Struktury Danych i Złożoność Obliczeniowa

BADANIE EFEKTYWNOŚCI OPERACJI NA DANYCH W PODSTAWOWYCH STRUKTURACH DANYCH

Autor:
Kamil Bauer 259102

Prowadzący: dr inż. Dariusz Banasiak
Kod zajęć: K02-10j
Wrocław, 22 kwietnia 2022

Politechnika Wrocławska,
Wydział Informatyki i Telekomunikacji

Spis treści

1 Założenia projektowe

Celem projektu była implementacja i pomiar czasu działania operacji takich jak dodawanie elementu, usunięcie elementu i wyszukanie elementu w następujących strukturach danych:

- tablica,
- lista dwukierunkowa,
- kopiec binarny (typu maksimum – element maksymalny w korzeniu),
- drzewo czerwono-czarne

Podstawowym elementem struktur była 4 bajtowa liczba całkowita ze znakiem. Wszystkie struktury danych powinny być alokowane dynamicznie (w przypadku tablic powinny zajmować możliwie jak najmniej miejsca tzn. powinny być relokowane przy dodawaniu/usuwaniu elementów). Pomiar czasu został wykonany wielokrotnie (100 razy), ponieważ pojedynczy pomiar może być obarczony znacznym błędem, jak również generując za każdym razem nowy zestaw danych - otrzymane wyniki mogą zależeć od rozkładu danych„ a wyniki uśredniono. Ilość elementów dla których testowano strukury to następująco: 1000, 2000, 5000, 10000 i 20000.

1.1 Wykorzystane narzędzia

Narzędzia wykorzystane podczas pracy:

- C++11
- std::chrono::high_resolution_clock do pomiaru czasu
- Visual Studio 2019

- Git

1.2 Link do repozytorium na githubie

Cały kod źródłowy znajduje się na zdalnym repozytorium:
`https://github.com/bauerkamil/SDiZO1`

2 Wstęp teoretyczny

2.1 Tabela dynamiczna

Tabela charakteryzuje się tym, że kolejne jej elementy znajdują się w pamięci obok siebie (jedno za drugim). Tabela dynamiczna natomiast zmienia swój rozmiar tak, aby można było do niej dodać dowolną liczbę elementów. Standardowo, mając na uwadze czas operacji, zwiększa się tablicę x2, gdy się wypełni, natomiast zmniejsza, gdy zajmuje 1/4 swojej wielkości. W tym projekcie zależy nam jednak na jak najmniejszym wykorzystaniu pamięci, więc tablica jest realokowana przy każdym dodaniu lub usunięciu elementu.

Zaimplementowane operacje:

- Dodanie elementu na początek
- Dodanie elementu na dowolne miejsce
- Dodanie elementu na koniec
- Usunięcie elementu z początku
- Usunięcie elementu z dowolnego miejsca
- Usunięcie elementu z końca
- Wyszukanie elementu o podanym kluczu
- Wyszukanie elementu o podanym indeksie

Wszystkie poza jedną przedstawione wyżej operacje mają złożoność obliczeniową równą **O(n)**, co jest związane z każdorazowym realokowaniem tablicy oraz kopiowaniem jej elementów (n razy). Wyszukanie po kluczu wymaga w najgorszym przypadku przejrzenia wszystkich elementów. Jedynie dostęp po indeksie ma złożoność **O(1)**, jako że tabela jest indeksowana.

2.2 Lista dwukierunkowa

Lista to rodzaj struktury, w którym każdy element poza swoim kluczem posiada wskaźnik na następny element. W liście dwukierunkowej natomiast elementy posiadają wskaźniki zarówno na ich następcę, jak i poprzednika. Lista przechowuje natomiast swój początek i koniec.

Zaimplementowane operacje:

- Dodanie elementu na początek
- Dodanie elementu na dowolne miejsce
- Dodanie elementu na koniec

- Usunięcie elementu z początku
- Usunięcie elementu z dowolnego miejsca
- Usunięcie elementu z końca
- Wyszukanie elementu o podanym kluczu
- Wyszukanie elementu o podanym indeksie

Operacje dodawania oraz usuwania elementu na pierwszą oraz ostatnią pozycję posiadają złożoność obliczeniową **O(1)**. Operacje związane z konkretnym indeksem na liście wymagają przejrzenia wszystkich elementów, więc ich złożoność to **O(n)**, jednak można zaimplementować metody, które będą to wykonywać w najgoszym wypadku w trakcie n/2+1 iteracji, wykorzystując rozmiar stuktury oraz początek oraz koniec listy. Wyszukiwanie elementu o kluczu wymaga O(n) czasu.

2.3 Kopiec binarny maksymalny

Kopiec jest pewnego rodzaju drzewem binarnym, które dąży do bycia pełnym. W kopcu rozróżniamy dzieci oraz rodziców, natomiast własnością kopca maksymalnego jest, iż każdy rodzic jest większy bądź równy swoim dzieciom co do przechowywanej wartości. Dzięki spełnieniu tej zależności, w korzeniu kopca (elemencie o indeksie 0) zawsze jest przechowywany największy element, przez co dostęp do niego ma złożoność **O(1)** Kopiec może zostać zaimplementowany na podstawie drzewa lub tablicy, w tym projekcie została wykorzystana tablica.

Zaimplementowane operacje:

- Dodanie elementu
- Usunięcie korzenia
- Wyszukanie elementu o podanym kluczu
- Wyszukanie elementu o podanym indeksie

Operacje dodawania elementu oraz usuwania korzenia posiadają teoretyczną złożoność obliczeniową **O(lg n)**, natomiast z powodu implementacji kopca na bazie tablicy, po wykonaniu każdej z tych operacji występuje konieczność zaalokowania na nowo pamięci i przepisania każdego elementu. Z tego powodu wynikowa złożoność wynosi **O(n)**. Wyszukiwania elementu o podanym kluczu wykonywane jest na tablicy, więc posiada złożoność **O(n)**. Jedynie dostęp za pomocą indeksu odbywa się w czasie **O(1)**.

2.4 Drzewo czerwono-czarne

Drzewo czaerwono-czarne jest specyficznym rodzajem drzewa poszukiwań (Binary Search Tree), które samo się balansuje. Obowiązuje więc własność drzewa BST - wartości mniejsze od danego elementu znajdują się w lewym poddrzewie, a większe - w prawym. Elementy drzewa czerwono-czarne poza standardowymi wartościami: wskaźników na ojca oraz prawego i lewegoo syna, posiada również pole kolor - czerwone lub czarne. Jest ono wykorzystywane przy balansowaniu drzewa tak, aby jego własności zostały zachowane:

1. Każdy węzeł jest czerwony albo czarny.

2. Korzeń jest czarny.
3. Każdy liść jest czarny (Można traktować nil jako liść).
4. Jeśli węzeł jest czerwony, to jego synowie muszą być czarni.
5. Każda ścieżka z ustalonego węzła do każdego z jego potomków będących liśćmi liczy tyle samo czarnych węzłów.

Zaimplementowane operacje:

- Dodanie elementu
- Usunięcie korzenia
- Wyszukanie elementu o podanym kluczu
- Wyszukanie elementu o podanym indeksie

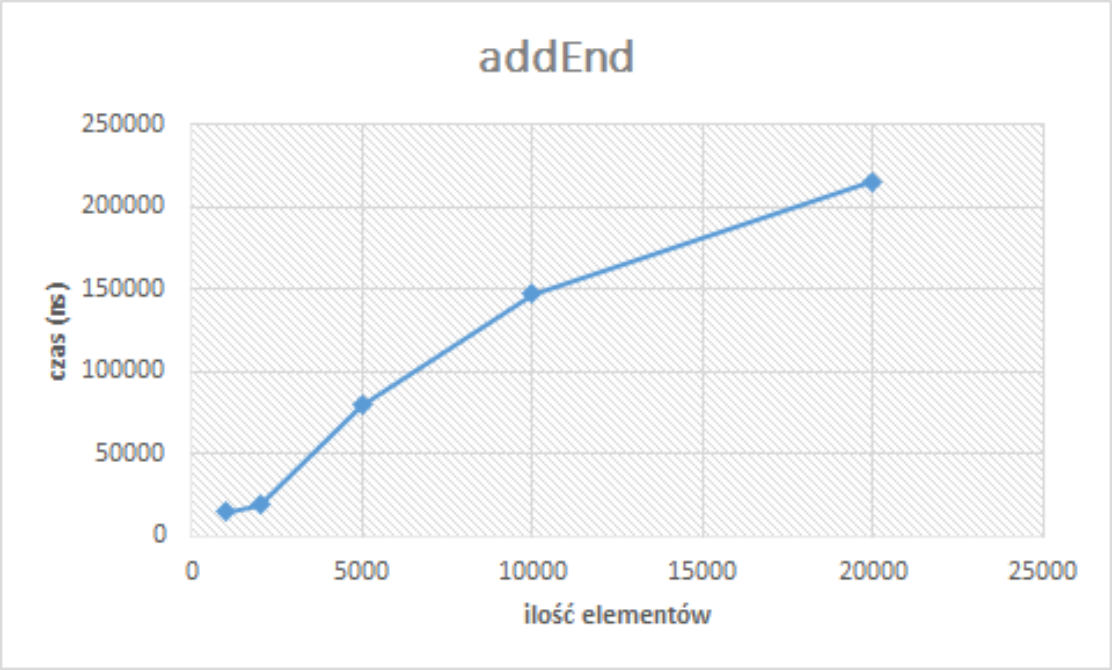
Wszystkie zaimplementowane operacje posiadają złożoność **O(lg n)**.

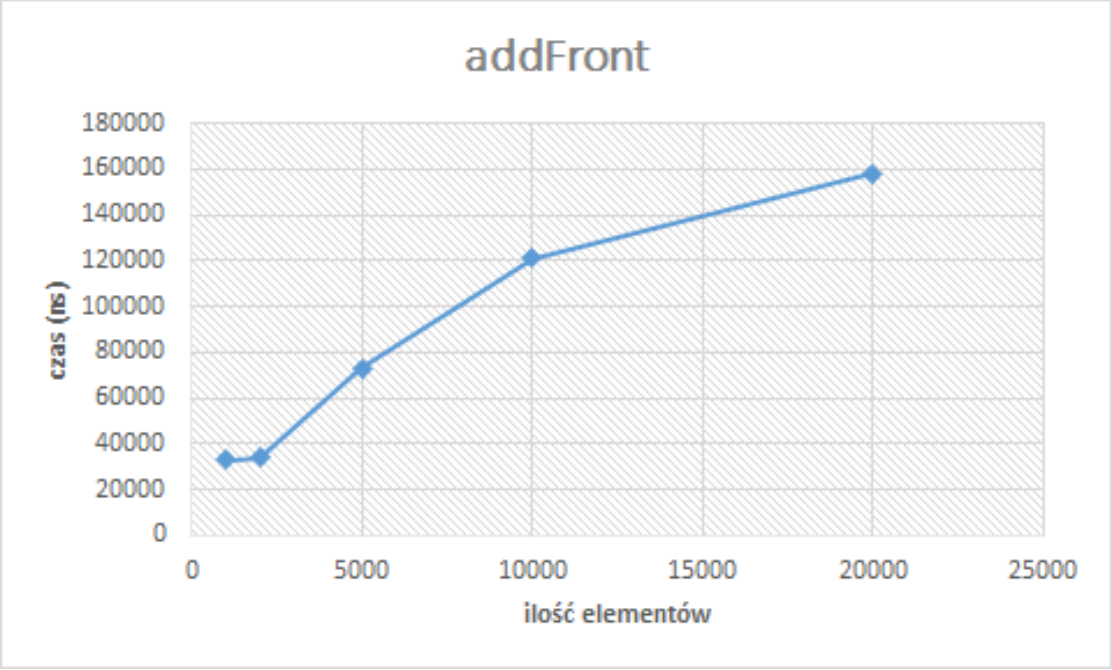
3 Uśrednione wyniki eksperymentów

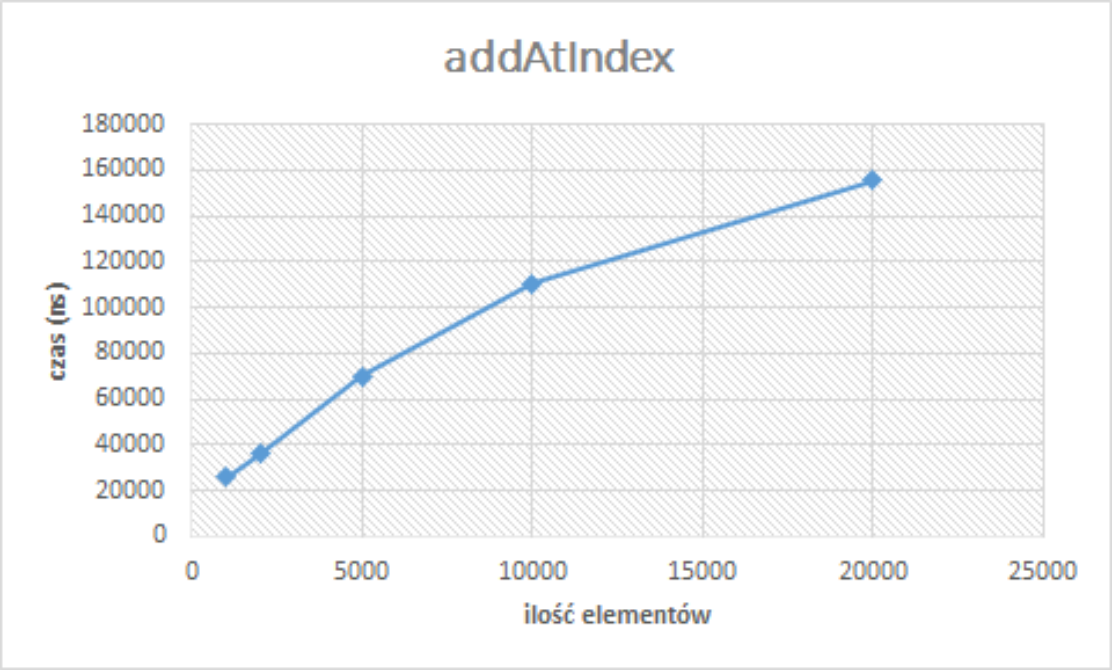
3.1 Tabela dynamiczna

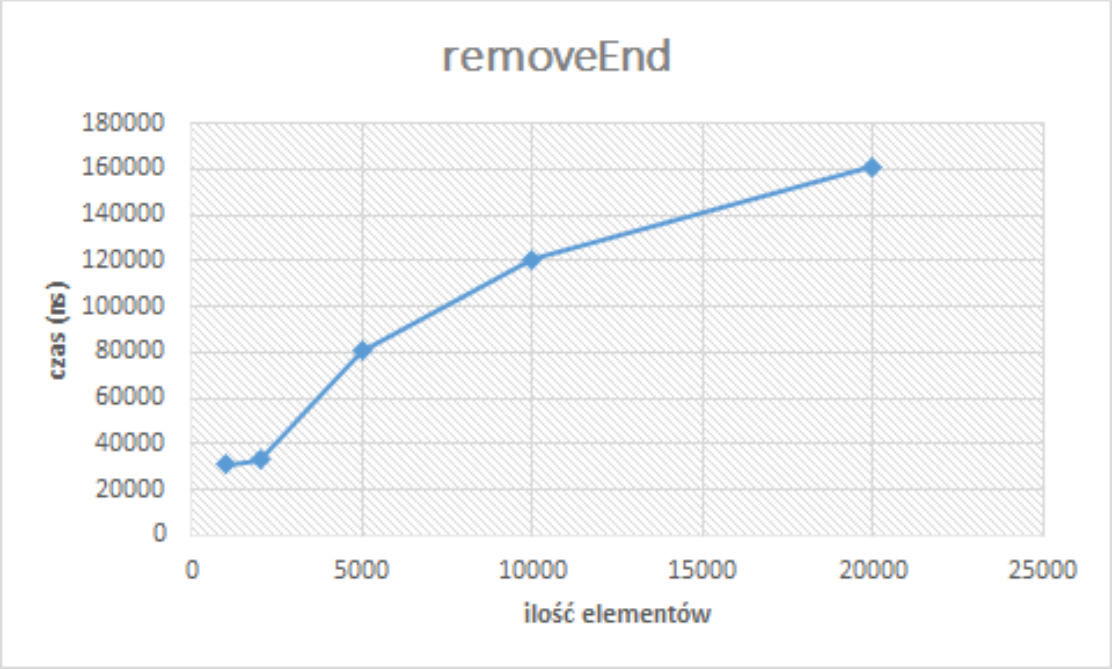
czas	addEnd	addFront	addAtIndex	removeEnd	removeFront	removeAtIndex	getByIndex	getIndex
1000	15337	33065	26074	31195	28209	23683	1148	2121350
2000	19083	34049	36382	33069	34963	29752	1234	1732726
5000	79668	73304	70195	80478	77943	73190	1227	3136905
10000	147338	120958	110539	120772	118921	132725	1546	2115416
20000	215530	158162	155848	161280	160520	150216	1468	1906176

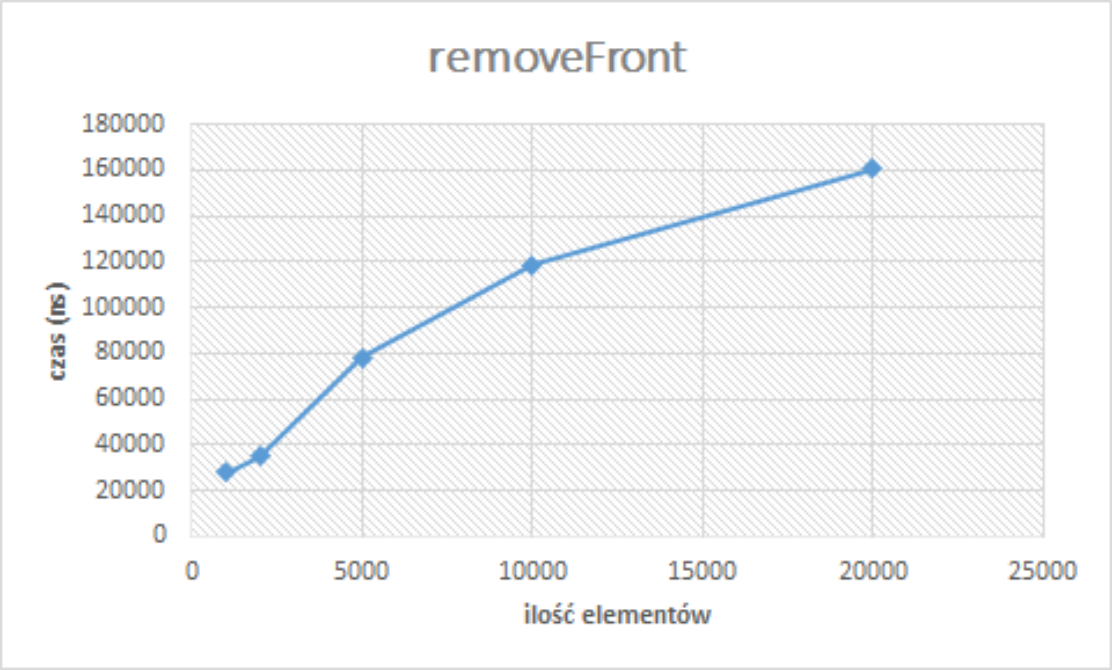
Tabela 1: Średnie wyniki w postaci tabeli na tabeli

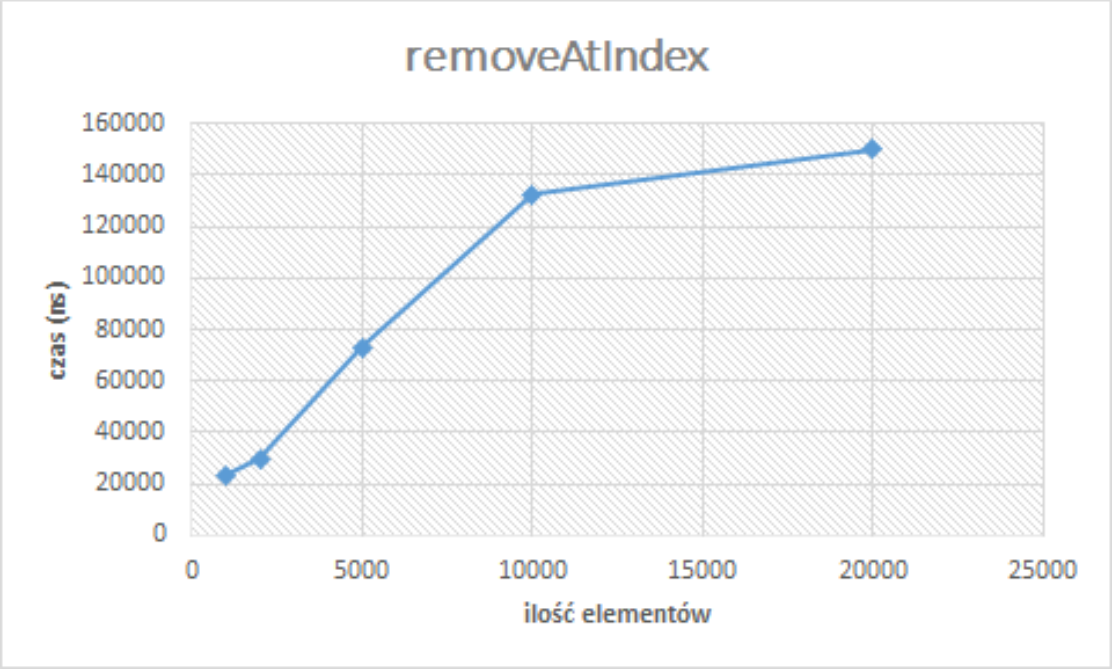

addEnd
250000
200000
150000
100000
50000
0
czas (ns)
0
5000
10000
15000
20000
25000
ilość elementów


addFront
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
czas (ns)
0
5000
10000
15000
20000
25000
ilość elementów

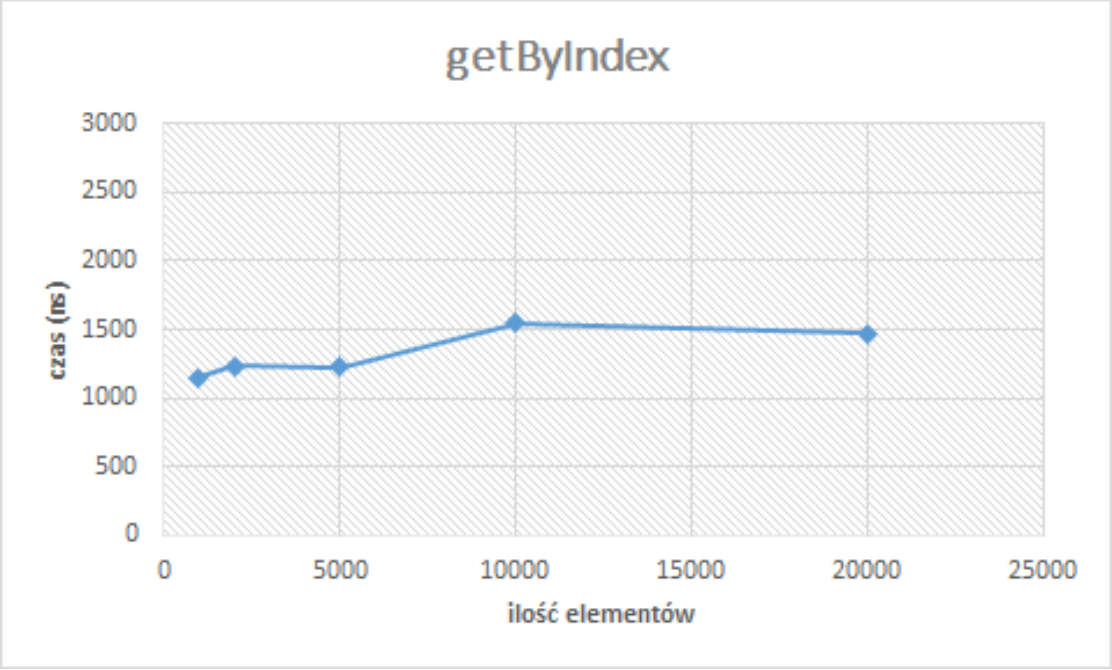

addAtIndex
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
czas (ns)
0
5000
10000
15000
20000
25000
ilość elementów


removeEnd
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
czas (ns)
0
5000
10000
15000
20000
25000
ilość elementów


removeFront
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
czas (ns)
0
5000
10000
15000
20000
25000
ilość elementów


removeAtIndex
160000
140000
120000
100000
80000
60000
40000
20000
0
czas (ns)
0
5000
10000
15000
20000
25000
ilość elementów

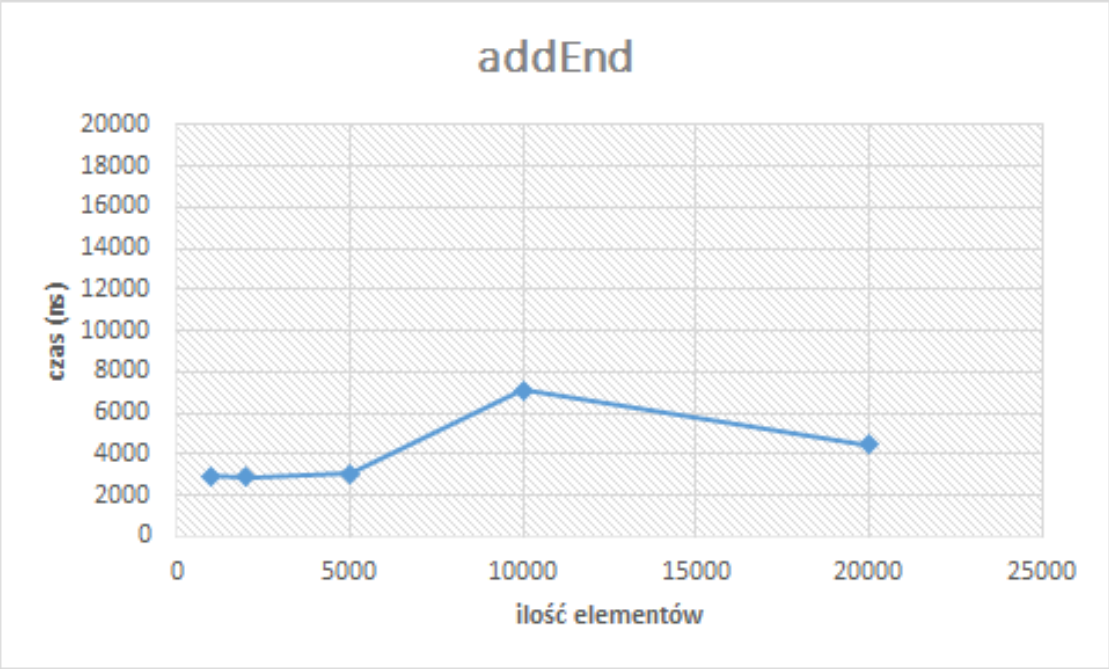

getIndex
3500000
3000000
2500000
2000000
1500000
1000000
500000
0
czas (ns)
0
5000
10000
15000
20000
25000
ilość elementów

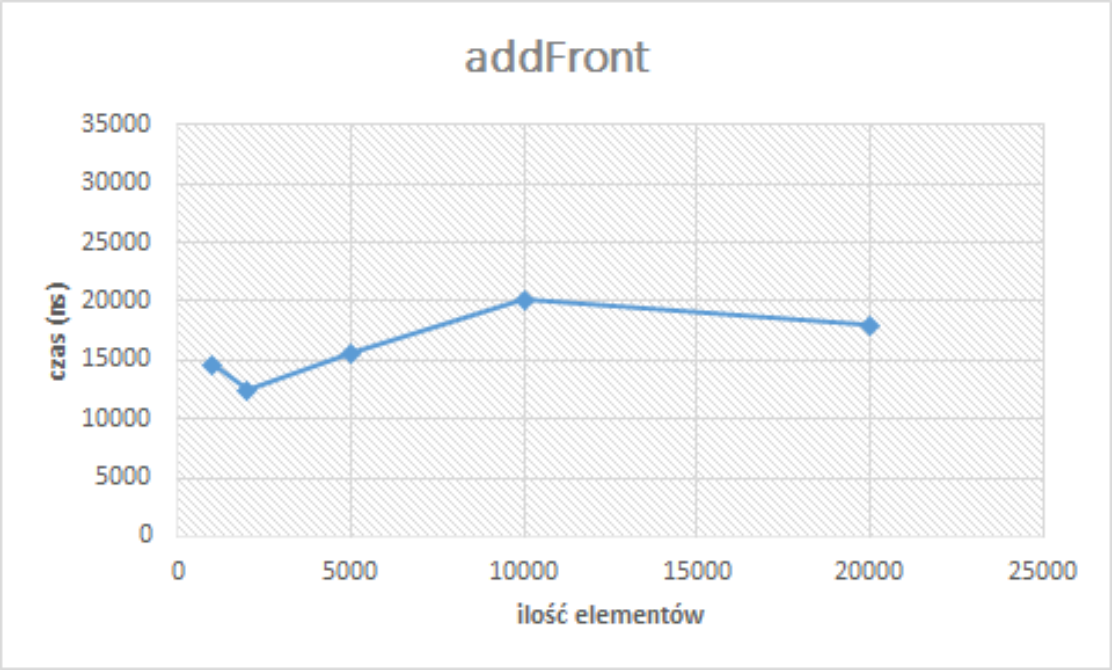

getByIndex
3000
2500
2000
1500
1000
500
0
czas (ns)
0
5000
10000
15000
20000
25000
ilość elementów

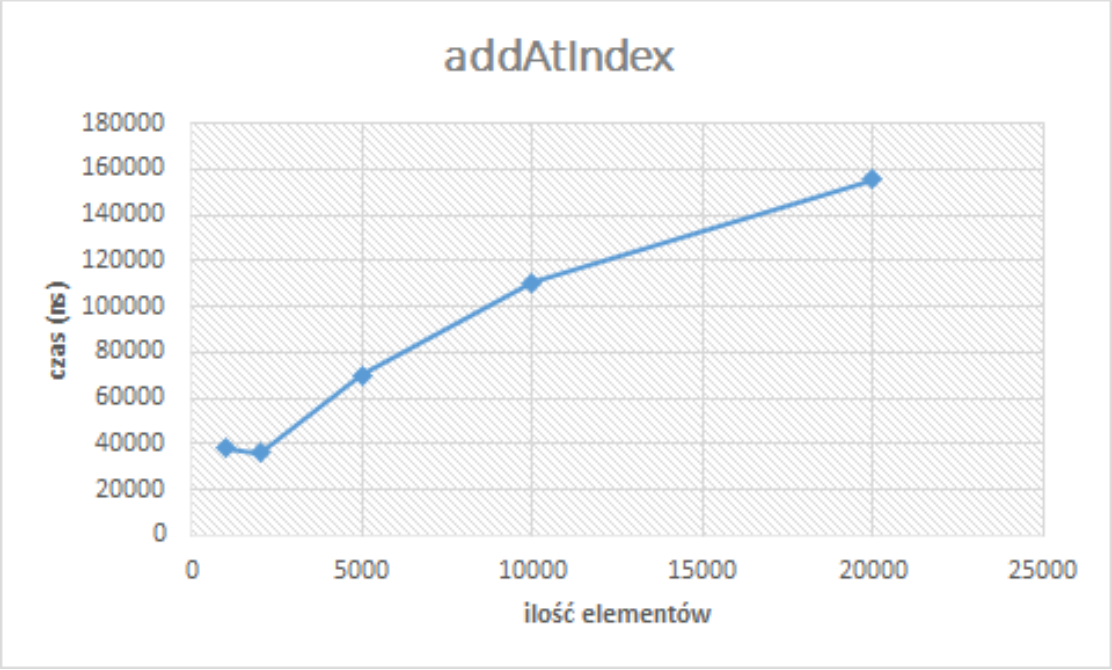
3.2 Lista dwukierunkowa

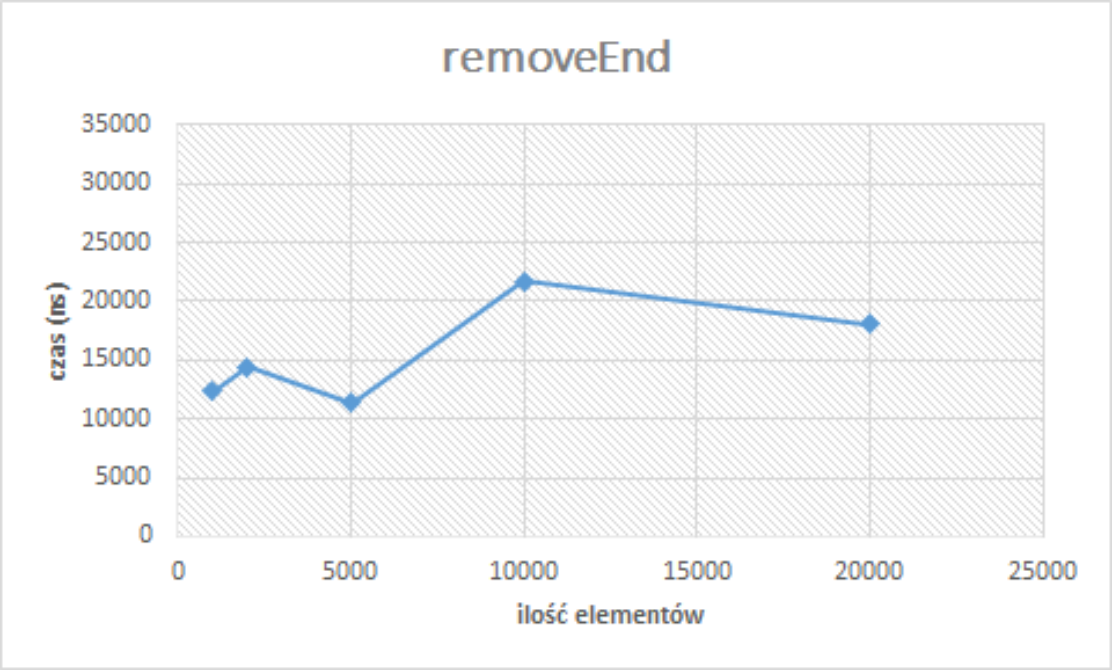
czas	addEnd	addFront	addAtIndex	removeEnd	removeFront	removeAtIndex	getByIndex	getIndex
1000	2924	14655	38246	12333	12303	44652	8971	1607451
2000	2910	12420	36382	14396	11085	29752	20946	1583794
5000	3070	15582	70195	11329	12501	73190	57604	1700853
10000	7095	20128	110539	21694	25644	132725	207225	2798863
20000	4482	17928	155848	18032	22079	150216	300953	2099516

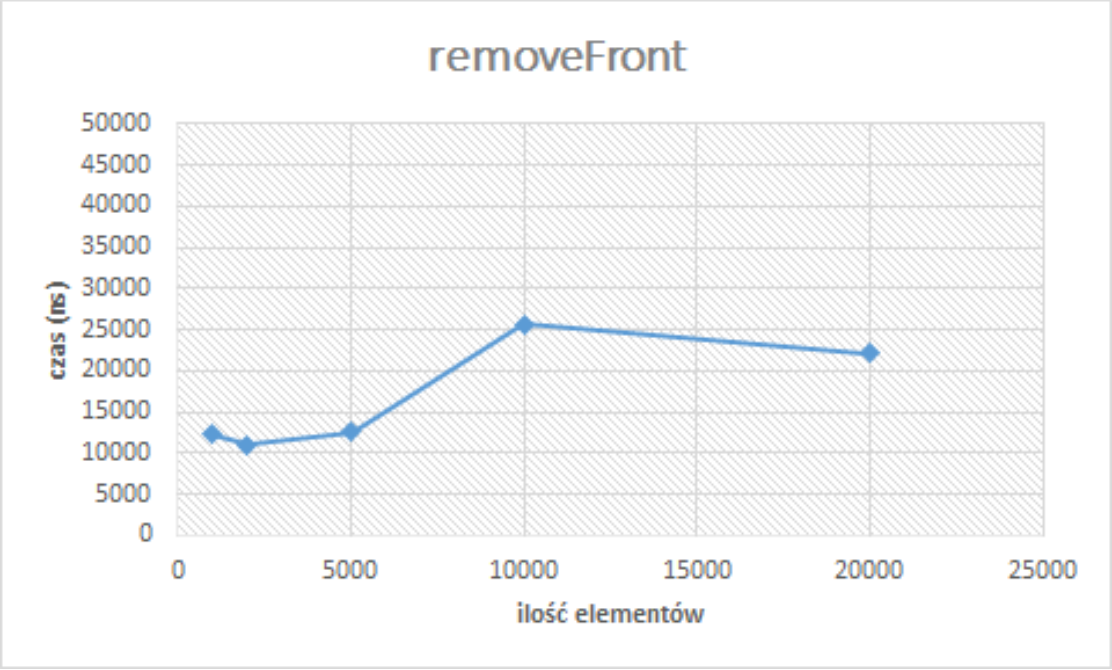
Tabela 2: Uśrednione wyniki eksperymentu na liście

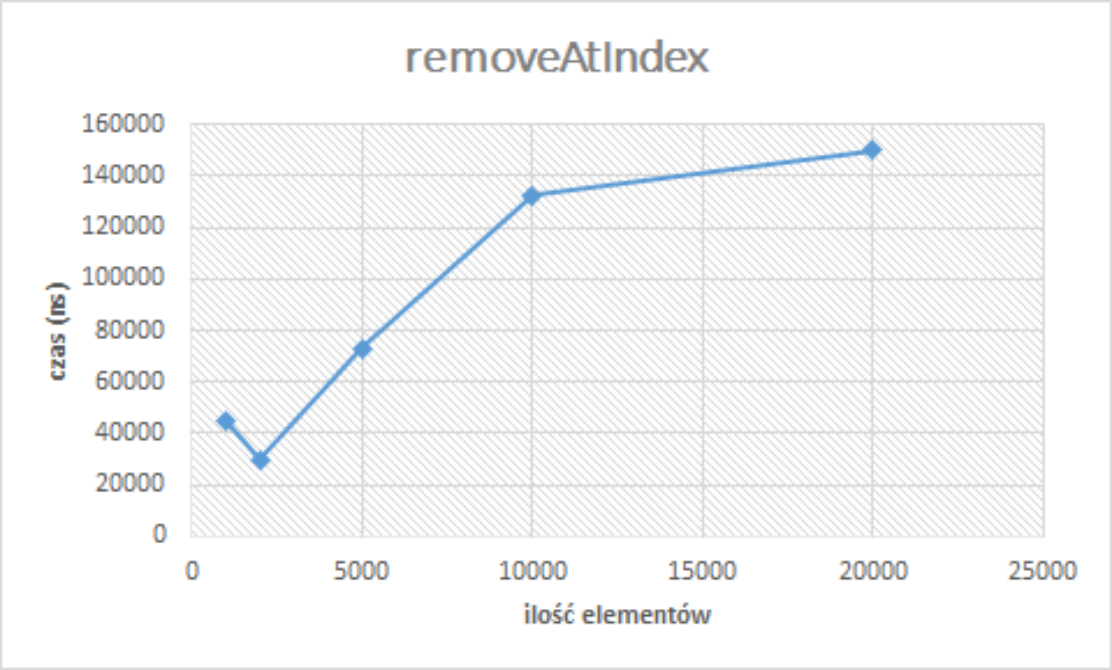


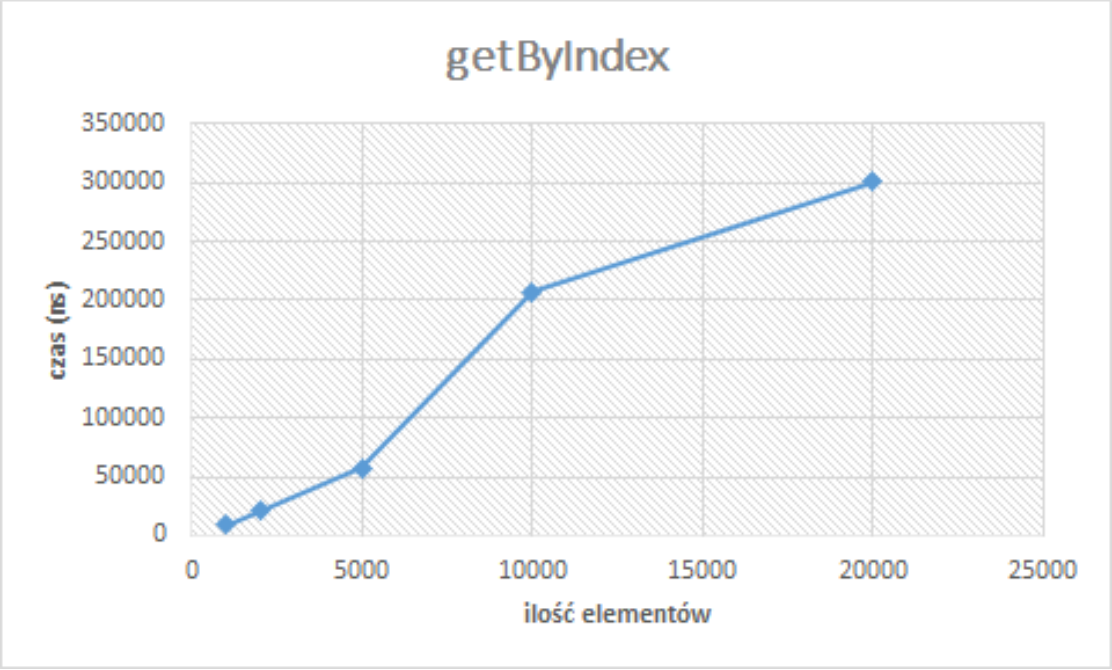

addFront
35000
30000
25000
20000
15000
10000
5000
0
czas (ns)
0
5000
10000
15000
20000
25000
ilość elementów

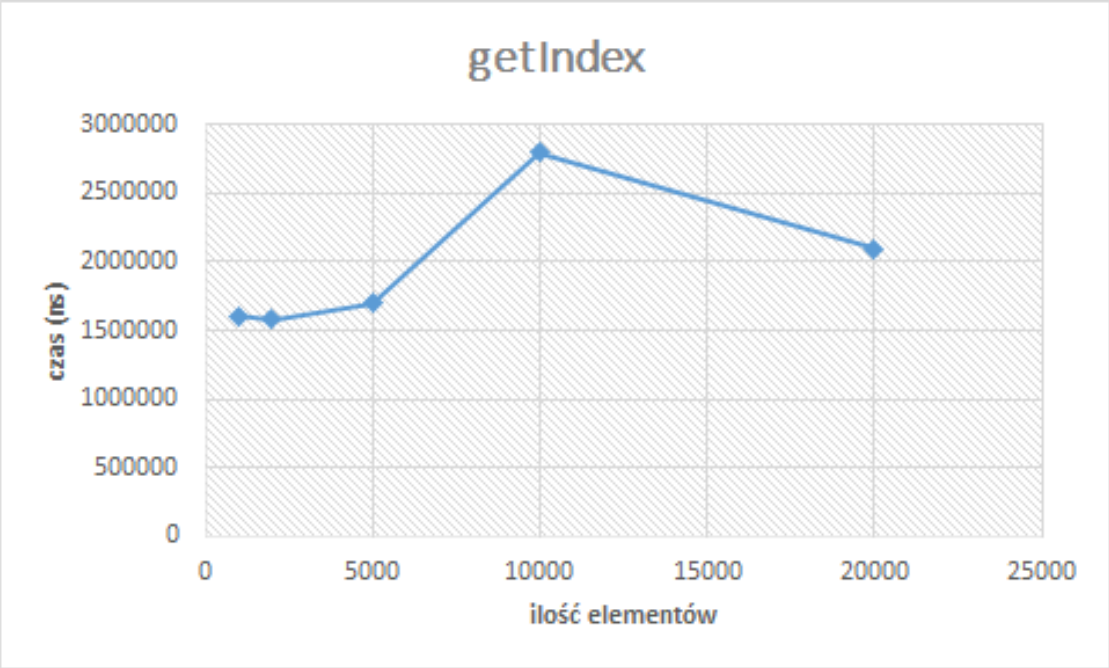

addAtIndex
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
czas (ns)
0
5000
10000
15000
20000
25000
ilość elementów


removeEnd
35000
30000
25000
20000
15000
10000
5000
0
czas (ns)
0
5000
10000
15000
20000
25000
ilość elementów


removeFront
50000
45000
40000
35000
30000
25000
20000
15000
10000
5000
0
czas (ns)
0
5000
10000
15000
20000
25000
ilość elementów


removeAtIndex
160000
140000
120000
100000
80000
60000
40000
20000
0
czas (ns)
0
5000
10000
15000
20000
25000
ilość elementów

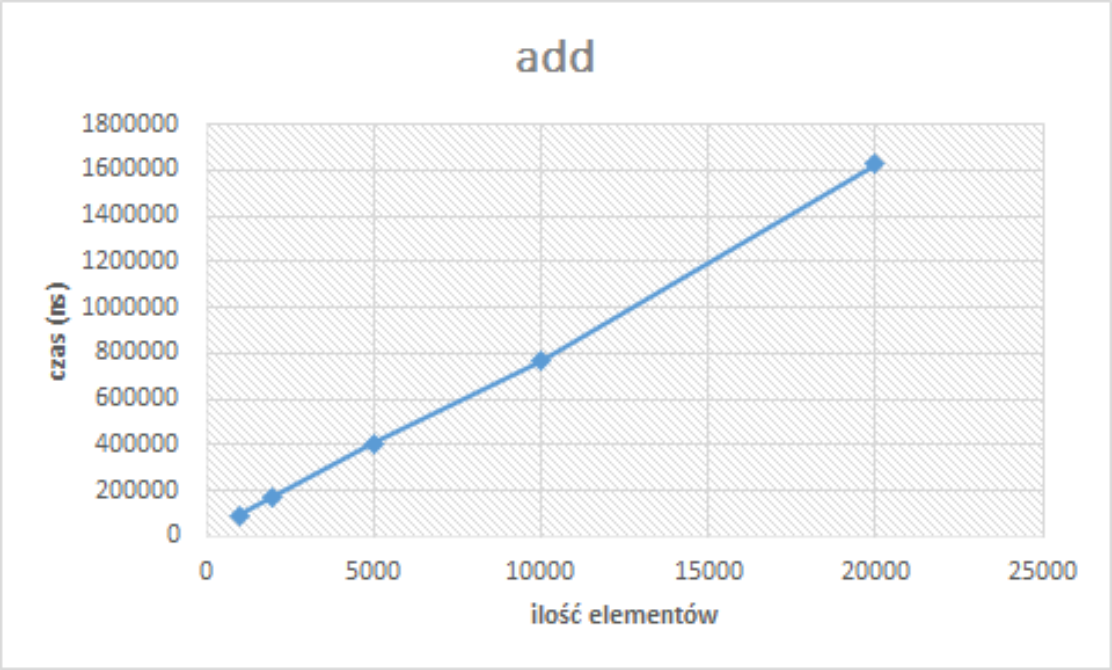

getByIndex
350000
300000
250000
200000
150000
100000
50000
0
czas (ns)
0
5000
10000
15000
20000
25000
ilość elementów

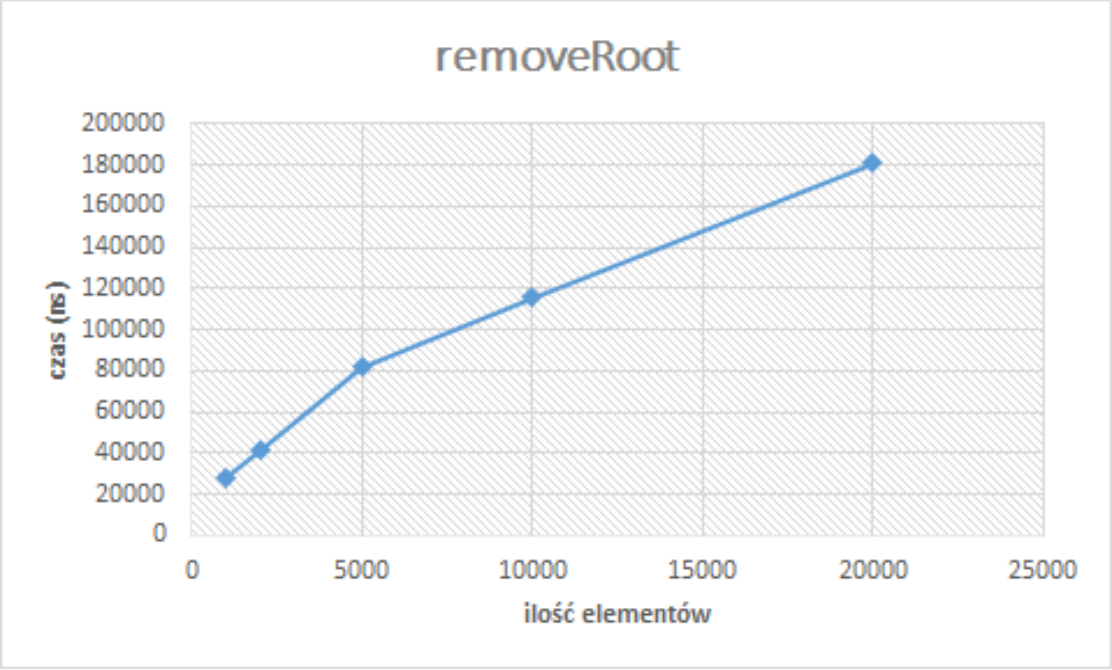


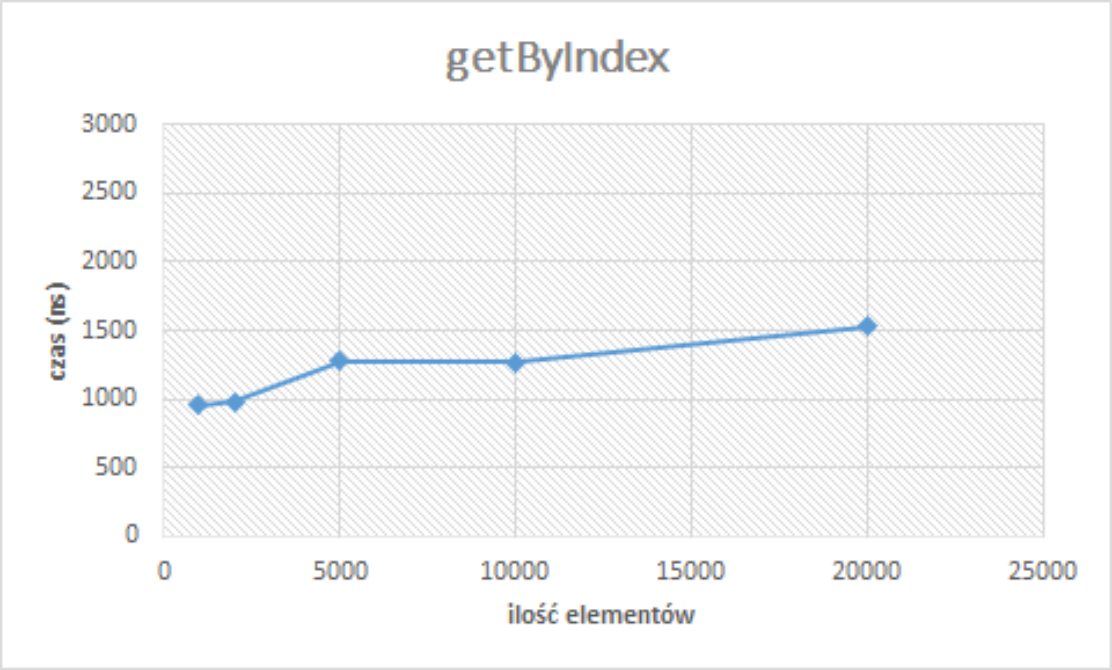
3.3 Kopiec binarny maksymalny

czas (ns)	add	removeRoot	getByIndex	getIndex
1000	91372	28200	958	1661442
2000	172865	41223	983	1708909
5000	404792	81736	1280	1736590
10000	766274	115536	1267	1782398
20000	1625143	180732	1534	1963295

Tabela 3: Uśrednione wyniki eksperymentu na kopcu


add
1800000
1600000
1400000
1200000
1000000
800000
600000
400000
200000
0
czas (ns)
0
5000
10000
15000
20000
25000
ilość elementów


removeRoot
200000
180000
160000
140000
120000
100000
80000
60000
40000
20000
0
czas (ns)
0
5000
10000
15000
20000
25000
ilość elementów

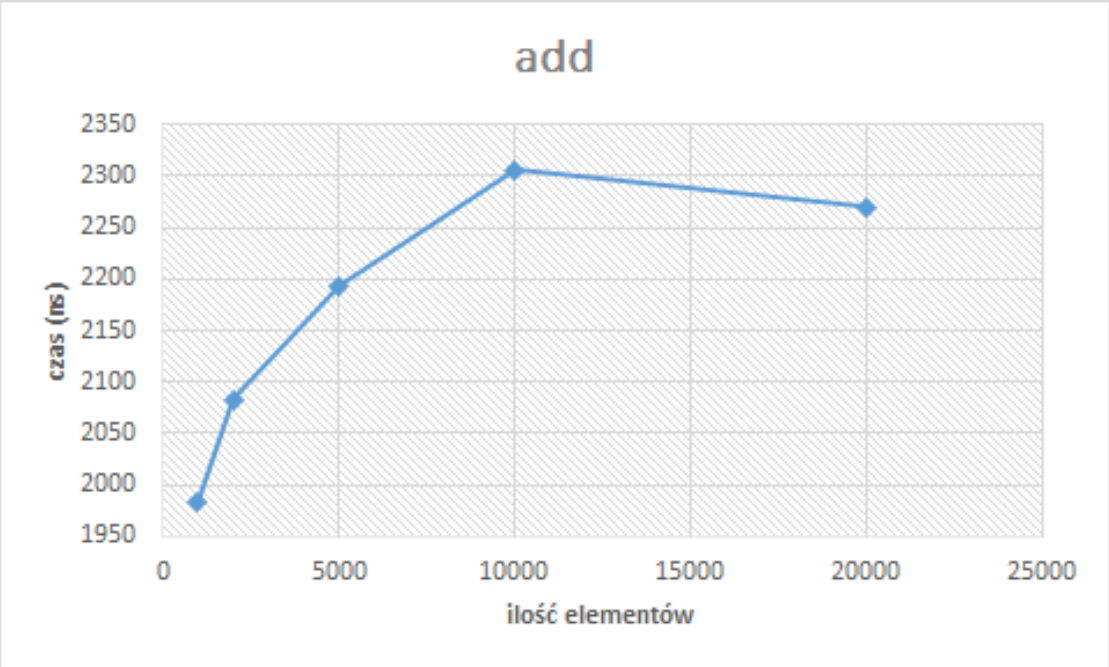

getByIndex
3000
2500
2000
1500
1000
500
0
czas (ns)
0
5000
10000
15000
20000
25000
ilość elementów

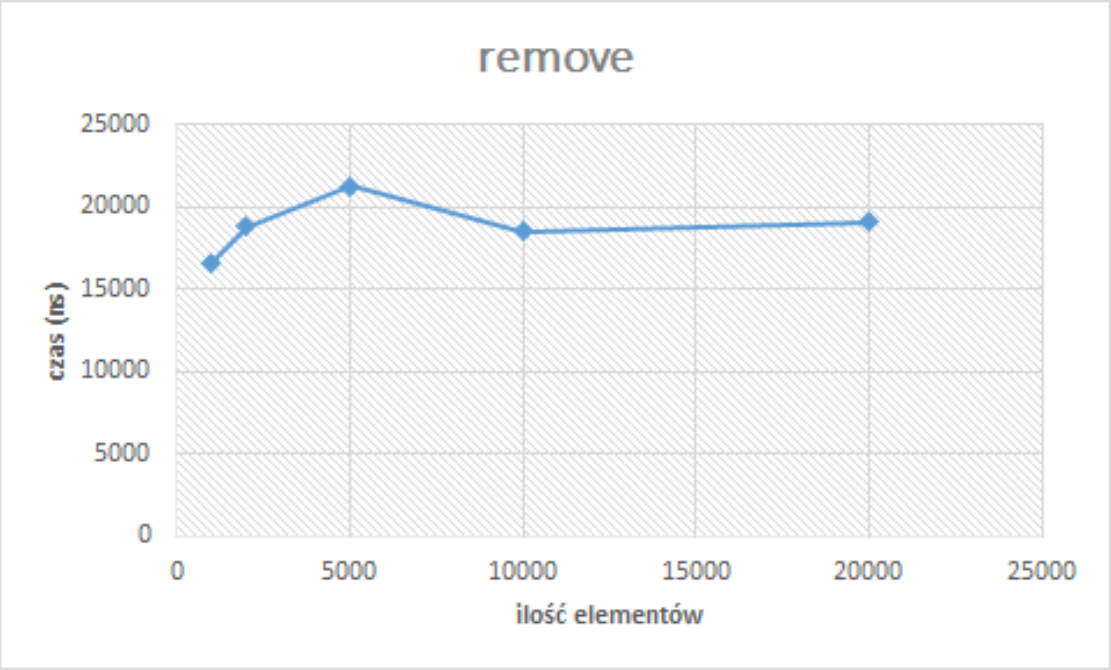

getIndex
2000000
1950000
1900000
1850000
1800000
1750000
1700000
1650000
1600000
czas (ns)
0
5000
10000
15000
20000
25000
ilość elementów

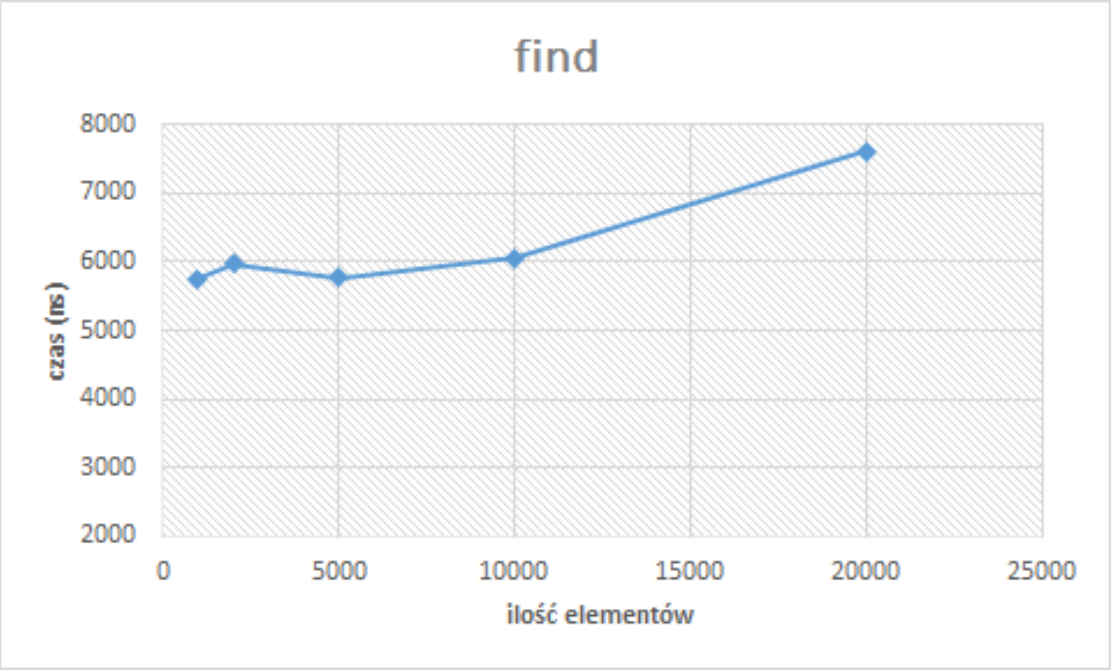
3.4 Drzewo czerwono-czarne

czas (ns)	add	remove	find
1000	1984	16638,32	5752,372
2000	2082,833	18794,33	5964,833
5000	2193,833	21271,5	5776
10000	2306	18503,33	6062,833
20000	2269,891	19062,5	7616,333

Tabela 4: Uśrednione wyniki eksperymentu na drzewie







4 Wnioski

Zaimplementowane struktury działają poprawnie i w większości spełniają zakładane oczekiwania. Głównym powodem, dla którego funkcje, w których należało znaleźć pewien

element (w tym również znaleźć i usunąć) różnią się wykresem z powodu użycia funkcji *throw* w połączeniu z wyjątkami, że dany indeks lub wartość nie zostały znalezione. Funkcja throw zwiększa mocno czas, który dana metoda potrzebuje na wykonanie swojej pracy. Pozostałe nieścisłości mogą być spowodowane zbyt małą liczbą prób (zaledwie 100) lub użyciem zbyt małego zróżnicowania (liczby z zakresu 1-1000) przy znacznie większej ilości liczby elementów (liczby z zakresu 1000-20 000).

4.1 Wady i zalety zaimplementowanych struktur

W kwestii efektywności, każda ze struktur ma swoje zalety, jak i wady:

- elementy tablicy możemy szybko uzyskać za pomocą indeksów, jednak pozostałe metody mają złożoność obliczeniową O(n). Dla wielu wartości potrzebuje również zarezerwować dużo ciągłego miejsca w pamięci
- elementy listy możemy szybko dodać i zdjąć z końca oraz początku. Nie potrzebujemy również nieprzerwanej pamięci. Jednak jeśli potrzeba coś ze środka - złożoność to O(n)
- kopiec pozwala na uzyskanie w prosty sposób wartości maksymalnej. Jednak oparcie go na bazie dynamicznej tablicy powoduje, że wszystkie operacje rosną z O(lg n) do złożoności liniowej
- drzewo dzięki swojemu skomplikowaniu jest jedną z najefektywniejszych zaimplementowanych struktur. Pozwala zarówno na szybkie dodanie, znalezienie, jak i usunięcie (wykres usuwania jest zniekształcony z powodów powyżej). Wadą jest jednak trudność w jego implementacji

5 Bibliografia

1. "Wprowadzenie do algorytmów" Clifford Stein, Ron Rivest i Thomas H. Cormen