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, 26 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 (przynajmniej 1000 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 struktury to następująco: 1000, 2000, 3000, 4000, 5000, 6000, 7000, 10000, 15000 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 Tabela dynamiczna

2.1 Wstęp teoretyczny

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.

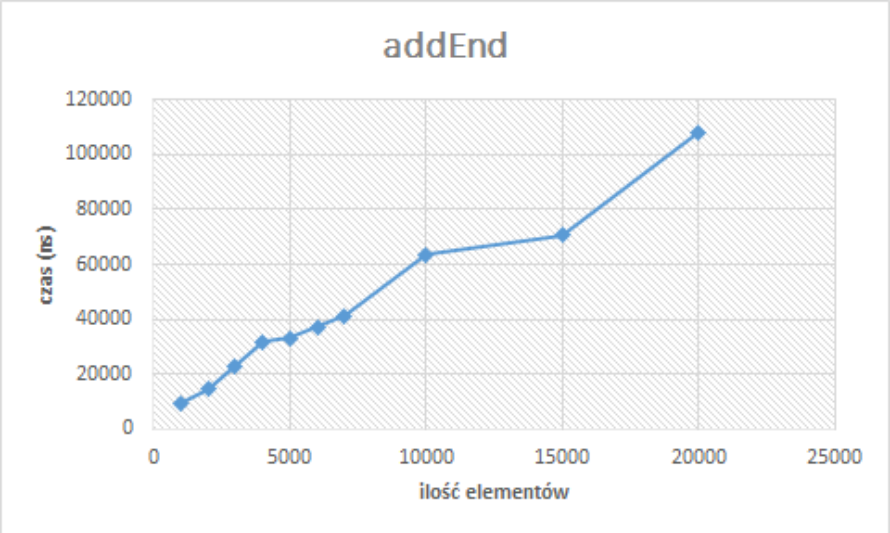
2.2 Uśrednione wyniki eksperymentów

czas(ns)	addEnd	addFront	addAtIndex	removeEnd	removeFront	removeAtIndex	getByIndex	getIndex
1000	9451,333	10256,73	8162,333	9018,533	8874,333	9024,4	796,6	6418,733
2000	14628,6	15214,27	14834,2	18179,2	14376,13	15384,27	883,6	12987,33
3000	23197,8	21532,2	21046,6	27544,27	23107,73	21386,73	988,8667	30194,13
4000	31719,6	32260,6	27754,2	29022	29945,8	30153,93	964,2667	24962,87
5000	33309,27	32519,73	31958,87	34728,67	33875,27	32487,33	887,0667	33113,73
6000	37147,76	34692,84	33228,64	38370,88	32765,4	33208,24	792,16	33468,6
7000	41377,83	42640,03	43045,9	41066,2	40991,43	41785,73	838,7667	39982,4
10000	63473,3	50240,8	48504,9	67877,2	48621	52258,8	809,6	46987,5
15000	70576,73	76800,73	75705,27	73010	72294,6	73049,6	849	72242,47
20000	108051,4	110963	111266,4	104872,9	114327,7	118106,4	1030,6	107494

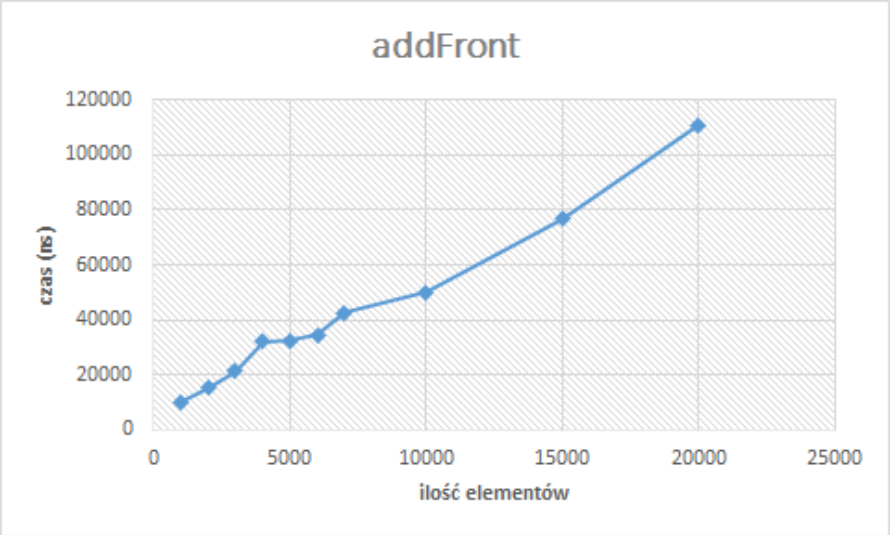
Tabela 1: Średnie wyniki w postaci tabeli na tabeli

2.3 Zaimplementowane operacje:

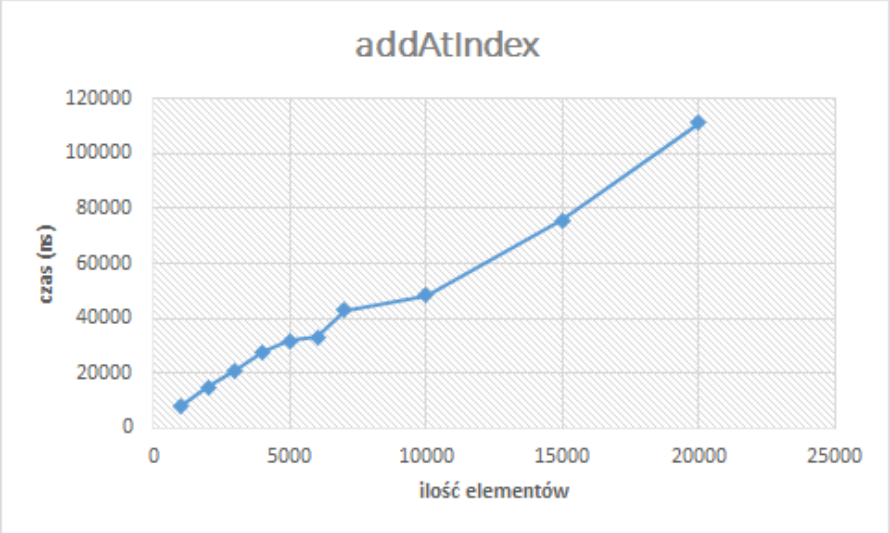
2.3.1 Dodanie elementu na koniec



2.3.2 Dodanie elementu na początek

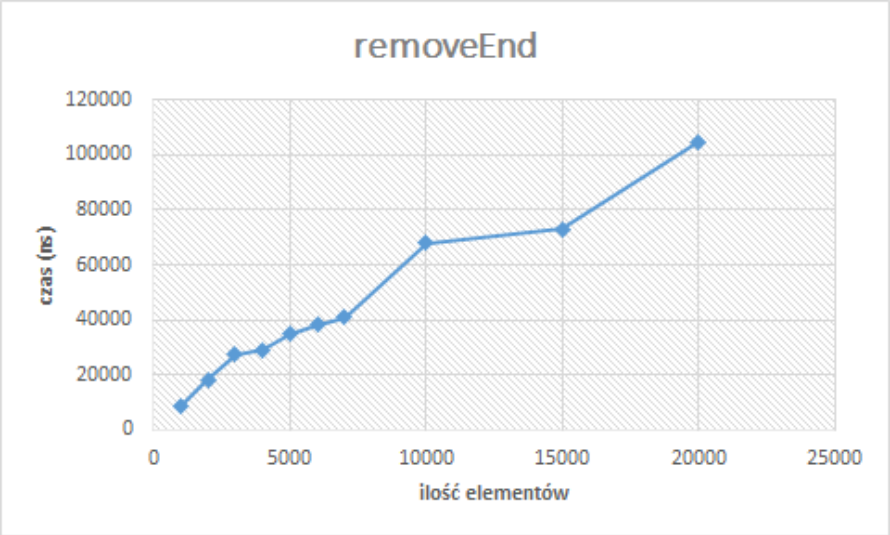


2.3.3 Dodanie elementu na dowolne miejsce

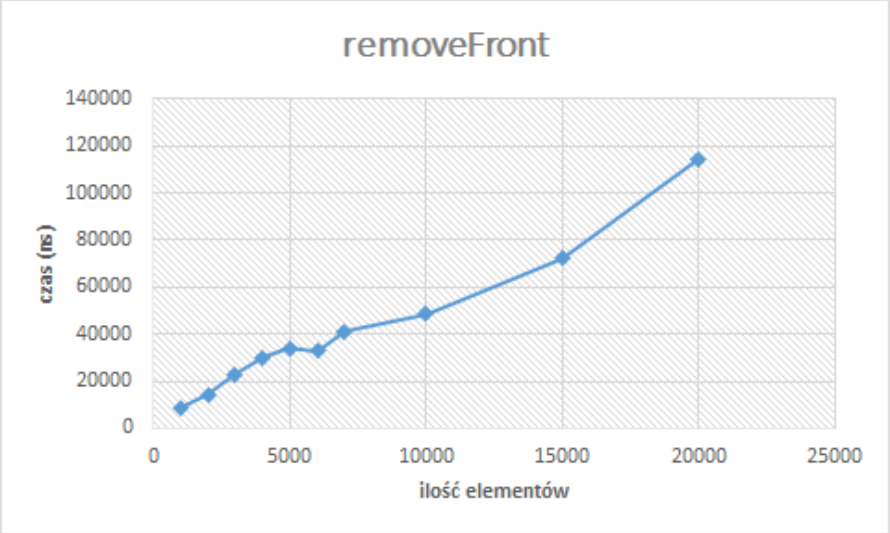


Dowolne miejsce zostało tu przyjęte jako najbardziej pesymistyczne dla tej funkcji - przedostatnie.

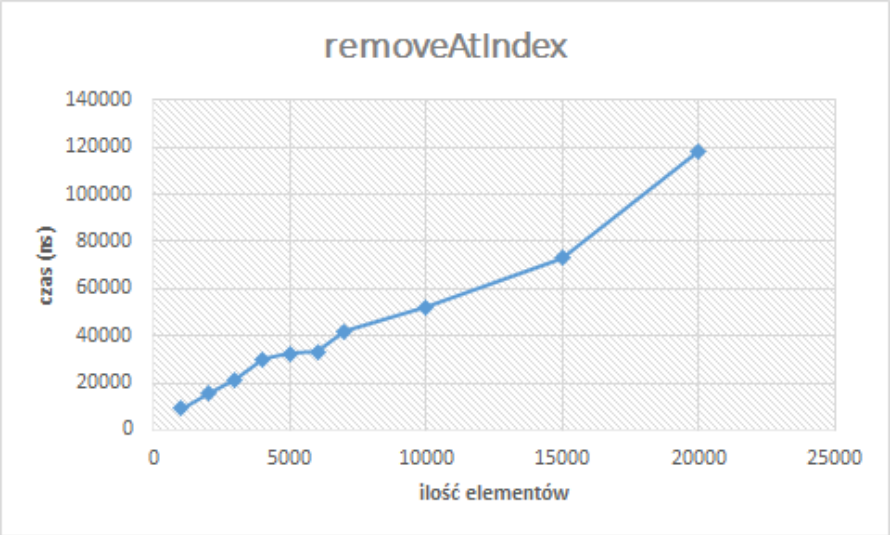
2.3.4 Usunięcie elementu z końca



2.3.5 Usunięcie elementu z początku

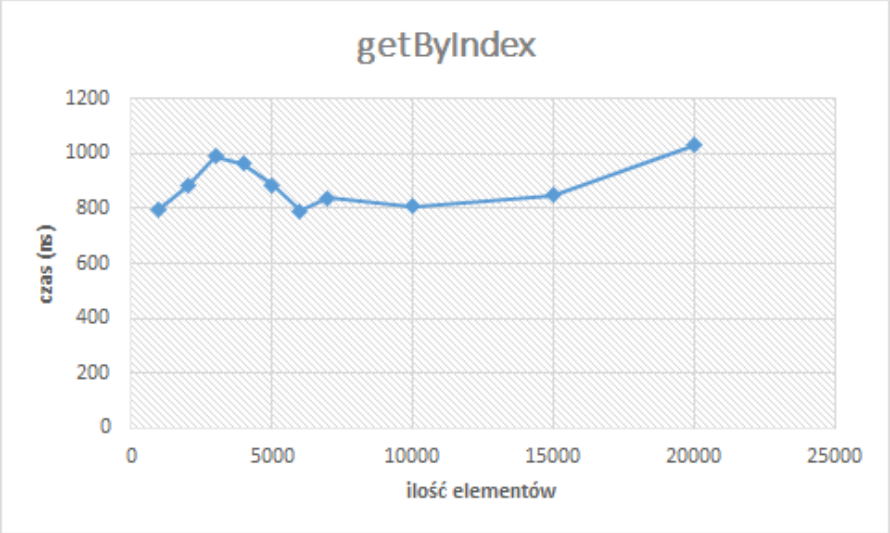


2.3.6 Usunięcie elementu z dowolnego miejsca

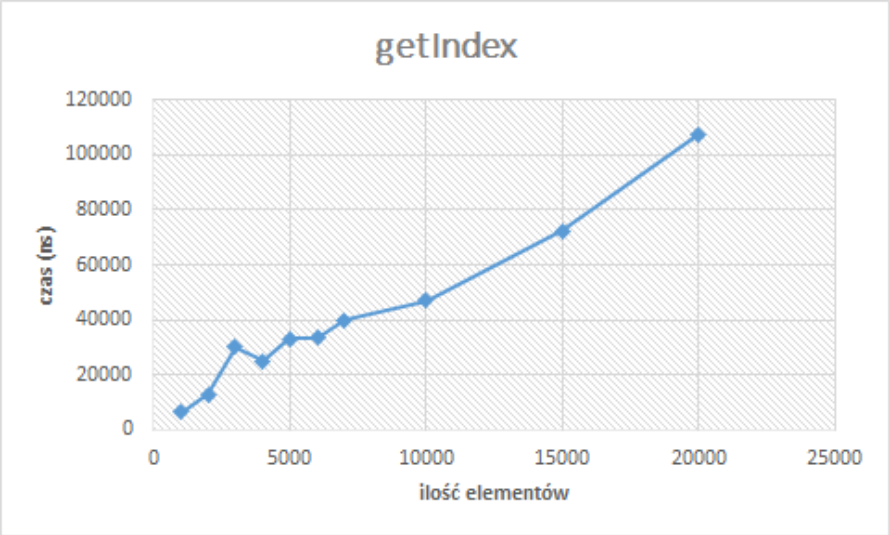


Dowolne miejsce zostało tu przyjęte jako najbardziej pesymistyczne dla tej funkcji - przedostatnie.

2.3.7 Wyszukanie elementu o podanym indeksie



2.3.8 Wyszukanie elementu o podanym kluczu



Dowolny element zostało tu przyjęte jako najbardziej pesymistyczny dla tej funkcji - ostatni.

3 Lista dwukierunkowa

3.1 Wstęp teoretyczny

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.

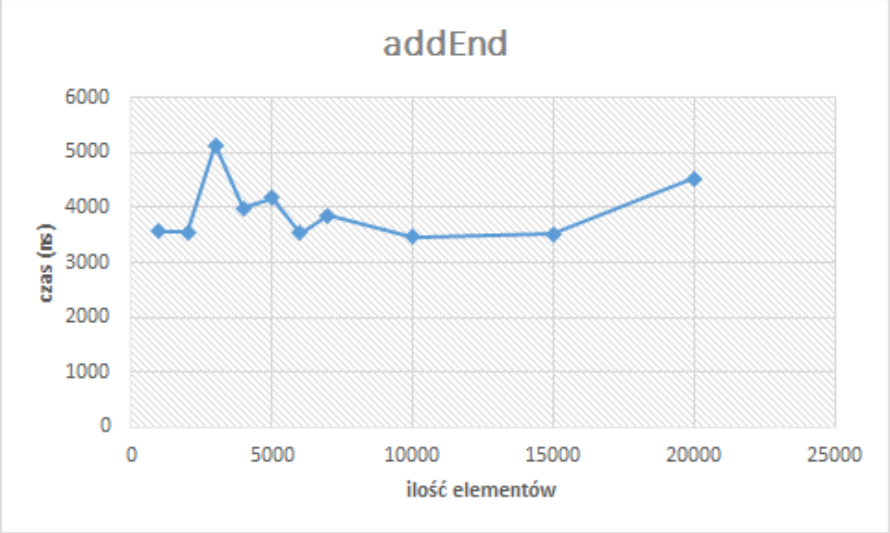
3.2 Uśrednione wyniki eksperymentów

czas(ns)	addEnd	addFront	addAtIndex	removeEnd	removeFront	removeAtIndex	getByIndex	getIndex
1000	3575,649	3600,599	14326,15	3628,743	3302,994	12391,82	7812,974	20065,27
2000	3553,892	3834,132	27064,07	3471,257	3199,202	23605,79	15823,15	40421,36
3000	5136,727	4284,032	43149,7	3940,12	3845,309	34958,08	28638,92	69161,88
4000	3995,4	3905	48146,2	3569,6	3474,8	37775,4	31206,4	75424,6
5000	4183,8	4068,8	67399,2	4176,4	3865,4	53669,2	47022,8	104413
6000	3527,4	3311,6	63229,2	3188,4	3154,2	48112,6	47249,6	101681,2
7000	3857,4	3619,2	77688,2	3372	3275	59622,4	56047	122129,2
10000	3466	3302,2	94276,8	3020,4	2856,2	73084,4	70822,4	156249,4
15000	3523,2	3311,8	155741	3010,4	2862,8	119570,6	126521	256071,8
20000	4524,048	3610,621	254814,2	3972,144	3278,958	209089,2	223605,4	412358,5

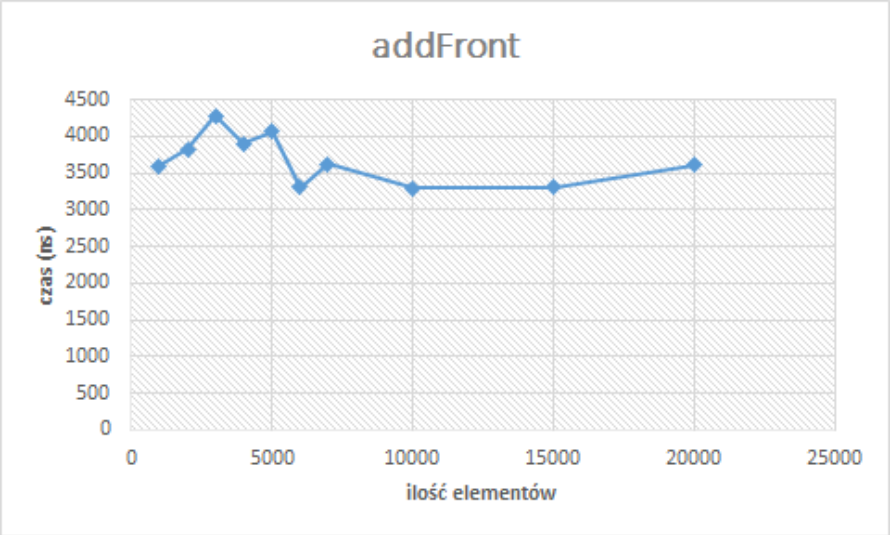
Tabela 2: Uśrednione wyniki eksperymentu na liście

3.3 Zaimplementowane operacje:

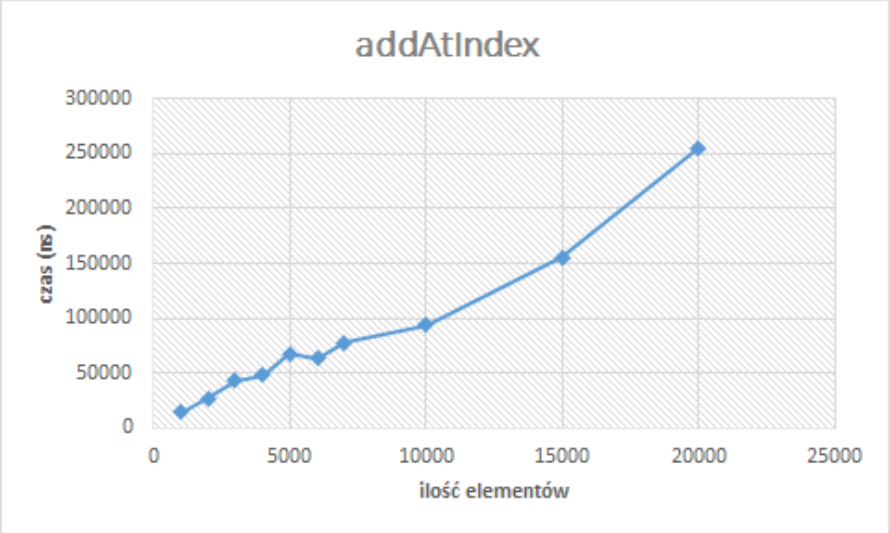
3.3.1 Dodanie elementu na koniec



3.3.2 Dodanie elementu na początek

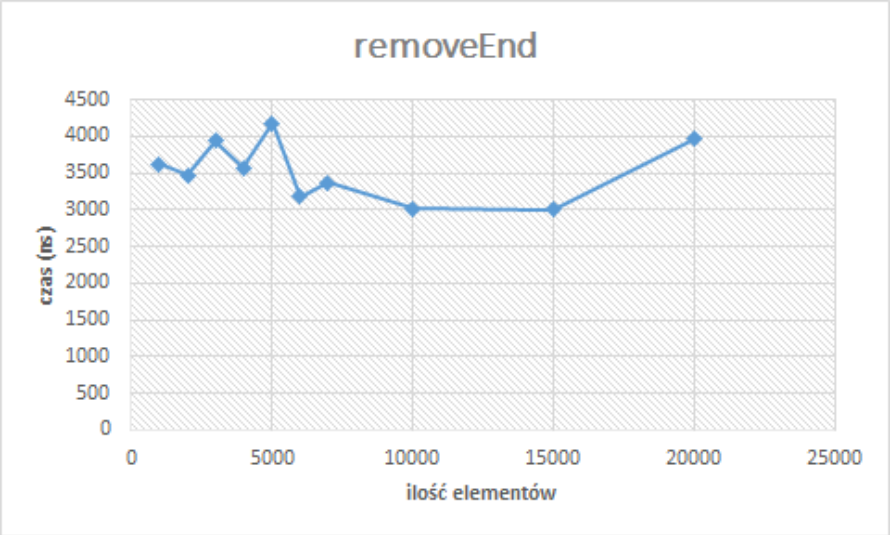


3.3.3 Dodanie elementu na dowolne miejsce

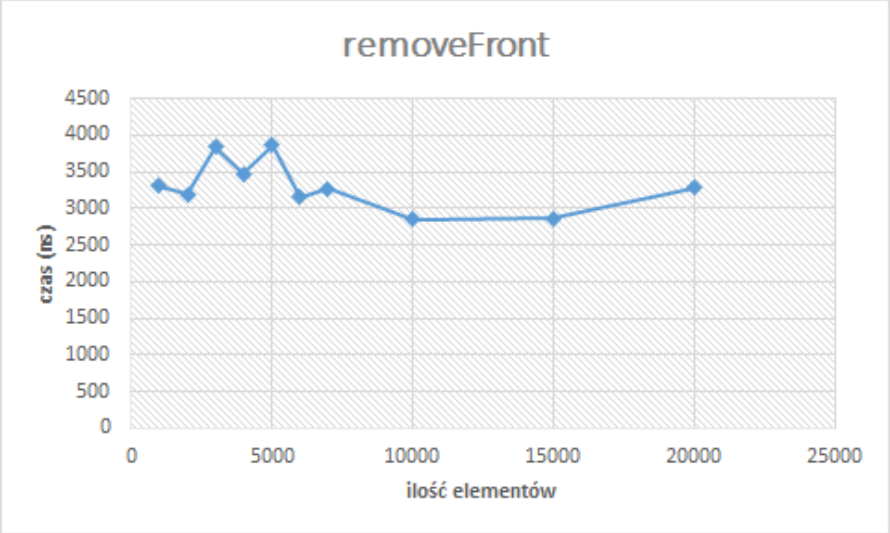


Dowolne miejsce zostało tu przyjęte jako najbardziej pesymistyczne dla tej funkcji - środkowy indeks.

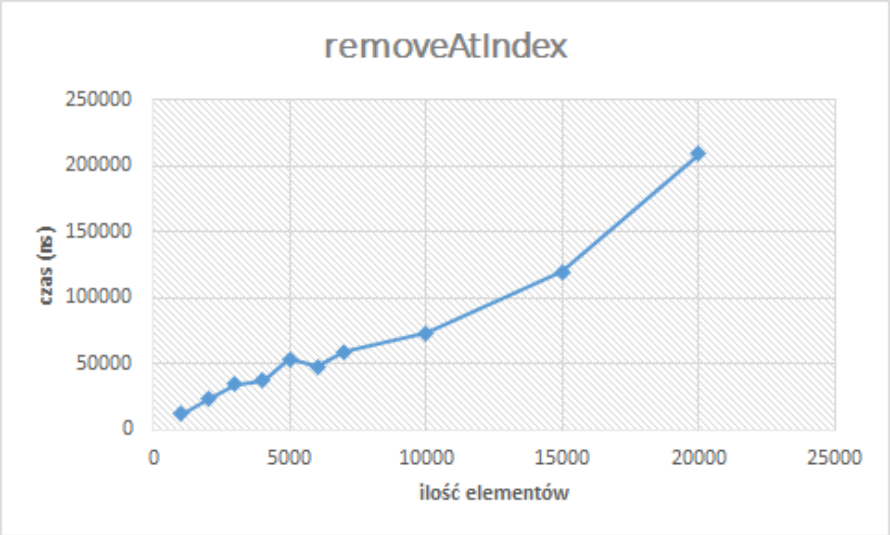
3.3.4 Usunięcie elementu z końca



3.3.5 Usunięcie elementu z początku

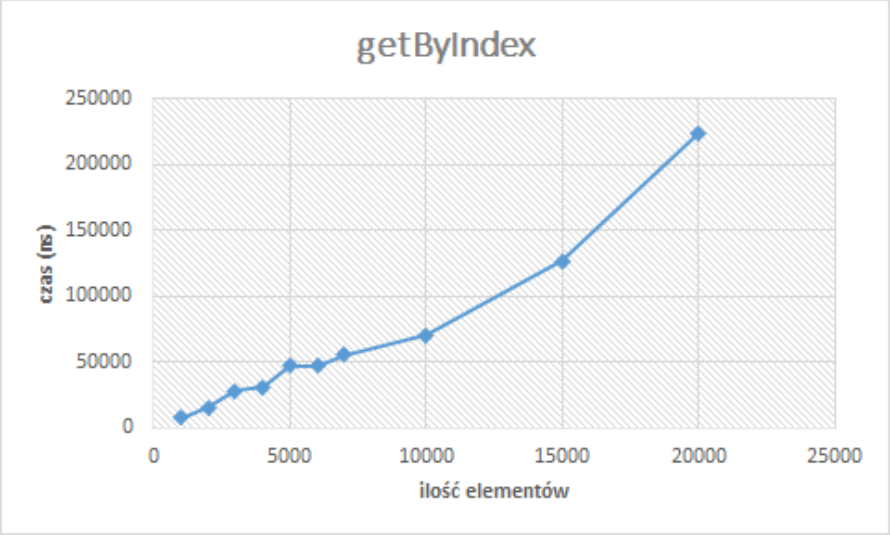


3.3.6 Usunięcie elementu z dowolnego miejsca



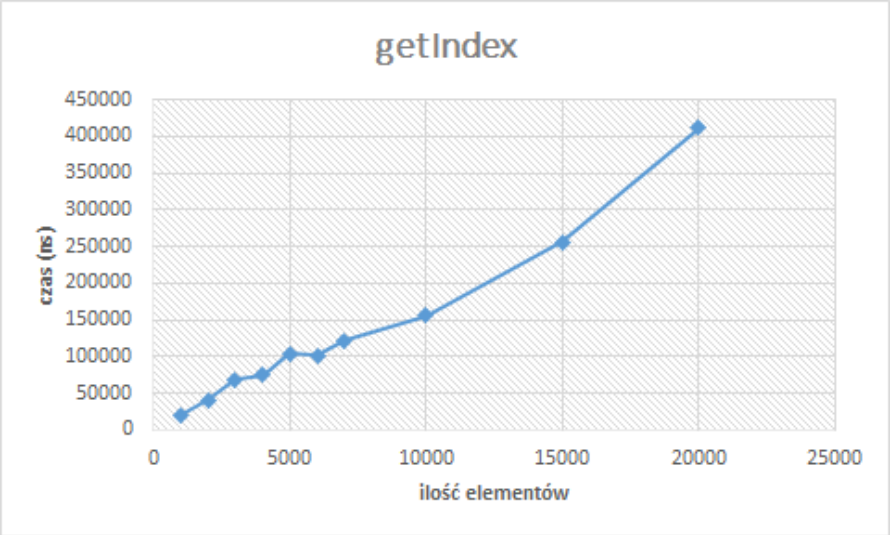
Dowolne miejsce zostało tu przyjęte jako najbardziej pesymistyczne dla tej funkcji - środkowy indeks.

3.3.7 Wyszukanie elementu o podanym indeksie



Dowolne miejsce zostało tu przyjęte jako najbardziej pesymistyczne dla tej funkcji - ostatni indeks.

3.3.8 Wyszukanie elementu o podanym kluczu



Dowolny element został tu przyjęty jako najbardziej pesymistyczne dla tej funkcji - ostatni.

4 Kopiec binarny maksymalny

4.1 Wstęp teoretyczny

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.

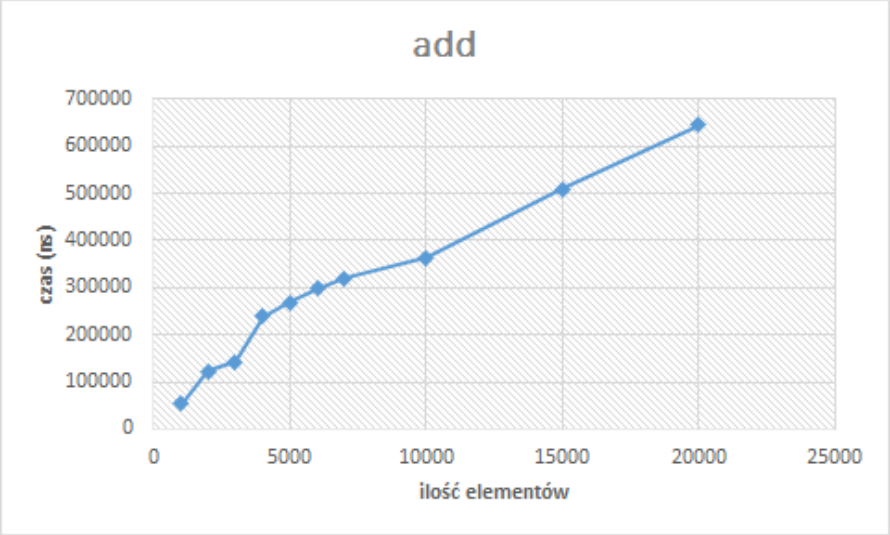
4.2 Uśrednione wyniki eksperymentów

czas (ns)	add	removeRoot	getByIndex	getIndex
1000	54226,6	10694,6	557,8	5110,8
2000	123515,6	18166,8	662,2	11433,6
3000	142220	18244,8	512	12955
4000	238452,4	27183,6	627,4	19963,6
5000	269419,4	28115,8	616,4	24119,6
6000	297971	30733,4	590	26960,2
7000	319381,4	32672,4	585,6	28901,8
10000	362307,6	39906,6	589,6	36350,2
15000	509609,4	56237,2	540,2	53696,2
20000	644970,8	72542,2	484,4	70647,4

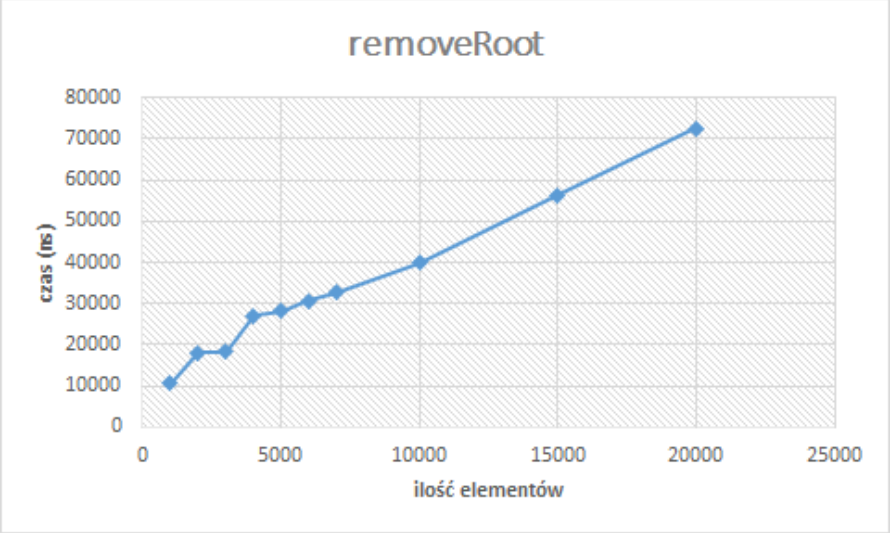
Tabela 3: Uśrednione wyniki eksperymentu na kopcu

4.3 Zaimplementowane operacje:

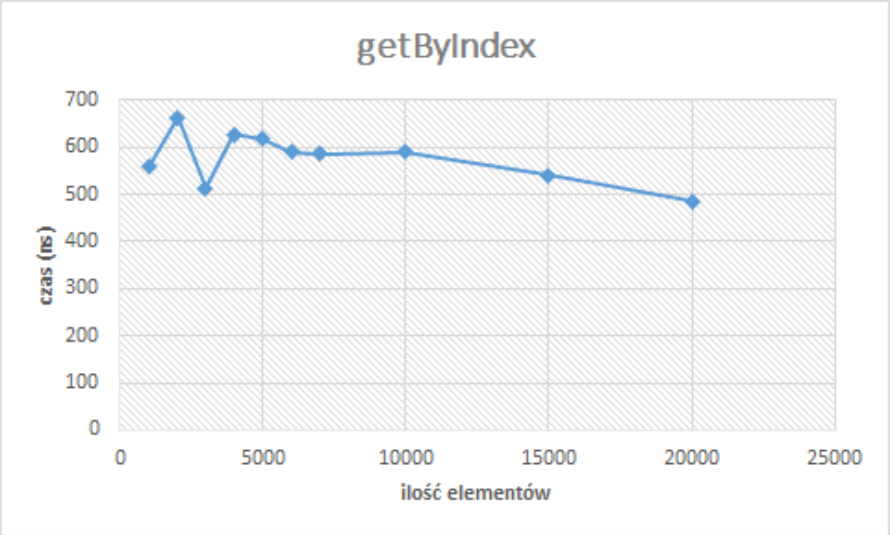
4.3.1 Dodanie elementu



4.3.2 Usunięcie korzenia

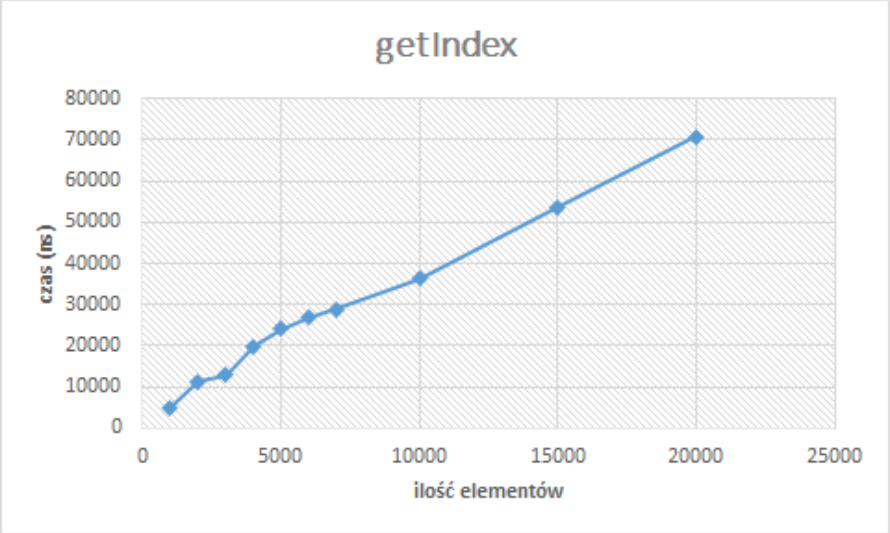


4.3.3 Wyszukanie elementu o podanym indeksie



Dowolne miejsce zostało tu przyjęte jako najbardziej pesymistyczne dla tej funkcji - ostatni indeks.

4.3.4 Wyszukanie elementu o podanym kluczu



4.4 Informacje dodatkowe na temat otrzymanych wyników

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)**.

5 Drzewo czerwono-czarne

5.1 Wstęp teoretyczny

Drzewo czerwono-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 lewego 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.

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

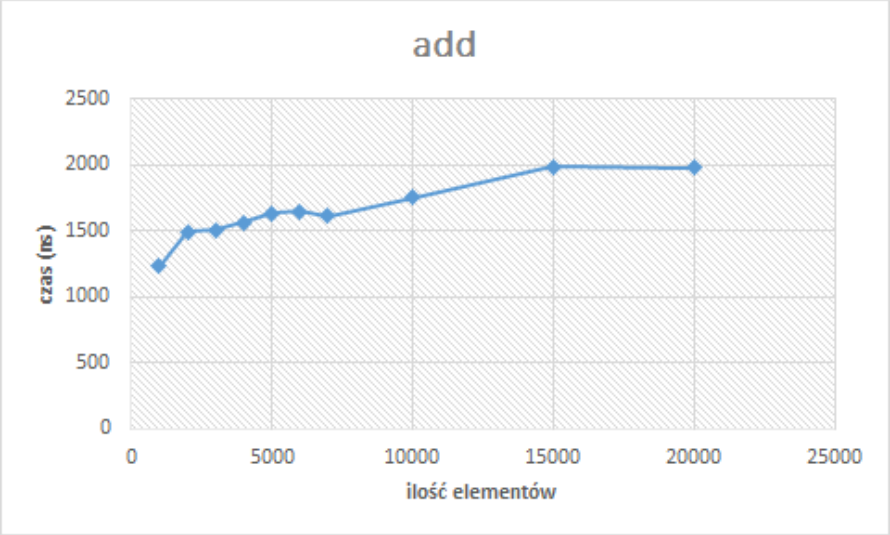
5.2 Uśrednione wyniki eksperymentów

czas (ns)	add	remove	find
1000	1230,95	3191,3	1683,9
2000	1492,55	3576,85	1976,95
3000	1508,1	3904,75	1843,85
4000	1562,22	4109,4	1854,425
5000	1630,9	4498,4	2172,4
6000	1648,96	4595,22	2320,18
7000	1610,7	4312,233	2442,5
10000	1752,2	4596,1	2474,3
15000	1985,9	5079,3	2723,2
20000	1977	5709,9	3124,3

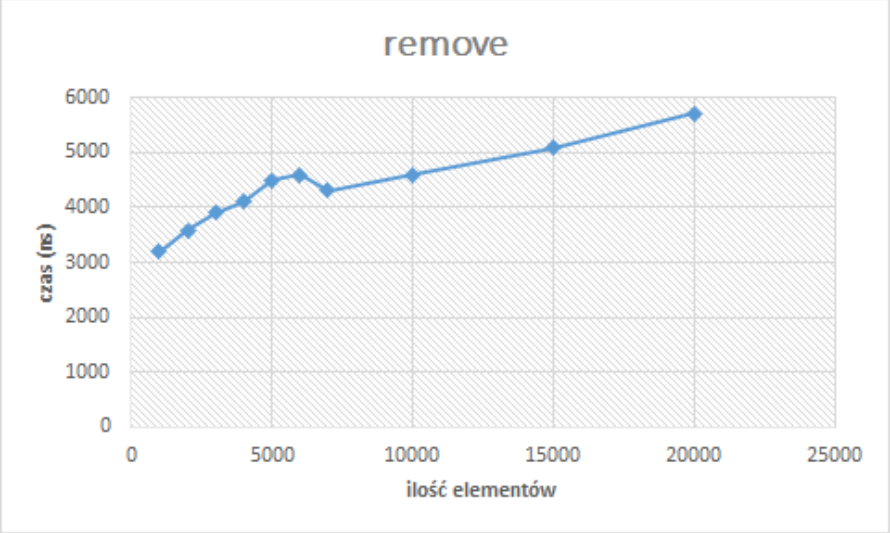
Tabela 4: Uśrednione wyniki eksperymentu na drzewie

5.3 Zaimplementowane operacje:

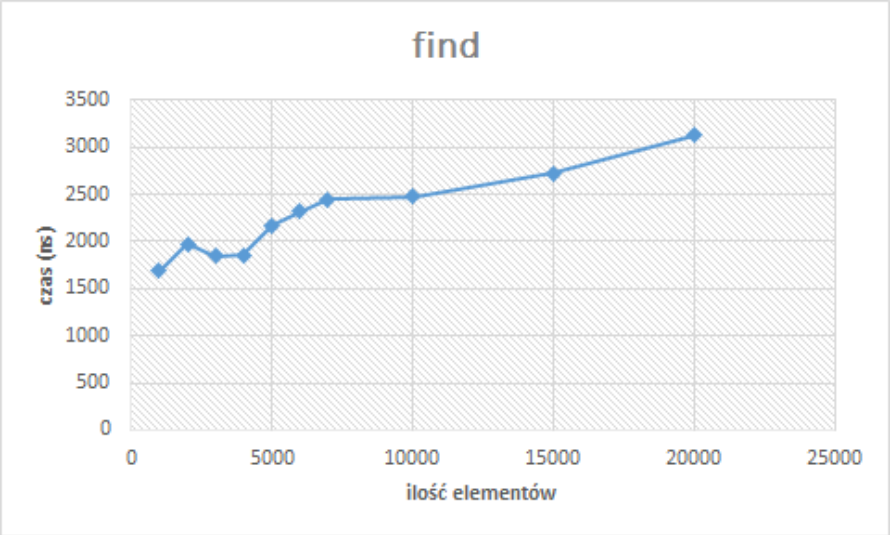
5.3.1 Dodanie elementu



5.3.2 Usunięcie korzenia



5.3.3 Wyszukanie elementu



6 Wnioski

Zaimplementowane struktury działają poprawnie i spełniają zakładane oczekiwania. Nieścisłości mogą być spowodowane zbyt małą liczbą prób (zaledwie 1000 dla każdej ilości elementów) lub użyciem zbyt małego zróżnicowania (liczby z zakresu 1-10 000). Dla ponawianych prób średni wynik poprawiał się oraz zbliżał bardziej do oczekiwanego

6.1 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

7 Bibliografia

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