podobnych warunkach obciążenia maszyny.
W przypadku optymistycznym i pesymistycznym za każdym razem poszukiwano najwyższego klucza w drzewie. W przypadku losowym poszukiwano losowej wartości.

3.2. Złożoność obliczeniowa

Zależy ona od rozkładu wartości w binarnym drzewie wyszukiwawczym. Im bardziej jest zrównoważone, tym skuteczniejsze będzie wyszukiwanie.

Złożoność optymistyczna

Jeśli BST jest doskonale zrównoważone ilość koniecznych sondowań jest zminimalizowana do wysokości drzewa, która jest równa zaokrągleniu w górę **lg(N)**.



Jak widać na powyższym przykładzie, maksymalna ilość sondowań w tym drzewie w celu znalezienia któregoś z kluczy jest równa **3**, czyli **ceil(lg(N))**.

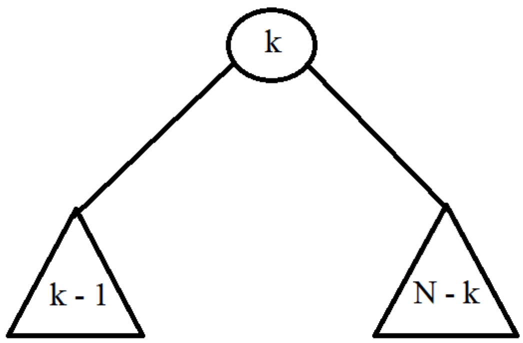
Złożoność średnia

Zależy ona jak wygląda losowe drzewo, na którym pracuje algorytm. Jeśli jest bliższe drzewu zdegenerowanemu, złożoność będzie zbliżona do liniowej **O(N)**, natomiast jeśli jest bliższe drzewu doskonale zrównoważonemu, złożoność będzie określona jako **O(lg(N))**.

W celu określenia średniej złożoności obliczeniowej analizę rozpoczęto od określenia średniej liczby prób przy sukcesie wyszukiwania. Następnie przy porażce.

Analiza w przypadku wyszukiwania zakończonego sukcesem

Przyjmując, że do pustego BST wstawiano w losowej kolejności klucze o wartościach **1, 2, …, N (N >= 1)**. Jako pierwszy klucz określono **k**. Wówczas drzewo ma taką postać:



W korzeniu znajduje się klucz **k**, lewe poddrzewo zawiera **k - 1** kluczy, a prawe poddrzewo zawiera **N - k** kluczy.
Liczba dokonanych przy poszukiwaniu sondowań jest równa poziomowi węzła zawierającemu żądany klucz.
Przyjmując oznaczenia:
$\mathbf{A_N}$ - średnia suma poziomów wszystkich węzłów w drzewie BST o **N** węzłach,
$\mathbf{A_N^k}$ - $\mathbf{A_N}$ przy warunku, że w korzeniu drzewa znajduje się klucz **k**,
$\mathbf{a_N}$ - średni poziom węzła w drzewie BST o N węzłach.

W powyższym drzewie poziom korzenia jest równy 1. W obu poddrzewach poziom każdego węzła jest o jeden większy niż poziom liczony wewnątrz danego poddrzewa. Stąd średnia suma poziomów w lewym poddrzewie wynosi $\mathbf{A_{k-1} + k - 1}$, a w prawym poddrzewie $\mathbf{A_{N-k} + N - k}$. Zatem:

$$A_N^k = 1 + A_{k-1} + k - 1 + A_{N-k} + N - k$$

$$A_N^k = A_{k-1} + A_{N-k} + N$$

Wszystkie permutacje wstawianych kluczy są jednakowo prawdopodobne, więc każdy klucz ma jednakową szansę zostania korzeniem. Stąd, uśredniając
po wszystkich możliwych wartościach **k**, otrzymano:

$$A_N = \frac{1}{N}\sum_{k=1}^{N}\left(A_N^k\right) = N + \frac{1}{N}\sum_{k=1}^{N}\left(A_{k-1} + A_{N-k}\right)$$

Pod znakiem sumy w powyższym równaniu występują wyrazy parami równe. Biorąc to pod uwagę, można uzyskać równanie rekurencyjne:

$$\begin{cases} A_N = N + \frac{2}{N}\sum_{k=0}^{N}\left(A_k\right), N \geq 1 \\ A_0 = 0 \end{cases}$$

Rozwiązaniem tego równania jest:

$$A_N = 2(N+1)H_N - 3N$$

Gdzie $\mathbf{H_N = 1 + 1/2 \ + 1/3 + ... + 1/N}$
Prawdopodobieństwo poszukiwania każdego z kluczy jest równe **1/N**, zatem średnia liczba sondowań jest równa średniemu poziomowi węzła w drzewie, czyli $\mathbf{a_n = A_N / N}$
Zgodnie z powyższym równaniem $\mathbf{a_n}$ można przedstawić

$$a_N = \frac{2(N+1)}{N} H_N - 3$$

Korzystając z wzoru na oszacowanie n-tego wyrazu ciągu harmonicznego H_N:

$$\sum_{k=1}^{n}\frac{1}{k} = \ln n + \gamma + O\left(\frac{1}{n}\right), \gamma = 0{,}577...$$

Otrzymano następujące przybliżenie wartości $\mathbf{a_N}$:

$$a_N \approx 2\ ln(N) + 2\gamma - 3 = 2\ ln(2)\ lg(N) + 2\gamma - 3$$

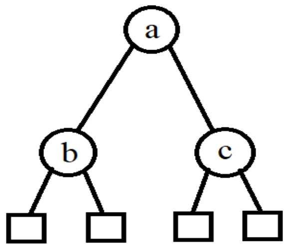
$$a_N = 2\ ln(2)\ lg(N) - O(1)$$

Średnią liczbę prób przy sukcesie można więc przybliżyć przez:

$$S_{śr} \approx 2\ ln(2)\ lg(N) \approx\ 1{,}386\ lg(N)$$

<u>Analiza w przypadku wyszukiwania zakończonego porażką</u>

Przy wyszukiwaniu zakończonym porażką ostatnim położeniem sprawdzanym położeniem w drzewie jest węzeł wewnętrzny, który może być ojcem węzła zewnętrznego. Zaprezentowano to na poniższej ilustracji drzewa z dodanymi węzłami zewnętrznymi.



Przyjęto oznaczenia:
B_N - średnia suma poziomów węzłów zewnętrznych
b_N - średni poziom węzła zewnętrznego w losowym drzewie o **N** węzłach wewnętrznych
Żeby wyznaczyć powyższe wartości wykorzystano zależność między sumą poziomów wewnętrznych i zewnętrznych w dowolnym drzewie binarnym:

$$B_N = A_N + 2N + 1$$

Korzystając z tego oraz ze wzoru otrzymanego w poprzedniej analizie:

$$A_N = 2(N+1)H_N - 3N$$

Otrzymano:

$$B_N = 2(N+1)H_N - N + 1$$

Liczba węzłów zewnętrznych w drzewie o **N** węzłach zewnętrznych wynosi **N + 1**. Skoro dotarcie do każdego z węzłów zewnętrznych jest jednakowo prawdopodobne, to **b_N = B_N / (N + 1)**, czyli:

$$b_N = 2\,H_N - \frac{N-1}{N+1}$$

Stosując przybliżenia analogiczne jak poprzednio, uzyskano:

$$b_N \approx 2\,ln(N) \;+\; 2\gamma - 1 = 2\,ln(2)\;lg(N) + O(1)$$

$$S_{śr} \approx 1{,}386\;lg(N)$$

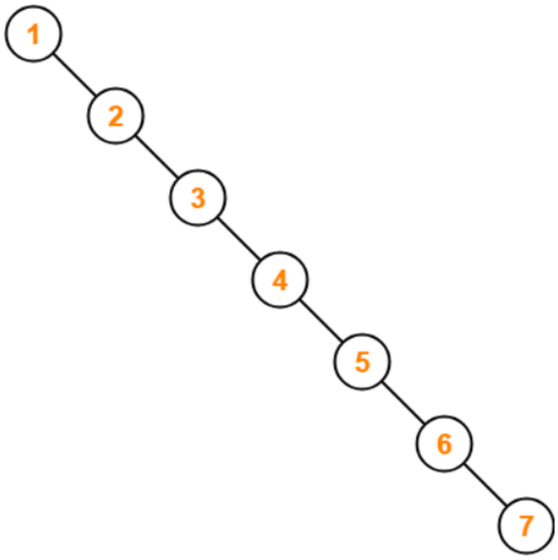
Z powyższych analiz wynika, że liczba sondowań jest bardzo zbliżona niezależnie od powodzenia operacji wyszukiwania. W obu przypadkach dla drzewa losowego liczba operacji algorytmu wykazała zależność opisywalną przez **lg(N)**.

Podsumowując, średnia złożoność obliczeniowa wyszukiwania w BST to **O(lg(N))**.

Złożoność pesymistyczna

Binarne drzewo wyszukiwawcze zdegenerowane do listy ma wysokość **N**, przez co złożoność wyszukiwania opisuje się jako **O(N)**.

Jest to jasno widoczne na przykładzie takiego drzewa:



3.3. Przykładowe dane wejściowe

Dane używane do testów generowano następującymi sposobami:

- przypadek optymistyczny

```
def generate_optymist(tab):
    if not tab:
        return None

    mid = len(tab) // 2

    root = Node(tab[mid])
    root.left = generate_optymist(tab[:mid])
    root.right = generate_optymist(tab[mid + 1:])

    return root
```

- przypadek pesymistyczny

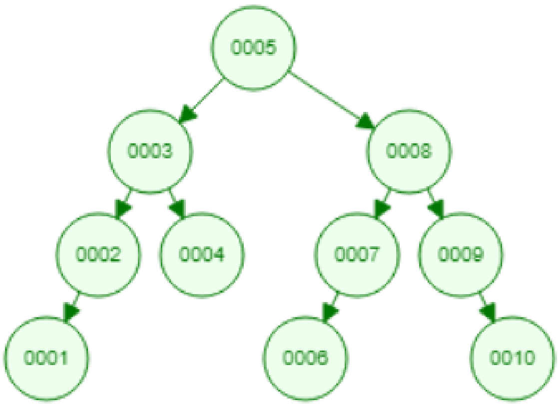
```
def generate_pesymist(n):
    tab = []
    for x in range(1, n + 1):
        tab.append(x)
    return tab
```

- przypadek średni

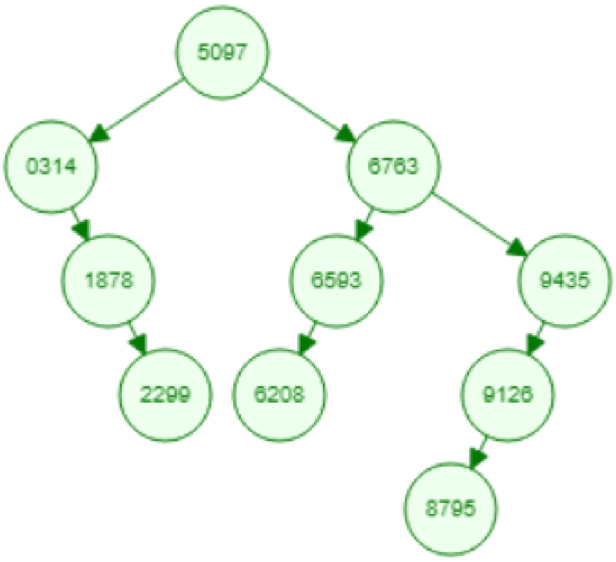
```
random.seed()
def generate_avg(n):
    with open('input.txt', 'w') as f:
        for x in range(0, n):
            rand = random.randint(0, 1000000)
            f.write(str(rand) + ', ')
```

Przykłady danych dla N = 10

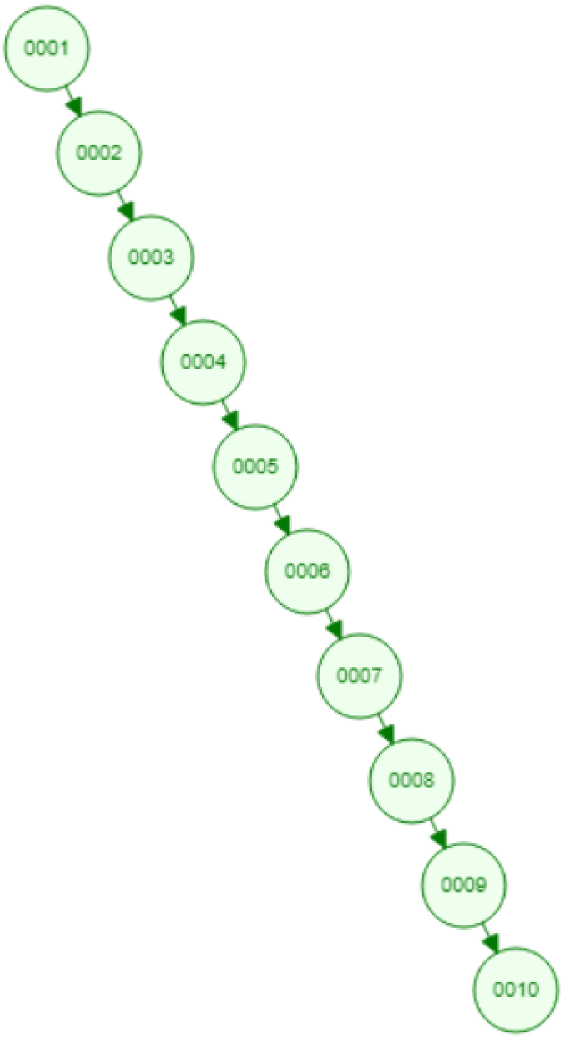
- optymistyczne - tablica wejściowa = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]



- losowe - tablica wejściowa = [5097, 6763, 314, 9435, 9126, 1878, 6593, 8795, 6208, 2299]



- pesymistyczne - tablica wejściowa = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]



3.4. Przykładowe dane zwrócone przez algorytm i weryfikator

Badana jest funkcja wyszukująca, która w żaden sposób nie zmienia przetwarzanych danych. Poniżej umieszczono komunikaty wydawane po wyszukaniu klucza = 3000.

```
C:\castel_gandolfo\conda\python.exe "E:/Pobrane/notatki_
function:  <__main__.Node object at 0x000001FAC07EF220>
verifier printout 3000
verifier:  <__main__.Node object at 0x000001FAC1B22940>

Process finished with exit code 0
```

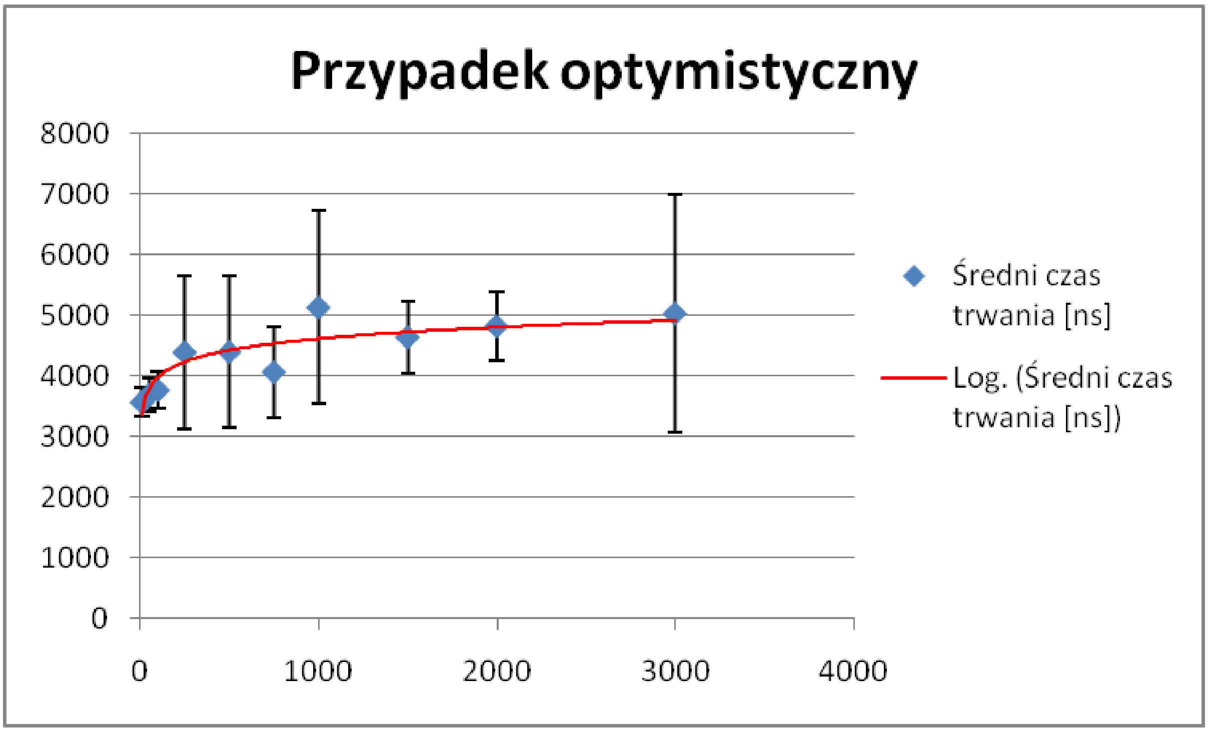
Zwrócone adresy są identyczne, co potwierdza skuteczność algorytmu.

4. Opracowanie wyników testów

4.1. Dane uzyskane w przypadku optymistycznym

L.p.	N	Średni czas trwania [ns]	Odchylenie standardowe
1	10	3566,67	230,94
2	50	3691,67	277,84
3	100	3766,67	302,51
4	250	4391,67	1260,2
5	500	4400	1257,7
6	750	4066,67	747,28
7	1000	5133,33	1599,05
8	1500	4641,67	590,01
9	2000	4825	556,16
10	3000	5033,33	1969

Wykres zależności czasu od N sporządzony na podstawie powyższych danych



Do wartości na wykresie dopasowano funkcję logarytmiczną o równaniu:

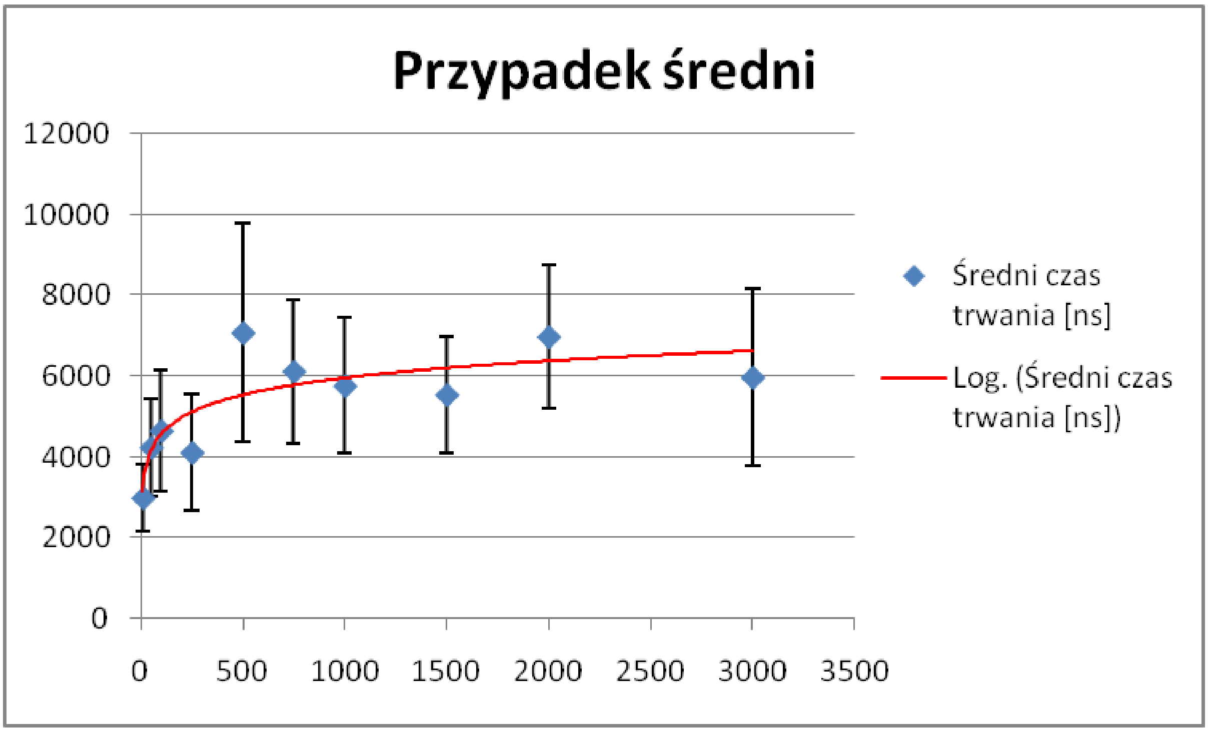
y = 1 * log(N) + 3926

Współczynnik wyznaczania $\mathbf{R^2}$ = **0,95**. Jest bardzo bliski **1**, co oznacza dobre dopasowanie funkcji.

4.2. Dane uzyskane w przypadku średnim

L.p.	N	Średni czas trwania [ns]	Odchylenie standardowe
1	10	2983,33	835,39
2	50	4233,33	1209,31
3	100	4650	1508,46
4	250	4108,33	1452,18
5	500	7083,33	2701,12
6	750	6125	1778,21
7	1000	5766,67	1668,6
8	1500	5541,67	1452,56
9	2000	6983,33	1767,81
10	3000	5975	2201,7

Wykres zależności czasu od N sporządzony na podstawie powyższych danych



Do wartości na wykresie dopasowano funkcję logarytmiczną o równaniu:

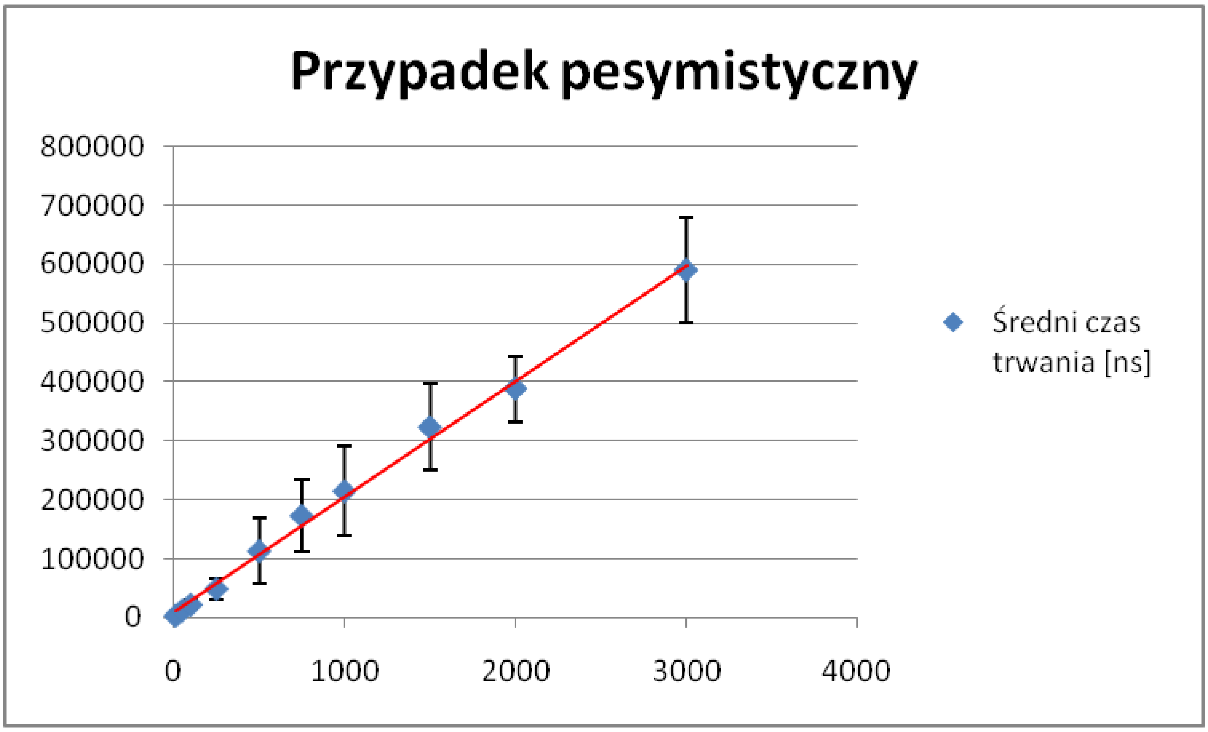
y = 1 * log(N) + 4461

Współczynnik wyznaczania $\mathbf{R^2}$ = **0,72**. Nie jest tak bliski **1** jak w przypadku optymistycznym, jednak nadal jest to najlepsze możliwe dopasowanie funkcji. Alternatywą byłoby dopasowanie funkcji liniowej, które wykazuje się bardzo niską korelacją.

4.2. Dane uzyskane w przypadku pesymistycznym

L.p.	N	Średni czas trwania [ns]	Odchylenie standardowe
1	10	3708,33	699,95
2	50	12825	4443
3	100	22525	8245,45
4	250	49408,33	17453,91
5	500	113166,67	55174,19
6	750	172950	60295,37
7	1000	214708,33	75985,77
8	1500	323116,67	72873,49
9	2000	388600	55892,66
10	3000	589258,33	89458,27

Wykres zależności czasu od N sporządzony na podstawie powyższych danych



Do wartości na wykresie dopasowano funkcję liniową o równaniu:
y = 196 * N + 9487
Współczynnik wyznaczania **R^2 = 0,99**. Jest to praktycznie idealne dopasowanie, co potwierdza liniową tendencję wzrostu czasu pracy względem ilości danych.

Wartości błędów pomiarowych wyznaczone przez odchylenie standardowe są dość znaczne w przypadku gdzie wystąpiła tendencja logarytmiczna. Dla przypadku liniowego są bardzo niskie. W każdym przypadku dopasowana krzywa mieści się w granicach błędów pomiarowych. Każdy pomiar dał przewidywany wynik.

5. Podsumowanie i wnioski

Dopasowanie złożoności obliczeniowej algorytmu do wyników pomiarowych jest akceptowalne. Krzywe naniesione na wykresy mieściły się w przedziałach błędów pomiarowych i charakteryzowały się wysoką korelacją z wynikami pomiarów.

W przypadku optymistycznym i średnim algorytm charakteryzuje się złożonością obliczeniową **O(lg(N))**. W przypadku pesymistycznym zmienia się na **O(N)**.

Podczas wyznaczania średniej złożoności obliczeniowej pokazano, że efektywność operacji wyszukiwania w drzewie losowym jest średnio gorsza o ok. 39 % niż w drzewie doskonale zrównoważonym, przy czym zachowana zostaje zależność logarytmiczna od liczby elementów drzewa.
Balansowanie drzewa i wyszukiwanie może być bardziej kosztowne, niż praca na losowym drzewie bez równoważenia (przez np. algorytm DSW, który podobnie do innych algorytmów równoważących BST charakteryzuje się złożonością **O(N)**). Jest to wysoce prawdopodobne dla operacji dodawania i usuwania elementów drzewa. Jeśli przeprowadzane będzie tylko wyszukiwanie, niewykluczone, że uprzednie zrównoważenie drzewa przyniosłoby korzyść. Niewątpliwie balansowanie byłoby również korzystne przy podejrzeniu wystąpienia najgorszego przypadku rozkładu kluczy w drzewie.

Wyszukiwanie z rekursją w binarnym drzewie wyszukiwawczym jest na ogół mniej wydajne niż taka sama operacja wykonana z pomocą iteracji. Rekursywne rozwiązania wymagają więcej pamięci, niż przy iteracji Są również wolniejsze od iteracyjnych z uwagi na konieczność kontroli nad stosem informacji o wykonanych operacjach tworzonym podczas kolejnych wywołań.
Zależnie od możliwości komputera i ilości danych ta różnica może być pomijalna, a zapis rekursywny niewątpliwie upraszcza (przede wszystkim skraca) kod.

6. Literatura

1. Krzysztof Goczyła *Struktury danych*, Wydawnictwo Politechniki Gdańskiej, 2002
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: *Wprowadzenie do algorytmów*. Wydawnictwa Naukowo-Techniczne, 2007. ISBN 978-83-204-3328-9. OCLC 749241843
3. https://en.wikipedia.org/wiki/Day%E2%80%93Stout%E2%80%93Warren_algorithm

Załączniki

1. folder skompresowany z kodem programu, spr_3_Kruczkowski.zip