SDiZO- Sprawozdanie

Zadanie projektowe nr 1

Temat: Badanie efektywności operacji dodawania, usuwania oraz wyszukiwania elementów w różnych strukturach danych.

Autor:	Władysław Nowak
Nr. albumu:	252700
Grupa Projektowa:	Poniedziałek, 11^{15}-13^{00} TP
Prowadzący:	Mgr inż. Antoni Sterna

Wstępne informacje dotyczące samego programu:

Program został wykonany w postaci **obiektowej**. Każda struktura danych, dodatkowe funkcje związane z działaniem programu zawarte są w odpowiednich klasach. Każda funkcja programu opisana jest przy pomocy bloków komentarzy zgodnych *Doxygen'em.*

Zawartość programu:

1. Struktury danych:
    1.1. Tablica dynamiczna
    1.2. Lista dwukierunkowa
    1.3. Kopiec binarny
    1.4. Drzewo poszukiwań binarnych
    1.5. Drzewo czerwono-czarne
2. Menu sterowania programem *(wersja konsolowa)*
3. Test działania zaimplementowanych struktur danych
4. Klasy pomocnicze odpowiedzialne za m.in.:
    4.1. Zapis wyników testów do pliku *.csv*
    4.2. Odczyt przykładowych danych z pliku
    4.3. Typy wyliczeniowe odpowiadające strukturom danych

Uproszczona struktura plików w projekcie:

```
├── include                    - Pliki nagłówkowe
└── src
    ├── Benchmark.cpp          - Funkcje testujące struktury
    ├── DataStructures         - Implementacja wszystkich struktury danych
    │   ├── BST.cpp
    │   ├── DoublyLinkedList.cpp
    │   ├── DynamicArray.cpp
    │   ├── Heap.cpp
    │   ├── RBTree.cpp
    │   └── helper-classes     - Klasy pomocnicze użyte w strukturach danych
    │       ├── BSTNode.cpp
    │       ├── BSTNode.h
    │       ├── DoublyLinkedNode.cpp
    │       ├── DoublyLinkedNode.h
    │       ├── RBNode.cpp
    │       └── RBNode.h
    ├── IO                     - Klasy odpowiedzialne za IO na plikach
    │   ├── CSVWriter.cpp
    │   ├── CSVWriter.h
    │   ├── FileReader.cpp
    │   └── FileReader.h
    ├── Menu.cpp               - Menu sterowania programem
```

Informacje dotyczące środowiska:

- Użyte środowisko automatycznego budowania: *CMake*
- Użyte IDE: *CLion*
- Dokumentacja generowana przy użyciu: *Doxygen*
- System kontroli wersji: *Git*

Kompletna dokumentacja wygenerowana na podstawie komentarzy znajduje się pod adresem: https://ultux.github.io/SDiZO-P/. W kodzie znajdują się dodatkowe komentarze dotyczące logiki implementowanych algorytmów.
Projekt został umieszczony na *GitHub'ie* pod adresem: https://github.com/ULTUX/SDiZO-P.

Opis struktur danych i ich implementacji

Strukturą danych nazywamy sposób przechowywania danych w pamięci komputera.

Tablica dynamiczna

Tablica dynamiczna zachowuje się podobnie jak struktura statyczna – można do niej wprowadzać dane, usuwać je i odczytywać. Jednak, gdybyśmy chcieli wprowadzić nową wartość do już zapełnionej tablicy statycznej to pojawiłby się błąd, w przeciwieństwie do tablicy dynamicznej. Dzieje się tak dlatego, że tablica statyczna ma stały rozmiar, a tablica dynamiczna automatycznie dostosowuje swój rozmiar do rozmiaru danych, które się w niej znajdują.
Wg źródeł złożoność obliczeniowa takiej struktury w zależności od operacji jest następująca:

- dodawanie/usuwanie na końcu – O(1) – element jest dodawany/usuwany z końca tablicy
- dodawanie/usuwanie na początku – O(n) – każdy element musi zostać przeniesiony o indeks
- wyszukiwanie/ustawianie elementu – O(1)

Zaimplementowana przeze mnie tablica w przypadku, gdy ma zostać zapełniona, zwiększa swoją wielkość 2-krotnie. Dzięki temu ilość alokacji i dealokacji pamięci jest mała, a algorytm wydajny – kosztem zajmowanej pamięci (w najgorszym przypadku jest to 2x faktyczna wielkość elementów w strukturze).

Lista dwukierunkowa

Lista dwukierunkowa jest strukturą, w której dane są przechowywane w połączonych ze sobą węzłach. Każdy węzeł, oprócz danej, posiada dwa wskaźniki – na następny węzeł i poprzedni. Dodawanie i usuwanie danych z początku jest dość proste – głowa listy jest odpowiednio ustawiana na nowy element. Jednak w przypadku, gdy chcemy dodać lub usunąć węzeł z końca listy zależnie od implementacji złożoności takich operacji mogą być różne. Jeśli lista przechowuje wskaźnik na ogon – operacja będzie symetryczna do operacji na początku listy, w przeciwnym wypadku wymagane jest wcześniejsze przejście po każdym elemencie takiej listy aż na sam koniec. Ja zdecydowałem się dokonać implementacji listy **bez wskaźnika na ogon.**
Złożoności obliczeniowe tablicy dwukierunkowej:

- dodawanie/usuwanie na początku – O(1)
- dodawanie/usuwanie na końcu – O(n)
- dodawanie/usuwanie w określonym miejscu – O(n)
- wyszukiwanie – O(n)

Kopiec binarny

Jest strukturą danych przechowywaną w tablicy przedstawiającą pełne drzewo binarne. Istnieją dwa rodzaje takich kopców – min i max. Rodzaj kopca jest definiowany przez relację między każdym rodzicem i jego dziećmi, dla kopca min wartość rodzica jest zawsze mniejsza niż wartość dzieci. Oznacza to, że korzeń ma wartość minimalną. W przypadku kopca max sytuacja jest odwrotna. Na kopcu można przeprowadzić operacje takie jak: dodawanie, usuwanie i wyszukiwanie (które tak naprawdę jest usuwaniem do momentu znalezienia). W zadaniu projektowym wymagane było przechowywanie kopca w tablicy dynamicznej.
Złożoności obliczeniowe kopca binarnego:

- Dodawanie – O(logn)
- Usuwanie – O(logn)
- Wyszukiwanie – O(nlogn)

Drzewo poszukiwań binarnych

BST składa się połączonych ze sobą węzłów, gdzie każdy węzeł posiada 4 pola – wskaźnik na rodzica, wskaźnik na lewego i prawego potomka oraz wartość. W BST każdy węzeł przechowuje wartość większą niż każdy węzeł lewego poddrzewa i większy niż każdy węzeł w prawym poddrzewie. Koszt wykonywania operacji na drzewie BST jest wprost proporcjonalny od wysokości drzewa ($O(h)$), gdzie wysokość w przybliżeniu równa jest $log_2 n$. Koszt operacji dodawania, usuwania i wyszukiwania w drzewie jest zależny od jego budowy, jeśli drzewo nie jest zbalansowane, złożoność może być równa nawet $O(n)$, jednak w przypadku zbalansowanego drzewa $\boldsymbol{O(log\, n)}$**.**

Drzewo czerwono-czarne

Drzewo czerwono-czarne jest typem samoorganizującego się drzewa poszukiwań binarnych. Każdy węzeł oprócz pól, które występują w węzłach drzewa BST dodatkowo posiada pole oznaczające jego kolor. W każdym drzewie binarnym dowolna ścieżka od węzła do dowolnego węzła NIL przechodzi przez taką samą liczbę czarnych węzłów. Złożoność obliczeniowa operacji: **dodawania, usuwania** oraz **wyszukiwania** na takim drzewie równa jest $\boldsymbol{O(log\, n)}$**.**

Przeprowadzanie testów

Cel eksperymentu

Celem eksperymentu było zmierzenie czasu wykonania poszczególnych operacji struktur danych w zależności od wielkości oraz porównanie wyników z wartościami odniesienia.

Wstęp

Testy zostały przeprowadzane dla kilku wielkości (zależnie od struktury), aby móc określić typ zależności. W celu zminimalizowania wpływu wartości elementów na wyniki, testy dla pojedynczej wielkości były **powtarzane po 15 razy** na losowych wartościach i uśredniane.
W celu łatwego wykonania testów i uzyskania wyników napisałem specjalną klasę *Benchmark*, która na podstawie danych wejściowych przeprowadza serię testów (dla każdej operacji w danej strukturze) i eksportuje ich wyniki do plików *.csv.* Dzięki temu możliwa była szybka manipulacja parametrami testów i wgląd w wyniki. Testy można uruchomić z poziomu *menu* głównego programu dostarczonego razem ze sprawozdaniem. Poniżej znajduje się lista parametrów startowych.

Parametry startowe klasy *Benchmark*:

- Rodzaj struktury
- Wielkość początkowa
- Krok – wielkość, o którą zwiększany jest rozmiar struktury przy każdej iteracji
- Mnożnik – wielkość, o którą mnożony jest rozmiar przy każdej iteracji (*najlepiej = 1*)
- Ilość powtórzeń
- Wielkość maksymalna

Przebieg eksperymentu

Dla każdego testu generowana jest populacja odpowiedniej długości składająca się z losowych liczb całkowitych z przedziału $(-10^7, 10^7)$. Następnie do testowanej struktury danych sekwencyjnie ładowane są dane z tej populacji. Gdy już wszystkie dane będą załadowane wykonywana jest operacja i mierzony jest jej czas wykonania. Wszystkie pomiary czasu wykonywane są przy użyciu funkcji *QueryPerformanceCounter,* a wyniki pomiarów podawane w μs.

Dla każdej struktury danych parametry testów zostały dobrane w taki sposób, aby czas wykonania samego testu nie był za długi.

Drzewo poszukiwań binarnych

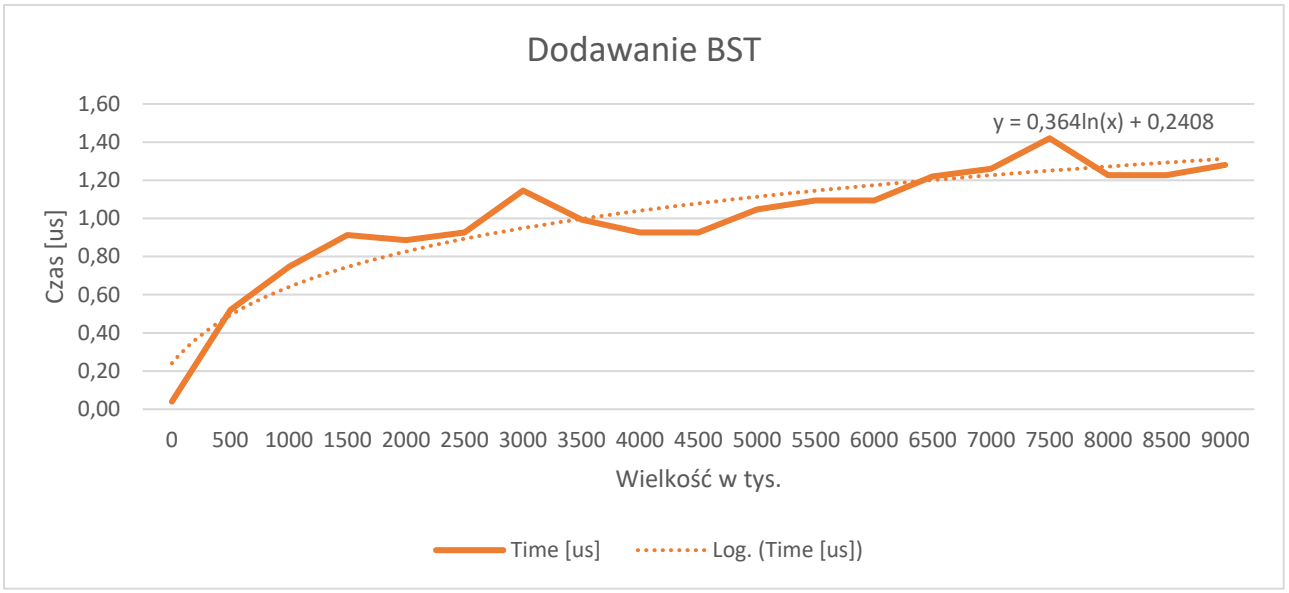
BST akceptuje 3 operacje: dodawanie, usuwanie oraz wyszukiwanie.
Parametry testów:

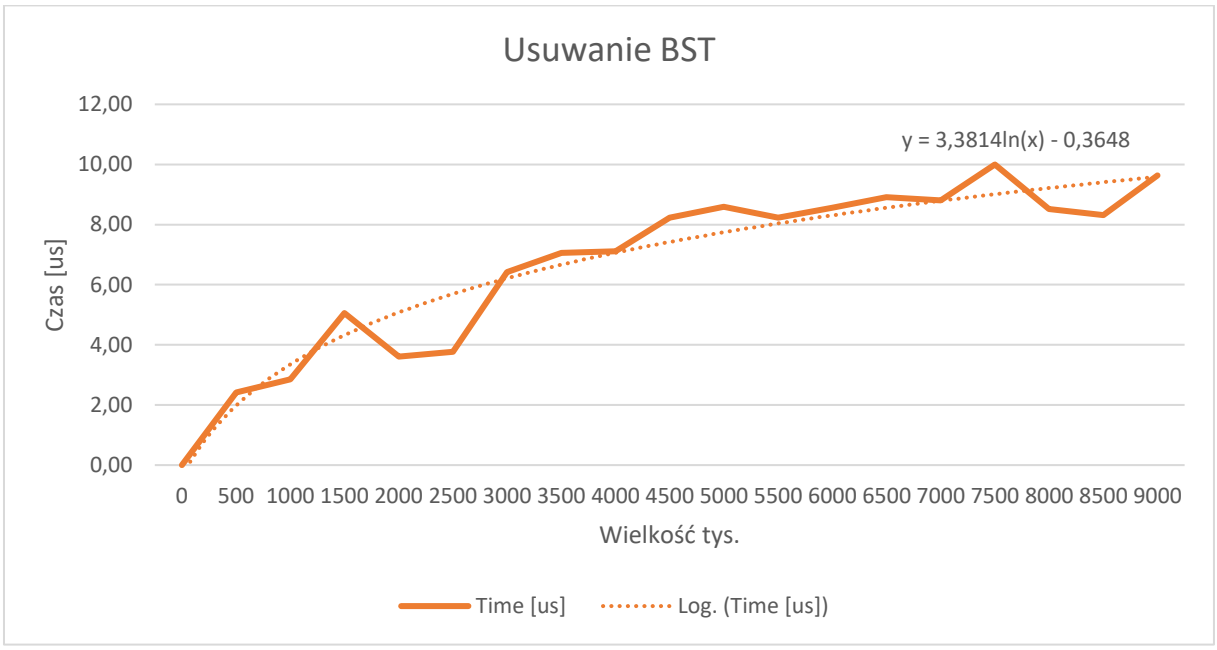
- Maksymalna wielkość: 9 milionów
- Krok: 0,5 miliona
- Ilość testów: 19

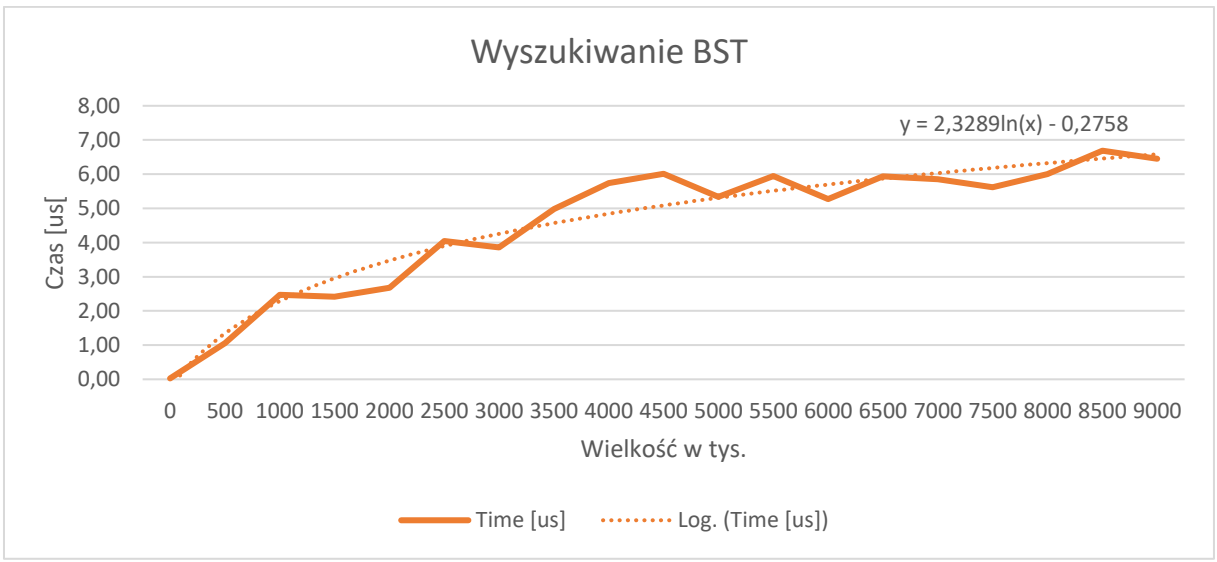
Wyniki testu dodawania do BST (czas wykonania operacji od wielkości struktury):

Structure Size $[10^3]$	0	500	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500	6000	6500	7000	7500	8000	8500	9000
Add $[\mu s]$	0	0,52	0,75	0,91	0,89	0,93	1,15	0,99	0,93	0,93	1,05	1,09	1,09	1,22	1,26	1,42	1,23	1,23	1,28
Delete $[\mu s]$	0	2,41	2,85	5,06	3,61	3,77	6,42	7,06	7,11	8,23	8,59	8,23	8,56	8,91	8,81	10,00	8,51	8,31	9,63
Search [us]	0	1,05	2,47	2,42	2,67	4,04	3,85	4,98	5,74	6,01	5,34	5,95	5,27	5,94	5,85	5,62	6,00	6,69	6,45

Wykresy z nałożoną logarytmiczną linią trendu:


Dodawanie BST
y = 0,364ln(x) + 0,2408
Czas [us]
1,60
1,40
1,20
1,00
0,80
0,60
0,40
0,20
0,00
0 500 1000 1500 2000 2500 3000 3500 4000 4500 5000 5500 6000 6500 7000 7500 8000 8500 9000
Wielkość w tys.
Time [us]
Log. (Time [us])


Usuwanie BST
y = 3,3814ln(x) - 0,3648
Czas [us]
12,00
10,00
8,00
6,00
4,00
2,00
0,00
0 500 1000 1500 2000 2500 3000 3500 4000 4500 5000 5500 6000 6500 7000 7500 8000 8500 9000
Wielkość tys.
Time [us]
Log. (Time [us])


Wyszukiwanie BST
y = 2,3289ln(x) - 0,2758
Czas [us[
8,00
7,00
6,00
5,00
4,00
3,00
2,00
1,00
0,00
0 500 1000 1500 2000 2500 3000 3500 4000 4500 5000 5500 6000 6500 7000 7500 8000 8500 9000
Wielkość w tys.
Time [us]
Log. (Time [us])

Po wyrysowaniu wykresów zależności operacji na drzewie poszukiwań binarnych widać wyraźną zależność logarytmiczną. Na wszystkie wykresy dodatkowo nałożyłem logarytmiczną linię trendu, aby łatwiej można było dostrzec podobieństwo. Na podstawie tych danych można więc stwierdzić, że rzeczywiście operacje na BST posiadają zależność czasową $O(\log n)$.

Lista dwukierunkowa

Jak już wcześniej wspomniałem, moja implementacja listy dwukierunkowej nie posiada wskaźnika na ogon. Oznacza to, że operacje dodawania na koniec będą miały bardzo czasochłonne i będą miały zależność liniową. W ramach testu zdecydowałem się na znacznie mniejszą wielkość niż w przypadku BST, gdyż w przeciwnym wypadku testy po prostu trwałyby zbyt długo.

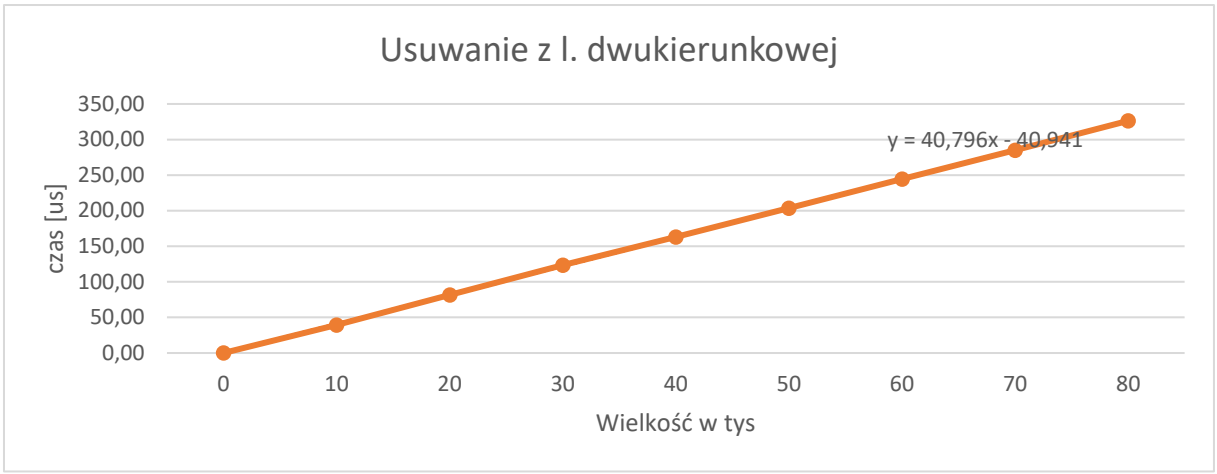
- Maksymalna wielkość: 80 tysięcy
- Krok: 10 tysięcy
- Ilość testów: 9

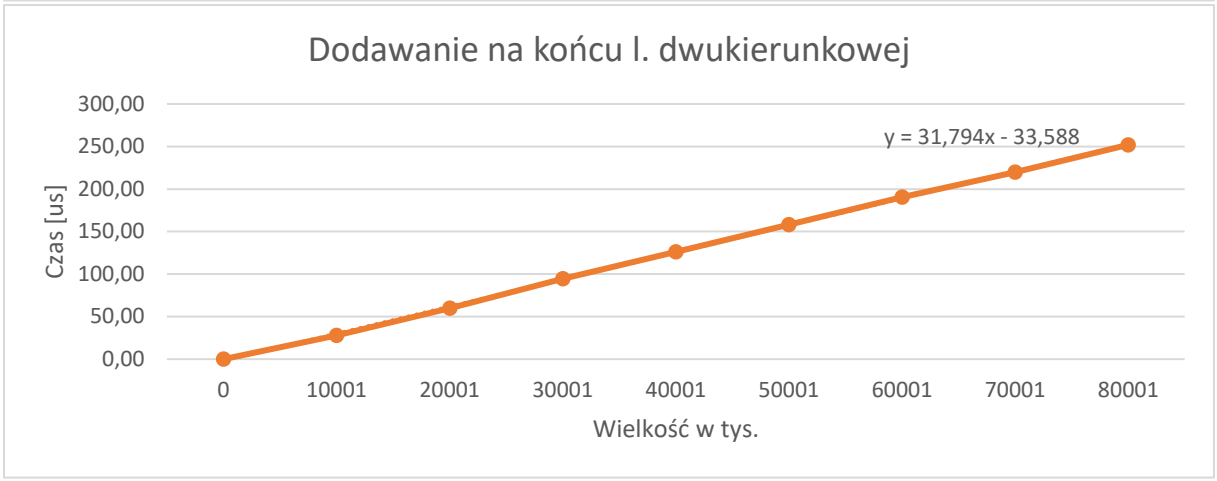
Tutaj będziemy mieli aż 4 operacje do testowania:

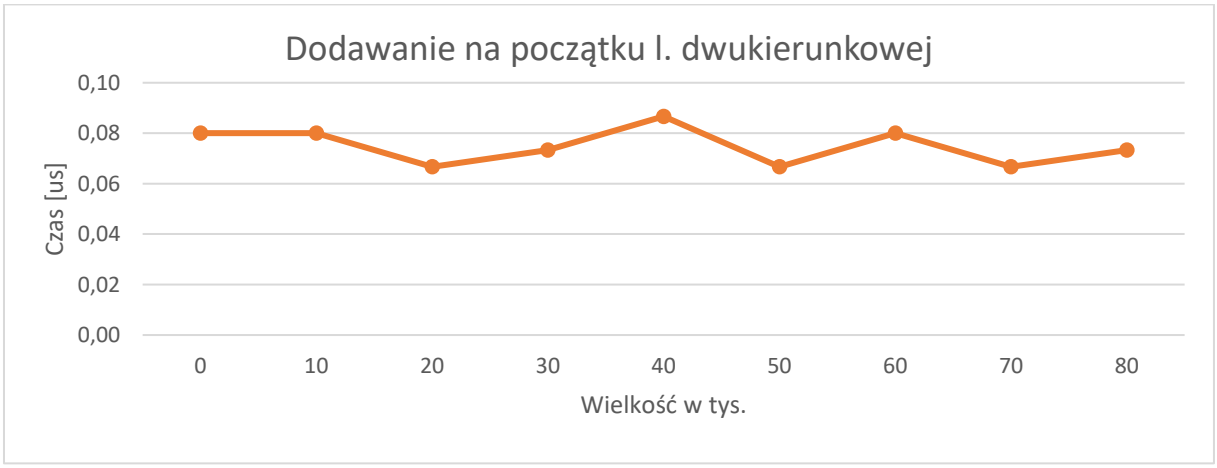
- Dodawanie na końcu
- Dodawanie na początku
- Wyszukiwanie
- Usuwanie

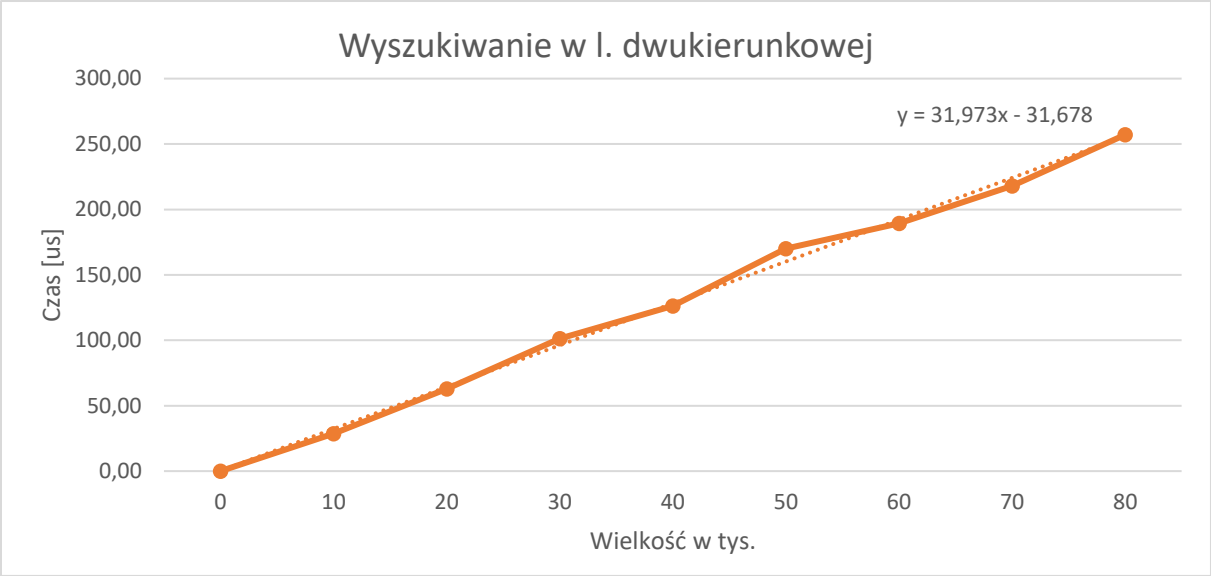
Wyniki testów listy dwukierunkowej:

Structure size [10^3]	0	10	20	30	40	50	60	70	80
Insert Back [μs]	0	27,92	59,94	94,45	125,96	158,107	190,47	219,68	251,91
Insert Front [μs]	0	0,08	0,07	0,07	0,09	0,07	0,08	0,067	0,07
Delete [μs]	0	39,5	81,56	123,53	163,17	203,59	244,63	285,11	326,23
Search [μs]	0,	28,65	62,9	101,28	126,17	170,1	189,48	218,03	257,06


Usuwanie z l. dwukierunkowej
350,00
300,00
250,00
200,00
150,00
100,00
50,00
0,00
czas [us]
y = 40,796x - 40,941
0
10
20
30
40
50
60
70
80
Wielkość w tys


Dodawanie na końcu l. dwukierunkowej
300,00
250,00
200,00
150,00
100,00
50,00
0,00
Czas [us]
y = 31,794x - 33,588
0
10001
20001
30001
40001
50001
60001
70001
80001
Wielkość w tys.


Dodawanie na początku l. dwukierunkowej
0,10
0,08
0,06
0,04
0,02
0,00
Czas [us]
0
10
20
30
40
50
60
70
80
Wielkość w tys.



Wykresy zależności operacji dodawania na końcu listy, usuwania oraz wyszukiwania posiadają zależność liniową. Operacja dodawania na początku listy posiada wykres, ze stałym trendem.

Tablica dynamiczna

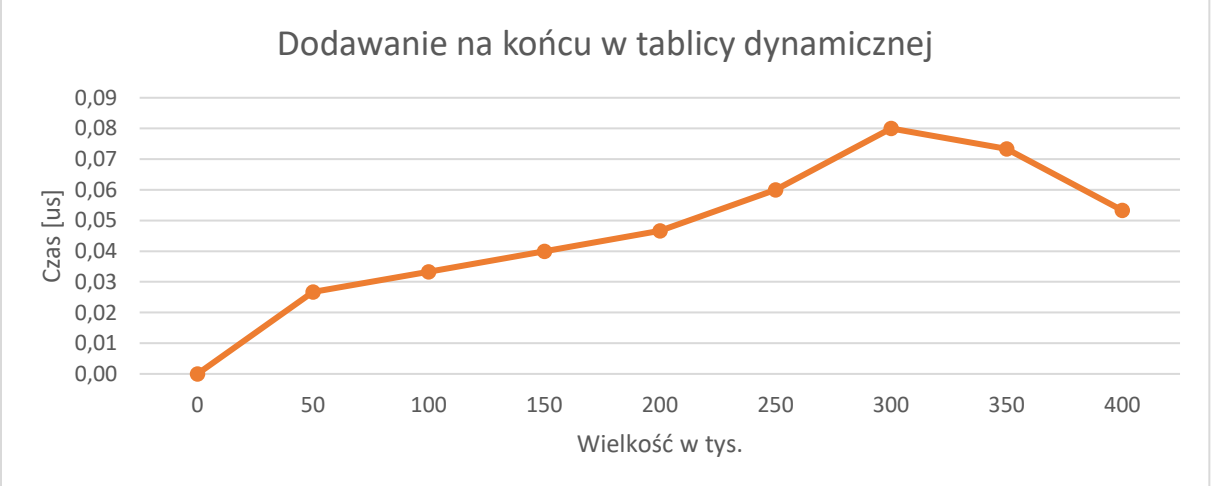
Tutaj także parametry startowe musiały zostać trochę zmniejszone:

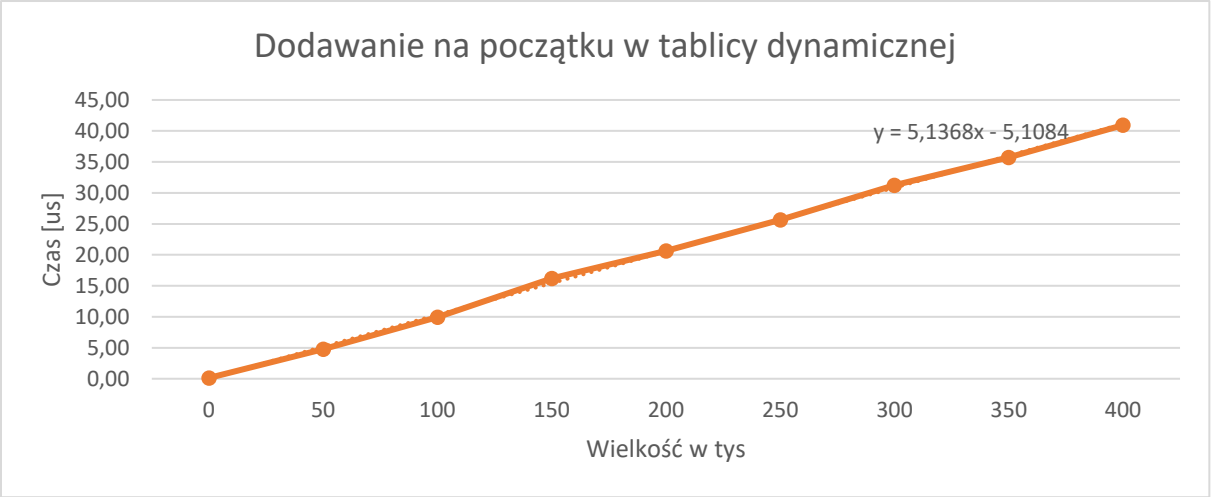
- Maksymalna wielkość: 400 tysięcy
- Krok: 50 tysięcy
- Ilość testów: 9

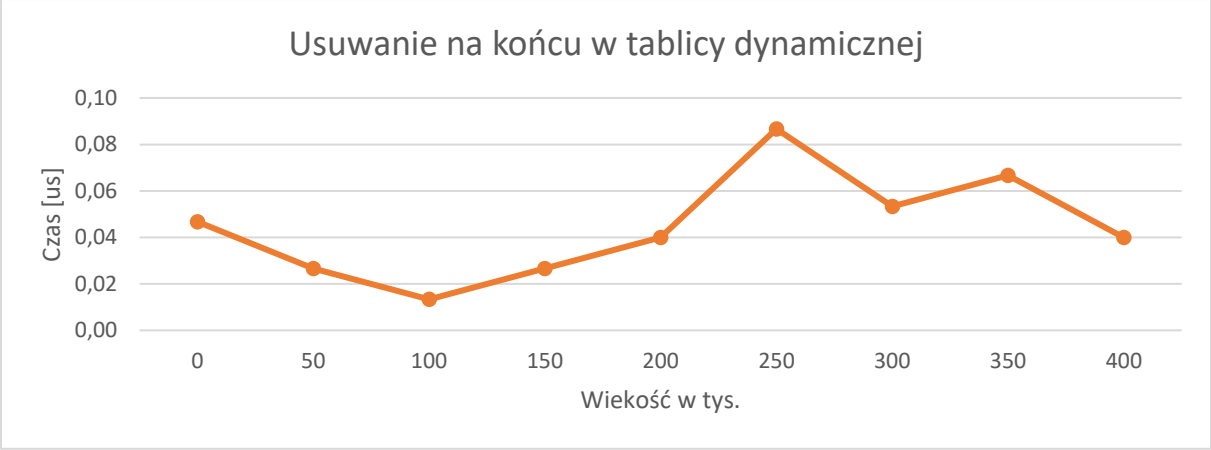
Testowane operacje:

- Dodawanie na początku
- Dodawanie na końcu
- Usuwanie

Structure size [10^3]	0	50	100	150	200	250	300	350	400
Insert Back [μs]	0	0,03	0,03	0,04	0,05	0,06	0,08	0,07	0,05
Insert Front [μs]	0	4,75	9,92	16,21	20,61	25,64	31,23	35,75	40,93
Delete [μs]	0	0,03	0,01	0,03	0,04	0,09	0,05	0,07	0,04







Wg. wcześniej podanych informacji dodawanie na końcu w tablicy dynamicznej powinno mieć zależność $O(1)$ (wykres powinien być stały). Niestety mój wykres nie przedstawia tej zależności poprawnie, możliwe że tego przyczyną jest realokacja pamięci w momencie dodania nowego elementu. Na końcu wykresu widoczny jest spadek czasu potrzebnego na wykonanie operacji, więc możliwe jest, że rzeczywiście przy operacjach na większych danych wykres by się ustabilizował, niestety nie byłem w stanie zwiększyć rozmiaru, gdyż już w tym momencie testy dość długo trwały.

Jeśli chodzi o dodawanie na początku, wykres bardzo ładnie przedstawia zależność liniową. Aby łatwiej było to dostrzec na wykres dodatkowo nałożyłem linię trendu i jej równanie.

Wyniki testów operacji usuwania tak jak w przypadku dodawania na końcu, nie są jednoznaczne. Na początku widać wyraźny spadek czasu wykonania operacji, potem znów wzrost i stabilizacja. Linia trendu tego wykresu jest w przybliżeniu stała, nie widać wyraźnego trendu wzrostowego ani spadkowego w tym wykresie, dlatego można uznać, że wyniki testów są zgodne z informacjami podanymi w literaturze.

Kopiec binarny

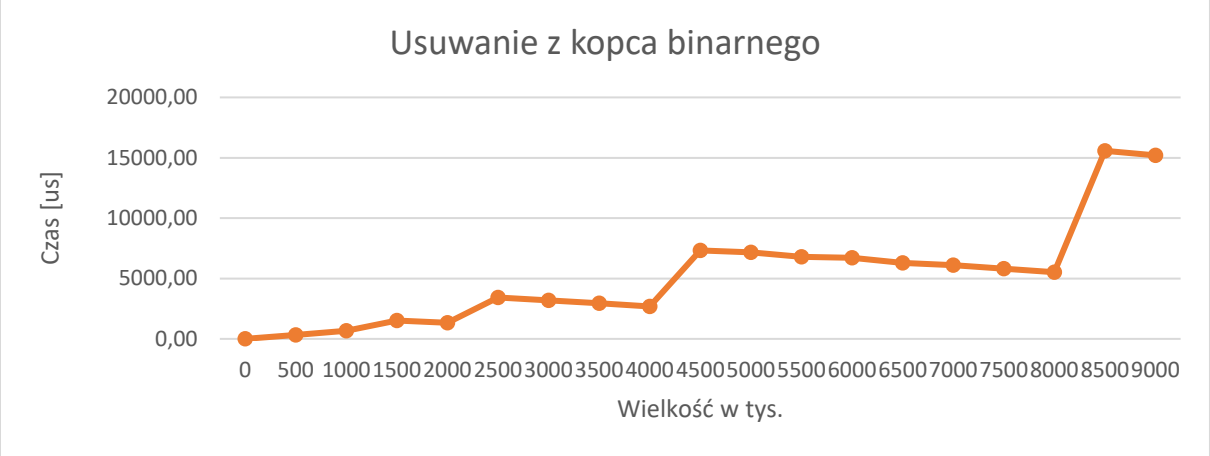
Kopiec binarny pomimo tego, że bazuje na tablicy dynamicznej, okazał się bardzo wydajny, dlatego w ramach testów próg wielkości ustawiłem tak jak w BST, czyli 9 milionów.

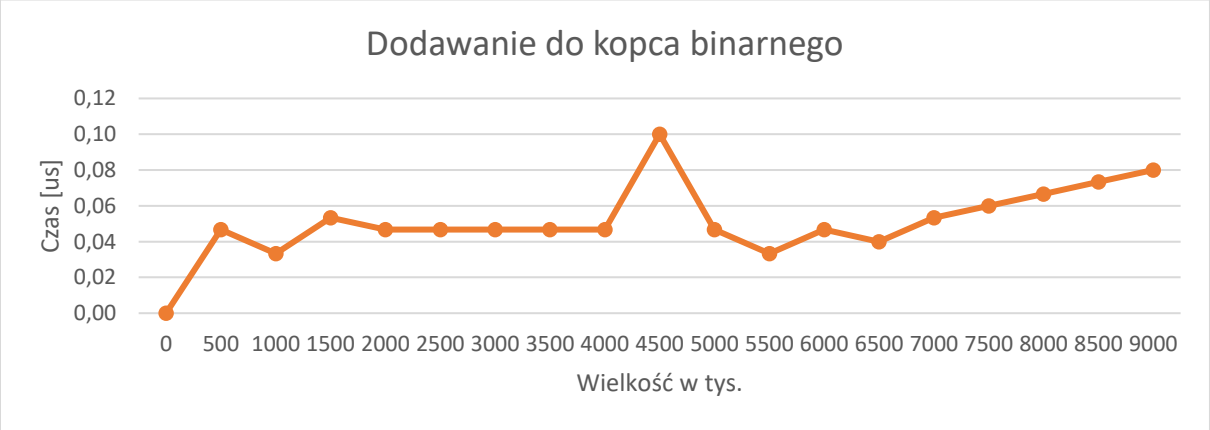
- Maksymalna wielkość: 9 milionów
- Krok: 500 tysięcy
- Ilość testów: 19

Testowane operacje:

- Dodawanie
- Usuwanie

Structure size [10^3]	0	500	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500	6000	6500	7000	7500	8000	8500	9000
Add [µs]	0	0,05	0,03	0,05	0,04	0,05	0,05	0,05	0,05	0,1	0,05	0,03	0,05	0,04	0,05	0,06	0,07	0,07	0,08
Delete [µs]	0	327,79	657,92	1529,64	1320,67	3434,31	3190,17	2957,13	2680,69	7335,82	7170,19	6798,73	6726,66	6286,79	6115,84	5818,05	5534,39	15576,00	15206,80





Według informacji zewnętrznych operacje dodawania oraz usuwania z kopca mają złożoność $O(\log n)$. Wyniki testów także wskazują na taką zależność.

Drzewo czerwono-czarne

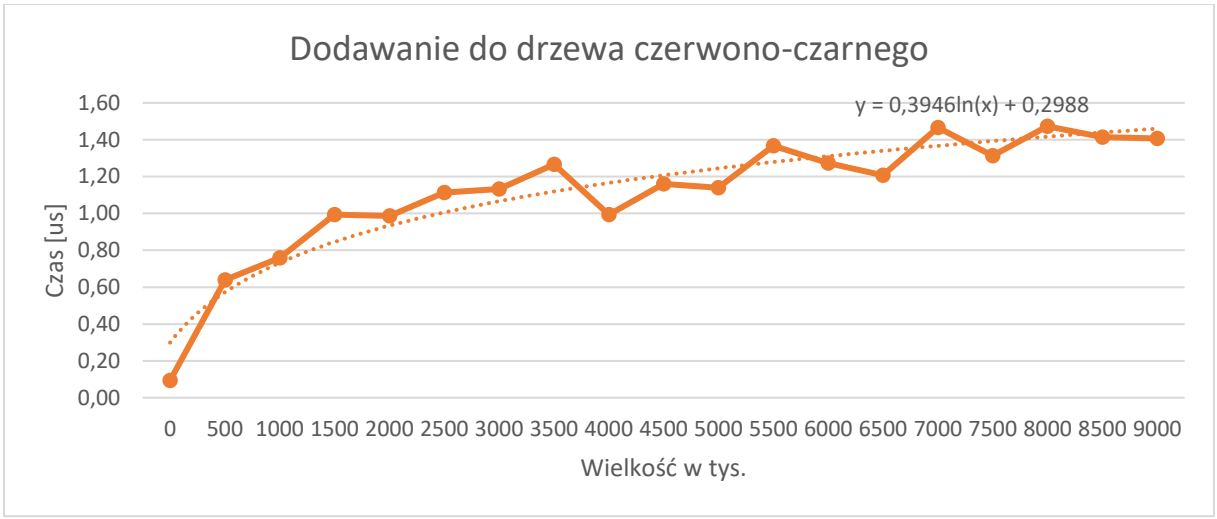
- Maksymalna wielkość: 9 milionów
- Krok: 500 tysięcy
- Ilość testów: 19

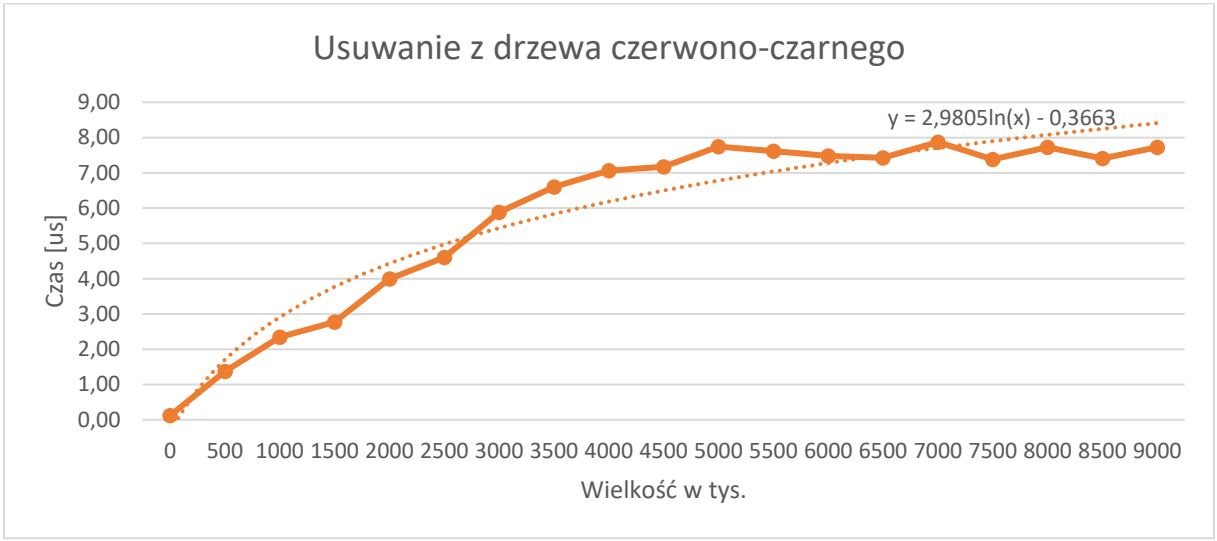
Testowane operacje:

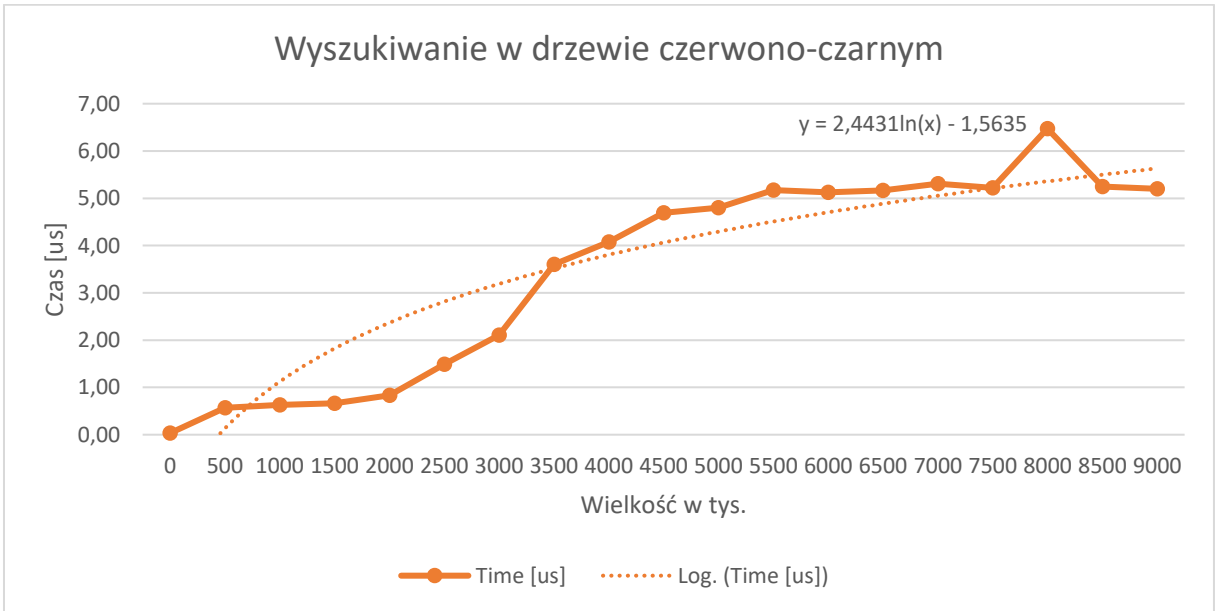
- Dodawanie
- Usuwanie
- Wyszukiwanie

Structure size [10^3]	0	500	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500	6000	6500	7000	7500	8000	8500	9000
Add [µs]	0,09	0,64	0,76	0,99	0,99	1,11	1,13	1,27	0,99	1,16	1,14	1,37	1,27	1,21	1,47	1,31	1,47	1,41	1,41

Delete [µs]	0,12	1,37	2,34	2,77	3,99	4,60	5,88	6,60	7,06	7,17	7,75	7,61	7,48	7,43	7,87	7,38	7,73	7,41	7,73
Search [µs]	0,03	0,57	0,63	0,67	0,83	1,49	2,11	3,60	4,07	4,69	4,80	5,17	5,13	5,17	5,31	5,22	6,47	5,25	5,20







W tym przypadku sytuacja jest bardzo podobna jak w BST, widoczna jest wyraźna zależność logarytmiczna operacji dodawania, usuwania oraz wyszukiwania.

Wnioski

Wykonane przeze mnie pomiary nie odbiegają od normy i są zgodne.
Pomiary dla tablicy dynamicznej jednak nie są jednoznaczne, znaczny wpływ na wyniki tych pomiarów mogła mieć realokacja danych, która w moim programie została zaimplementowana w dość *nieregularny* sposób – realokacja następuje tylko wtedy, gdy wielkość tablicy osiągnie wartość równą potędze dwójki. Takie rozwiązanie jest bardzo dobre jeśli chodzi o ogólne działanie tablicy, jednak podczas badania czasu wykonania pojedynczej operacji to może wprowadzić nieregularności (dana operacja może realokować dane lub nie), które nie są zależnie od danych wejściowych, tylko od wielkości tablicy (nie da się przeprowadzić testów na różnych populacjach).
Podsumowując, za wyjątkiem tablicy dynamicznej, wykonane przeze mnie pomiary złożoności czasowej operacji na strukturach danych nie odbiegają od złożoności czasowych podanych w literaturze.

Spis treści

Bibliografia

Doubly linked list. (2021, 04 23). Pobrano z lokalizacji Wikipedia: https://en.wikipedia.org/wiki/Doubly_linked_list

Dynamic Array. (2021, 04 23). Pobrano z lokalizacji Brilliant: https://brilliant.org/wiki/dynamic-arrays/

Kopiec Binarny. (2021, 04 24). Pobrano z lokalizacji Wikipedia: https://pl.wikipedia.org/wiki/Kopiec_binarny

Red-black tree. (2021, 04 24). Pobrano z lokalizacji Wikipedia: https://en.wikipedia.org/wiki/Red%E2%80%93black_tree

Struktura danych. (2021, 04 23). Pobrano z lokalizacji Wikipedia: https://pl.wikipedia.org/wiki/Struktura_danych