STRUKTURY DANYCH I ZŁOŻONOŚĆ OBLICZENIOWA

Zadanie projektowe nr 1: Badanie efektywności operacji dodawania (wstawiania), usuwania oraz wyszukiwania elementów w podstawowych strukturach danych

Autor: Bartosz Rodziewicz, 226105

Prowadzący: Dr inż. Jarosław Rudy

Grupa: Wtorek, 11:15

1. Wstęp teoretyczny

a. Złożoność obliczeniowa

Miara ilości zasobów potrzebnych do rozwiązania problemów obliczeniowych. Rozważanymi zasobami są takie wielkości jak czas, pamięć lub liczba procesorów.

b. Tablica

Rodzaj struktury danych w której wszystkie elementy alokowane są w jednym spójnym bloku pamięci.

i. Dodawanie

Dodawanie do tablicy polega na zalokowaniu nowej o element większej tablicy, przepisaniu elementów o indeksie mniejszym niż na który dodajemy nową wartość (jeśli istnieją), wpisaniu naszej wartości i przepisaniu dalszych elementów, aż do końca tablicy (jeśli istnieją).

Złożoność czasowa (wynikająca z konieczności przepisania wszystkich elementów) wynosi **O(n)**.

ii. Usuwanie

Usuwanie elementu z tablicy polega na zalokowaniu nowej, o element mniejszej tablicy i przepisaniu wszystkich elementów z wyjątkiem usuwanego.

Złożoność czasowa (wynikająca z konieczności przepisania wszystkich elementów) wynosi **O(n)**.

iii. Wyszukiwanie

Z uwagi, że dane w tablicy nie muszą być w żaden sposób posortowane wyszukiwanie polega na manualnym przejściu przez każdy element i porównaniu jego wartości do wartości szukanej.

W najgorszym przypadku konieczne będzie przejście przez wszystkie elementy (lub za każdym razem, jeśli poza sprawdzeniem czy element jest, również zliczamy ilość wystąpień), więc złożoność czasowa wynosi **O(n)**.

c. Lista dwukierunkowa

Rodzaj struktury, w której każdy element jest osobno alokowany w pamięci. Element poza swoją wartością zawiera adres położenia poprzedniego i następnego elementu. W liście dwukierunkowej mamy bezpośredni dostęp do pierwszego i ostatniego elementu.

i. Dodawanie i usuwanie na początku, bądź końcu

Jako że mamy bezpośredni dostęp do pierwszego i ostatniego elementu, dodanie lub usunięcie któregoś z nich kompletnie nie zależy od pozostałych, więc złożoność czasowa wynosi **O(1)**.

Samo dodawanie pierwszego lub ostatniego elementu polega na podmienieniu zapamiętanej wartości pierwszego lub ostatniego elementu na ten nowy, a obecnego zapisanie jako adres następnego/poprzedniego.

Usunięcie działa podobnie, adres drugiego bądź przedostatniego zapisujemy jako pierwszy/ostatni i kasujemy informacje o poprzednim/kolejnym elemencie z niego.

ii. Dodawanie i usuwanie w dowolnym miejscu

W przypadku dodawania i usuwania elementów na dowolnym miejscu poza samym dodaniem/usunięciem (stworzeniem/usunięciem elementu i odpowiednim ustawieniem wskaźników następny/poprzedni) musimy jeszcze doskoczyć do odpowiedniego miejsca w liście. Tutaj, inaczej niż w przypadku tablicy, musimy zrobić to manualnie, przechodząc po kolejnych elementach.

Dla listy dwukierunkowej mamy to ułatwienia/przyśpieszenie, że do elementów znajdujących się w drugiej połowie listy nie musimy przechodzić od początku, a możemy od końca, co daje nam mniej elementów do „przejścia".

Z tego powodu złożoność czasowa tej operacji wynosi **O(n/2)**.

iii. Wyszukiwanie

Wyszukiwanie w liście, tak jak i tablicy, polega na manualnym przejściu przez każdy element i porównaniu jego wartości do wartości szukanej. Złożoność wynosi **O(n)**.

d. Kopiec

Jest to rodzaj struktury danych, oparty na drzewie, które posiada jeden warunek – w każdym węźle wartość rodzica, jest niemniejsza od wartości obu synów. Struktura ta opiera się na drzewie kompletnym, co powoduje, że najwygodniejszą metodą jej implementacji, jest implementacja na tablicy.

i. Dodawanie na końcu

Dodawanie elementu na końcu polega na dopisaniu tego elementu, do pierwszego wolnego liścia od lewej w ostatnim niepełnym rzędzie. Po tym dopisaniu, trzeba wykonać sprawdzenie w górę, czy jest zachowana zasada kopca.

Złożoność obliczeniowa sprawdzenia warunku kopca, dla nowo powstałego elementu, jest zależna od ilości poziomów drzewa, czyli wynosi O(log(n)).

Jednak z powodu wykonanej przeze mnie implementacji kopca na tablicy, złożoność ta wzrasta do **O(n)** z powodu czasu koniecznego na realokowanie tej tablicy.

ii. Dodawanie na początku bądź w dowolnym miejscu

Ciężko mówić o dodaniu elementu do kopca w innym miejscu niż na końcu – ta struktura nie służy do tego. W mojej implementacji jednak polega to na przesunięciu następnych elementów (tak jak w tablicy) o jeden w prawo, lub do następnego poziomu. Powoduje to całkowite zachwianie struktury kopca i wymaga wykonania sprawdzenia warunku kopca dla każdego elementu (sprawdzenie w górę, aż do korzenia).

Sprawdzenie to, posiada złożoność zależną od ilości poziomów drzewa, więc całkowita złożoność takiego dodawania wynosi **O(n log(n))**.

iii. Usuwanie

Każde usunięcie (z wyjątkiem usuwania na końcu) wymaga po usunięciu wykonania sprawdzenia warunku kopca dla każdego elementu, więc złożoność czasowa wynosi dla niego **O(n log(n))**. Dla usuwania na końcu złożoność wynosi **O(n)** z powodu czasu potrzebnego na realokacje tablicy.

iv. Wyszukiwanie

Wyszukiwanie, tak jak w poprzednich strukturach wymaga liniowego sprawdzenia każdego elementu i jego złożoność wynosi **O(n)**.

e. Drzewo BST (z algorytmem balansowania DSW)

Struktura oparta na drzewie, gdzie każde lewe poddrzewo każdego węzła zawiera elementy z kluczem mniejszym od klucza węzła, a prawe niemniejszym. Z uwagi na implementacje algorytmu balansującego DSW, drzewo to jest również drzewem kompletnym, ostatni poziom ma zapełniany od lewej.

i. Dodawanie

Dodawanie do drzewa polega szukaniu pierwszego wolnego liścia zgodnie z warunkiem drzewa tzn. próbujemy dodać na pozycję korzenia, jeśli jest pusty – dodajemy, jeśli zajęty sprawdzamy czy jego wartość jest większa czy mniejsza od wartości dodawanej i idziemy w lewo lub w prawo, powtarzamy czynności aż do znalezienia wolnego liścia.

Nie rozumiem idei indeksowania elementów drzewa i w mojej implementacji dodaje się element tylko i wyłącznie po wartości.

Złożoność samego dodawania, dzięki algorytmowi równoważenie wynosi O(log n). Jednak, że po dodaniu uruchamiany jest algorytm równoważenia złożoność ta wzrasta do **O(n)**.

ii. Usuwanie

Usuwanie wartości z drzewa polega najpierw na wyszukaniu go, a później usunięciu. Wyszukanie działa prosto – po prostu idziemy po kolejnych poziomach i porównujemy wartość. Usunięcie jednak zależy od ilości poddrzew elementu, który usuwamy:

1. *Usuwanie elementu bez poddrzew*
   Najprostsza sytuacja, po prostu usuwamy element i usuwamy wskaźnik na ten element u rodzica.

2. *Usuwanie z jednym synem*
   Syn wskakuje na miejsce usuwanego elementu.

3. *Usuwanie z dwoma synami*

   Najbardziej skomplikowana sprawa – na miejsce usuwanego elementu wskakuje najmniejszy element z prawego poddrzewa.

Złożoność czasowa usuwania zależy od wysokości drzewa – O(log n), jednak z uwagi, że po usunięciu uruchamiany jest algorytm DSW, złożoność ta wzrasta do **O(n)**.

iii. Wyszukiwanie

Sposób wyszukiwania już po krótce opisałem w poprzednich punktach. Idziemy do korzenia, porównujemy jego wartość z wartością szukaną, jeśli się zgadza – znaleźliśmy obiekt, jeśli jest wartość szukana jest mniejsza – idziemy do lewego poddrzewa, jeśli większa idziemy do prawego. Po przejściu do kolejnego poziomu znowu porównujemy i albo znaleźliśmy element, albo przechodzimy do kolejnego poddrzewa.

Złożoność przeszukiwania drzewa zależy od wysokości, czyli jest równa **O(log n)**.

2. Implementacja programowa struktur

a. Klasa Programu

Klasa ta odpowiada za wyświetlanie menu i podejmowanie w nim działań. W większości zawiera lekko przerobiony kod napisany przez dr Mierzwę z przykładu jak powinien wyglądać przykładowy interfejs programu realizującego to zadanie.

b. Klasa Struktury

Jest to klasa praktycznie czysto wirtualna, po której dziedziczą wszystkie kolejne struktury danych. Wprowadziłem ją, aby znacznie uprościć klasę programu (wystarcza stworzenie jednego menu dla wszystkich struktur), jak i zwiększyć czytelność i porządek w kodzie.

Zawiera deklarację wirtualnych metod które powinny być zawarte każdej dziedziczącej strukturze.

c. Klasa Tablicy

Klasa realizująca strukturę dynamicznej tablicy. Poza implementacją każdej wirtualnej metody z klasy Struktury zawiera jedynie jedno prywatne pole wskaźnika na początek dynamicznej tablicy.

d. Klasa Listy

Klasa realizująca listę dwukierunkową. Posiada w sobie pomocniczą klasę pojedynczego elementu, z którego lista ta jest zbudowana.

Dodatkowo posiada wymagane prywatne wskaźniki na pierwszy, jak i ostatni element oraz kilka prywatnych metod pomagających wykonać dodawanie i usuwanie elementów.

Już po skończeniu tej klasy zauważyłem, że implementacja metody dodającej, jak i usuwającej powinna zostać podzielona na więcej prywatnych podmetod, aby zachować

większy porządek w kodzie, jednak nie chciałem już przeformatowywać tej klasy i zostało tak jak jest.

Dodatkowo pod koniec zauważyłem, że metody:

- dodająca z pliku (gdzie pomocniczo wykorzystuje std::vector),
- generowania losowych wartości
- testowania struktury

są napisane w ten sam sposób w każdej klasie i można by je przenieść jako uniwersalne dla każdej struktury.

e. Klasa Kopca

Jako jedyna z klas struktur nie dziedziczy bezpośrednio z klasy Struktury, a z klasy Tablicy, ponieważ moja realizacja kopca jest wykonana na tablicy.

Poza niektórymi przedefiniowanymi metodami z tablicy posiada metody odpowiedzialne za przywracanie zasady kopca, jak i uzyskiwania indeksu rodzica, czy poziomu, na którym dany indeks się znajduje.

f. Klasa drzewa BST

Jest to najbardziej skomplikowana klasa w całym programie, ponieważ poza realizacją drzewa zawiera również metody do algorytmu DSW.

Posiada w sobie również deklarację podklasy reprezentującej obiekty, z których drzewo się składa.

Zaimplementowane przeze mnie metody do dodawania i usuwania elementów wykorzystują algorytm rekurencyjny, metoda do wyszukiwania algorytm iteracyjny.

Algorytm DSW podzieliłem na kilka podmetod, w tym dwie główne realizujące dwa etapy – etap tworzenia listy z drzewa i drugi etap przywracania drzewa.

g. Klasa licznika

Ostatnia główna klasa w programie. Jest ona reprezentacją licznika QueryPerformanceCounter wykorzystywanego przeze mnie do pomiaru czasu trwania poszczególnych algorytmów.

Praktycznie w całości zawiera kod ze strony podanej w treści zadania dotyczącej tego licznika.

3. Metoda testowania i plan eksperymentu

Testować moje algorytmy zamierzam za pomocą licznika z klasy licznik (QueryPerformanceCounter).

Testy będą wykonywane na trzech rozmiarach struktur:

- Zawierającej 50 elementów,
- Zawierającej 10 000 elementów

- Oraz zawierającej 20 000 elementów.

Dla każdego rozmiaru będę wykorzystywał trzy rodzaje zakresów danych:

- [0, 100],
- [0, 16383],
- [0, 32767]

Daje mi to 9 przypadków do testów.

Test polega na stworzeniu struktury z konkretnego przypadku za pomocą metody generowania, uruchomienie licznika, wykonanie testowanego algorytmu (np. dodanie do tablicy wartości z zakresu konkretnego przypadku), zapisanie wyniku, cofnięcie operacji, i kilkukrotne powtórzenie jej.

Powtarzać zamierzam 10 razy i po każdy wynik zapisuję do pliku.

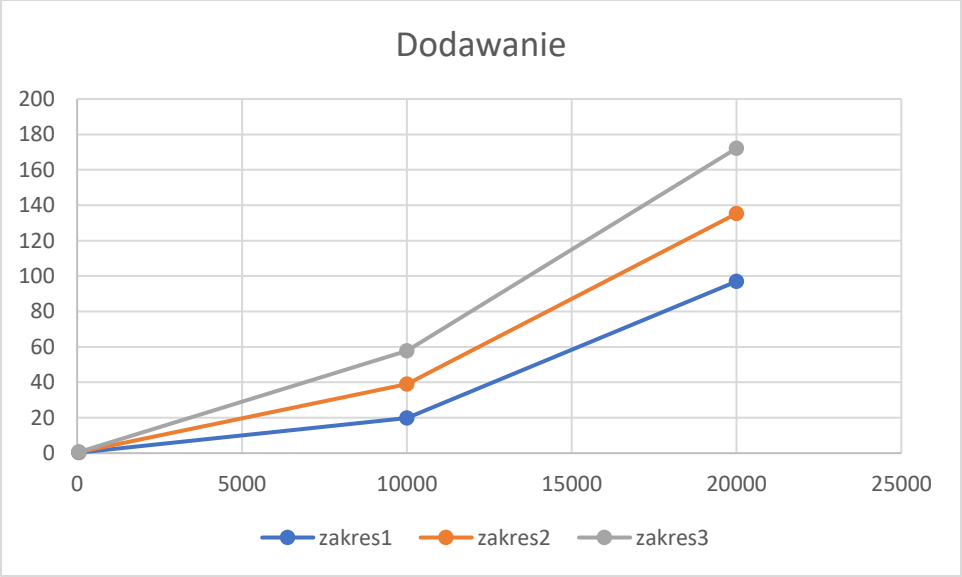
Później z tych danych wyliczam średnią dla każdej operacji i tworzę wykresy, które załączam w dalszej części dokumentu. Do wyliczenia średniej pozbyłem się wartości, które bezsprzecznie były błędne.

Wyniki pomiarów podane są w mikrosekundach.

Na sam koniec dokonuję porównania poszczególnych struktur w przełożenie na konkretne operacje.

4. Wyniki testów

a. Tablica

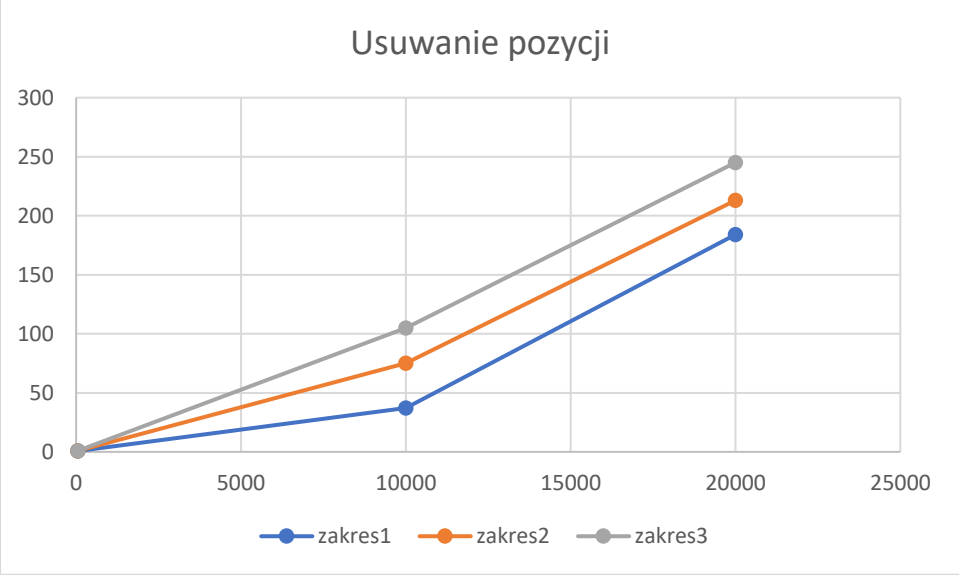


Wykres potwierdza złożoność O(n).

		Usuwanie wartości	

	50	10000	20000
zakres1	0.972482	3682.569360	40661.193580
zakres2	1.662630	132.696724	643.406573
zakres3	1.286186	155.204974	761.568352

W przypadku usuwania widzimy nieliniową zależność w przypadku dużej liczby elementów, a małego zakresu. Jest to spowodowane tym, że wartość, którą usuwamy występuje w strukturze wiele bardzo wiele razy, a algorytm usuwa, każde wystąpienie po kolei.



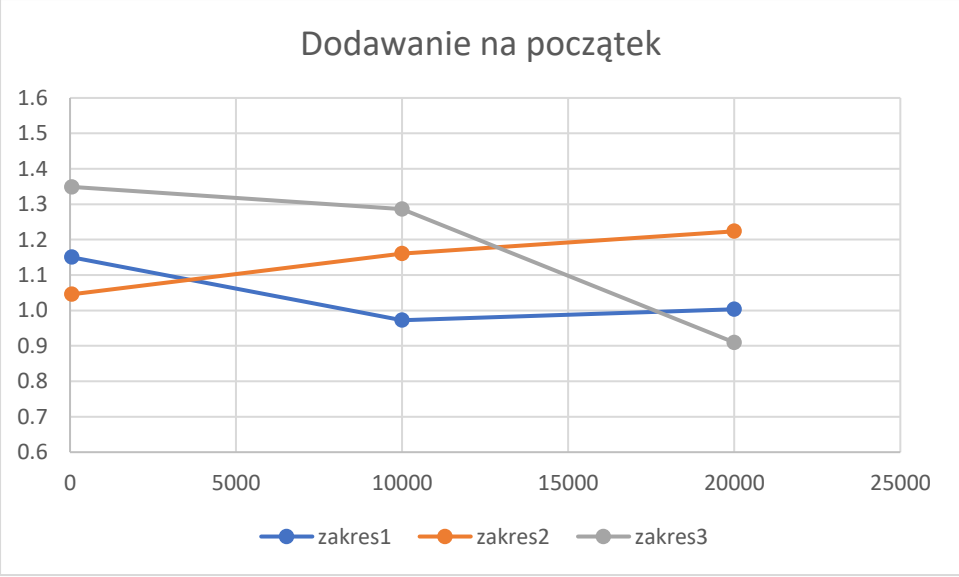
Wykres potwierdza złożoność O(n).

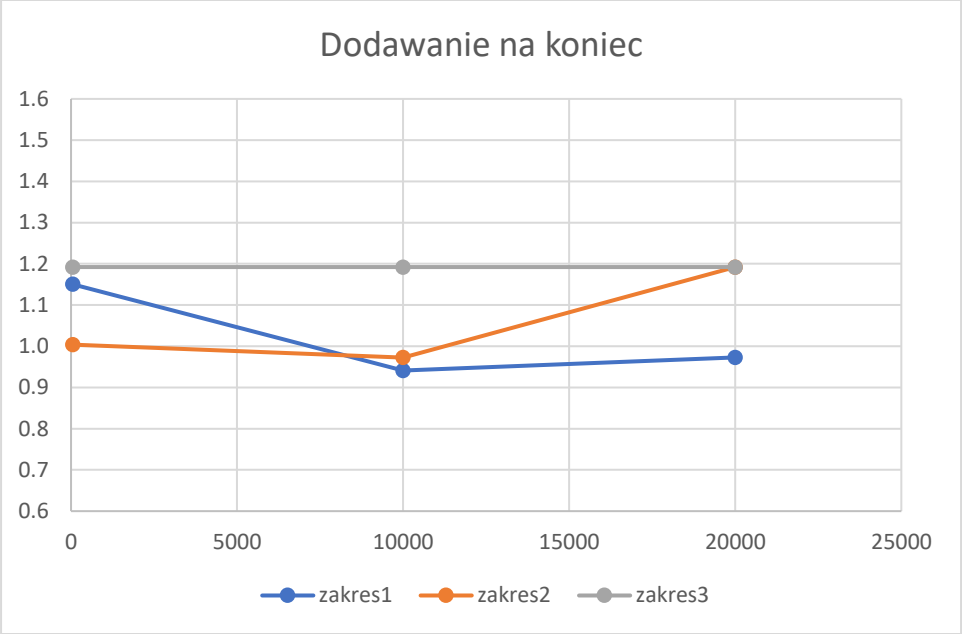
		Wyszukiwanie wartości	

	50	10000	20000
zakres1	258.875379	37866.061009	98181.992145
zakres2	204.911347	295.509016	796.250054
zakres3	327.332528	203.633004	692.971779

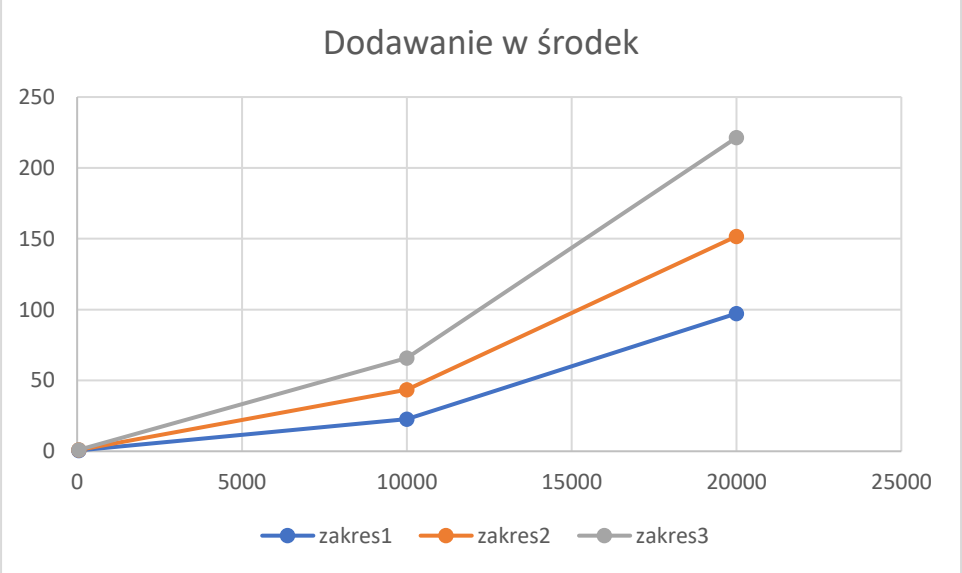
W algorytmie wyszukiwania problem nieliniowości polega na tym, że algorytm w trakcie działają za każdym razem, gdy znajdzie wyświetla informację, że odnalazł element i indeks na którym się on znajduje. Dla zakresu pierwszego, tych wystąpień jest bardzo dużo i wiele razy następuje wypisanie linijki, co oczywiście spowoduje znaczne spowolnienie algorytmu.

b. Lista

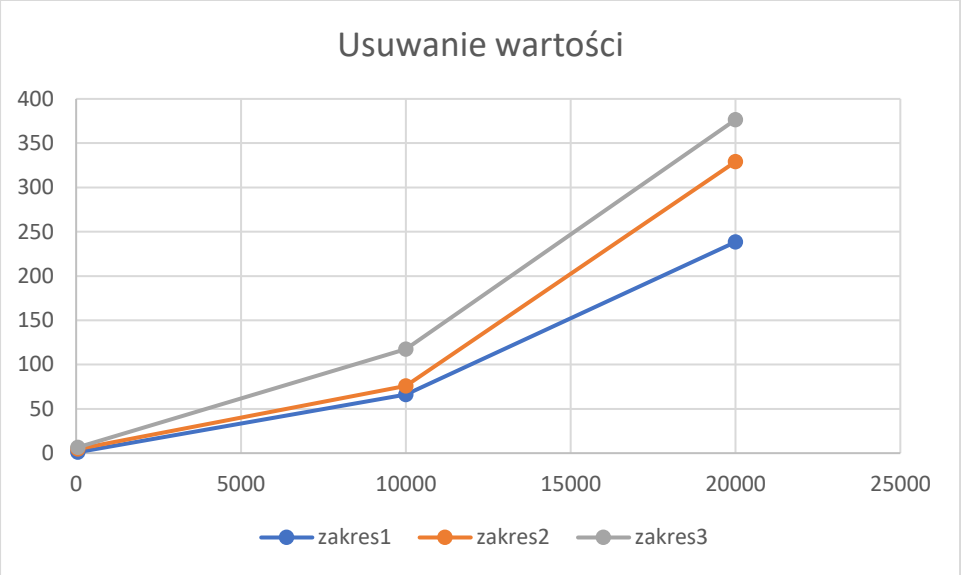




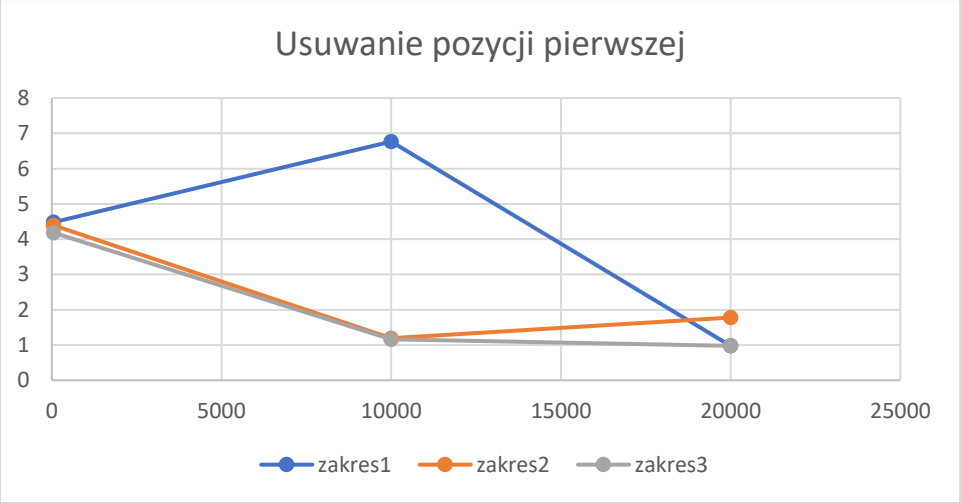
Wykresy dodania na początek i na koniec potwierdzają złożoność O(1).



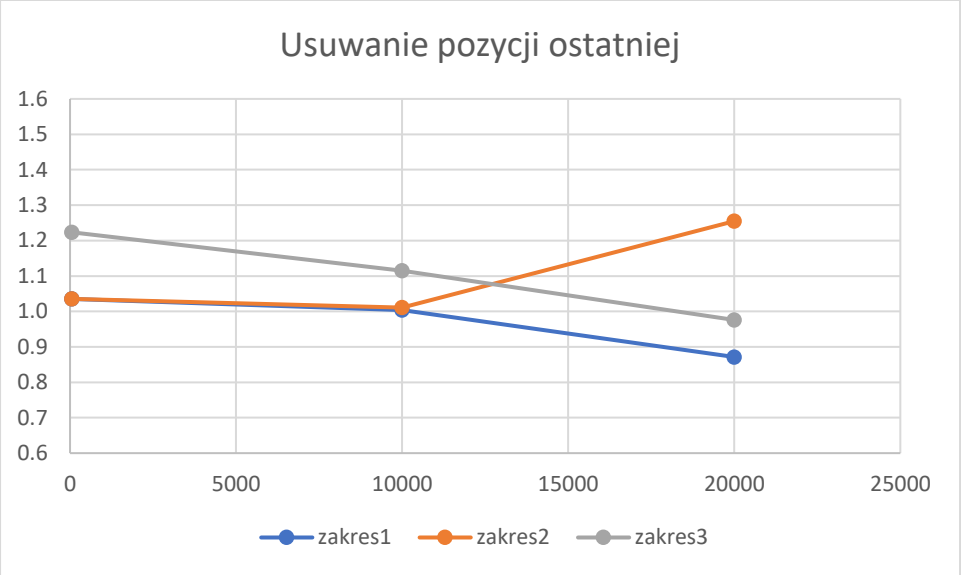
Wykres potwierdza złożoność O(n/2).



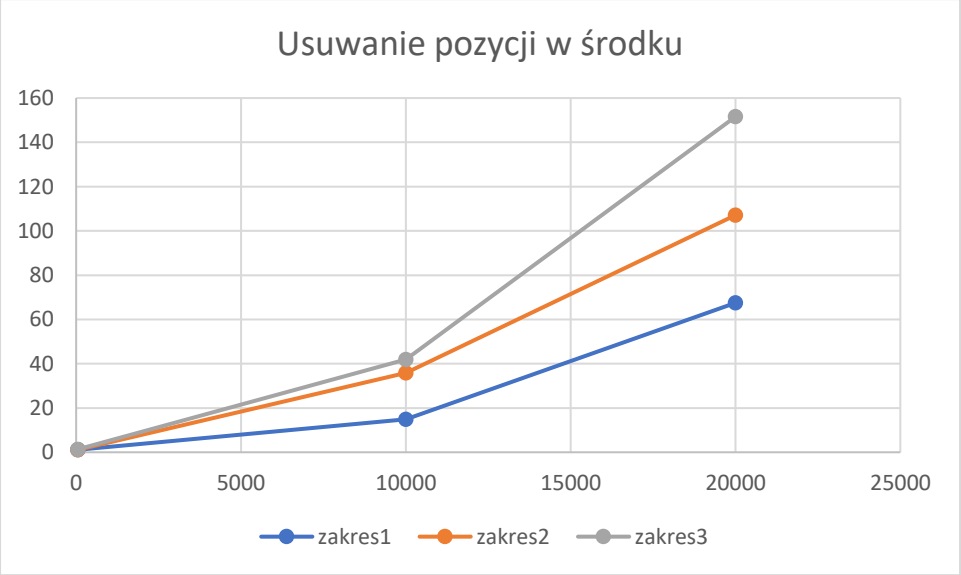
Wykres potwierdza złożoność O(n).



Wykres generalnie potwierdza złożoność O(1). Wahnięcie dla zakresu 1 było chwilowe i spowodowane większym obciążeniem komputera w chwili pomiarów.

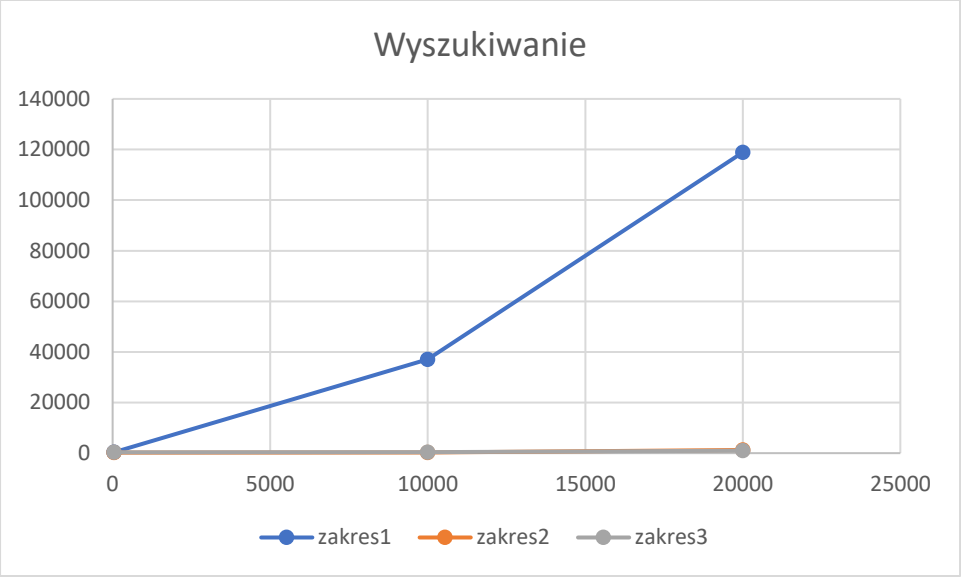


Wykres potwierdza złożoność O(1).



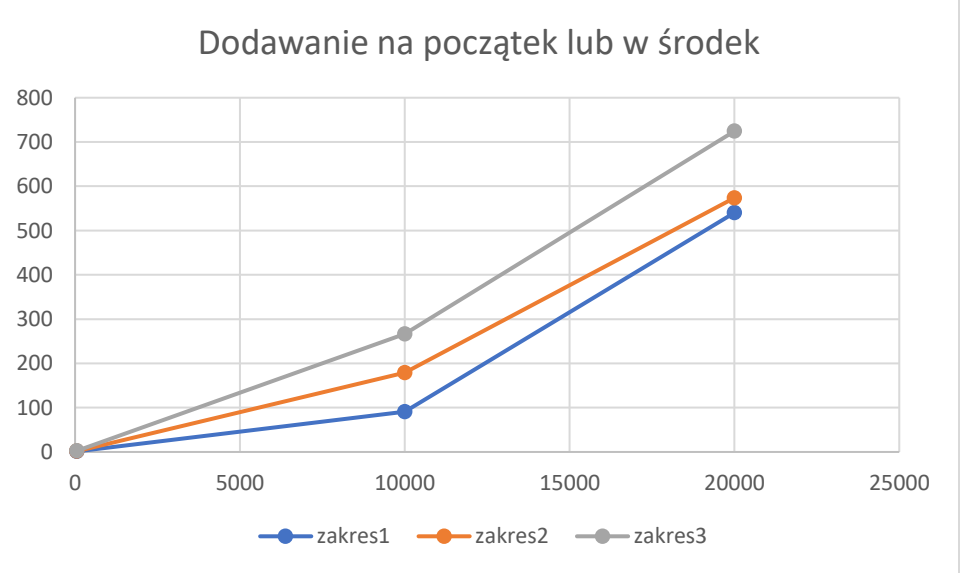
Wykres potwierdza złożoność O(n/2).

		Wyszukiwanie	
	50	10000	20000
zakres1	365.918100	37015.310141	118870.749836
zakres2	197.633418	302.828772	1241.922126
zakres3	325.729152	428.833147	966.207823

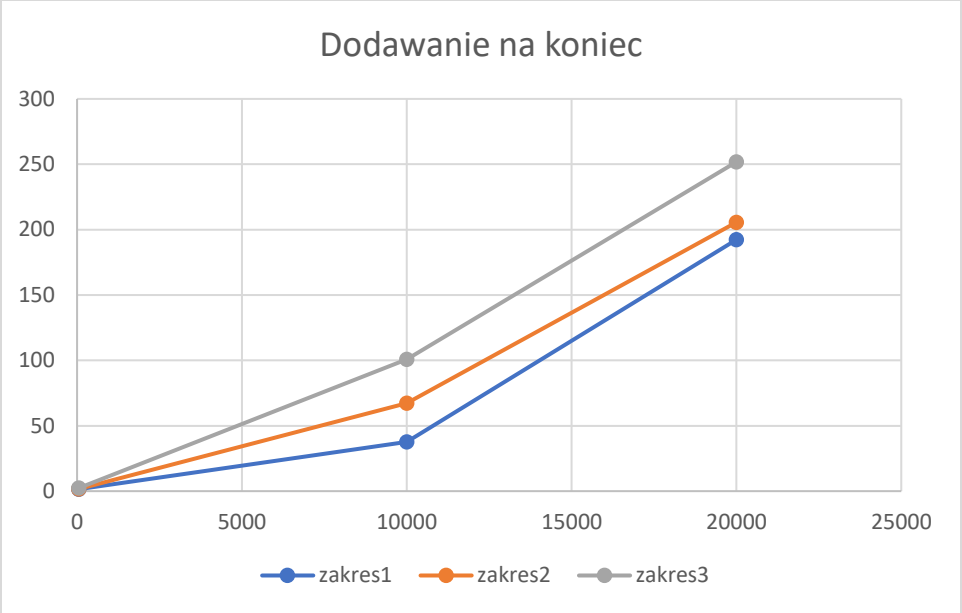


Występuje tu ten sam problem co w przypadku wyszukiwania w tablicy.

c. Kopiec



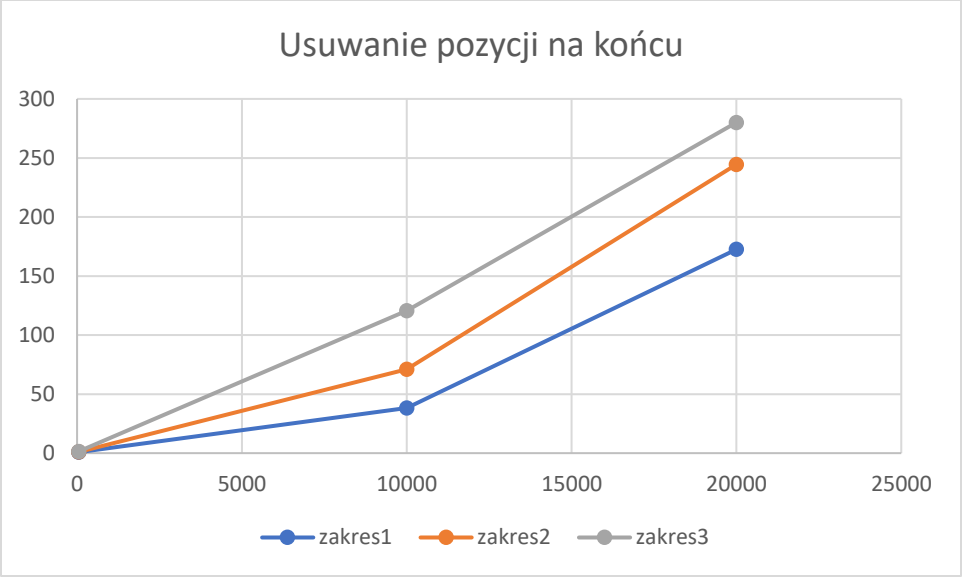
Wykres potwierdza złożoność O(n log(n)).



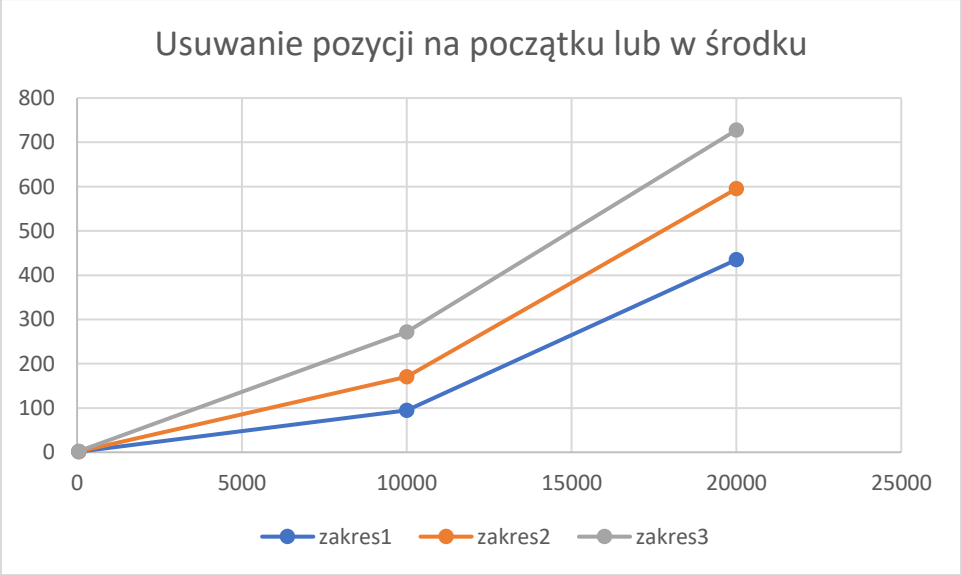
Wykres potwierdza O(n).

		Usuwanie wartości	
	50	10000	20000
zakres1	2.509631	9324.951167	118684.702546
zakres2	2.893047	365.386546	2491.357459
zakres3	3.293890	464.085616	2252.079857

Tutaj następuje ten sam problem, który był w poprzednich strukturach – usunięcie wartości to tak naprawdę usunięcie wielu pozycji dla zakresu1.

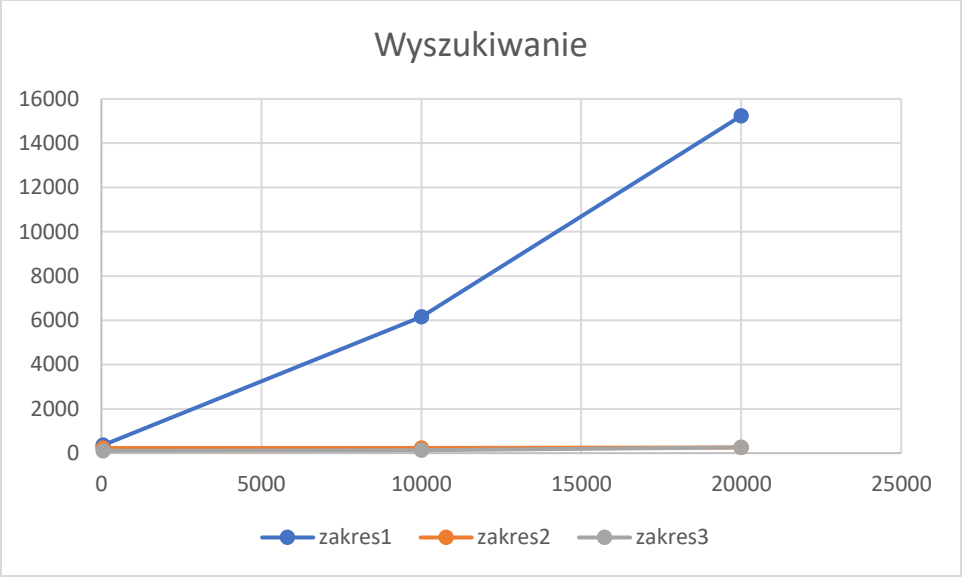


Wykres potwierdza złożoność O(n).



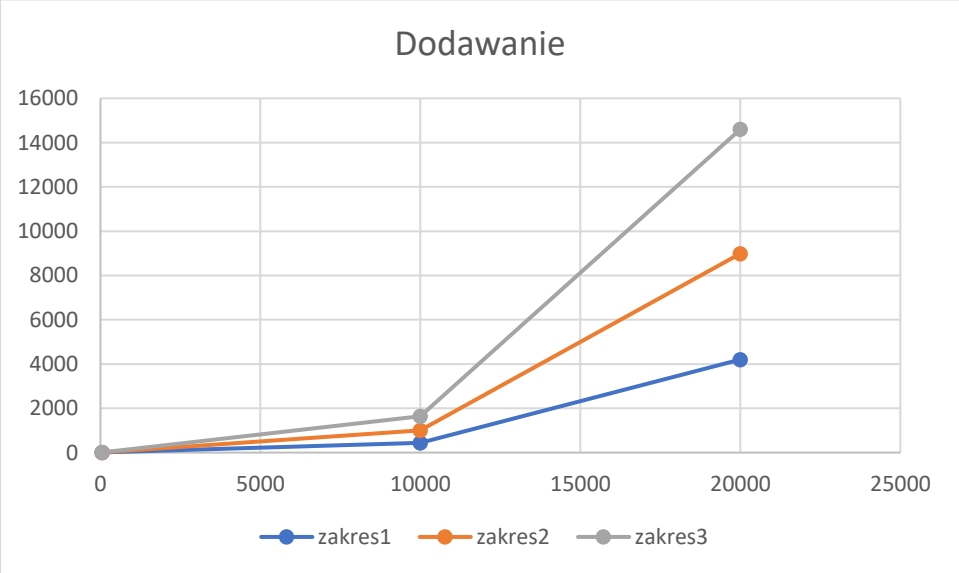
Wykres potwierdza złożoność O(n log(n)).

		Wyszukiwanie	
	50	10000	20000
zakres1	359.888015	6151.194494	15236.846398
zakres2	239.112037	232.799619	255.145789
zakres3	107.958935	131.975205	257.272004

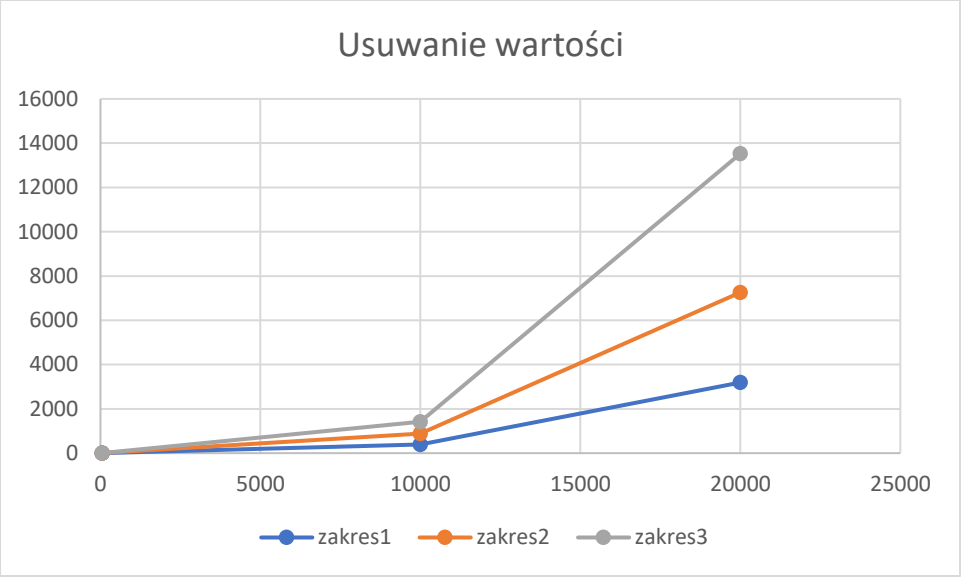


Następuje tu ten sam problem co w poprzednich strukturach i wywołanie funkcji wypisującej na konsole całkowicie zaburzyło wynik pomiarów.

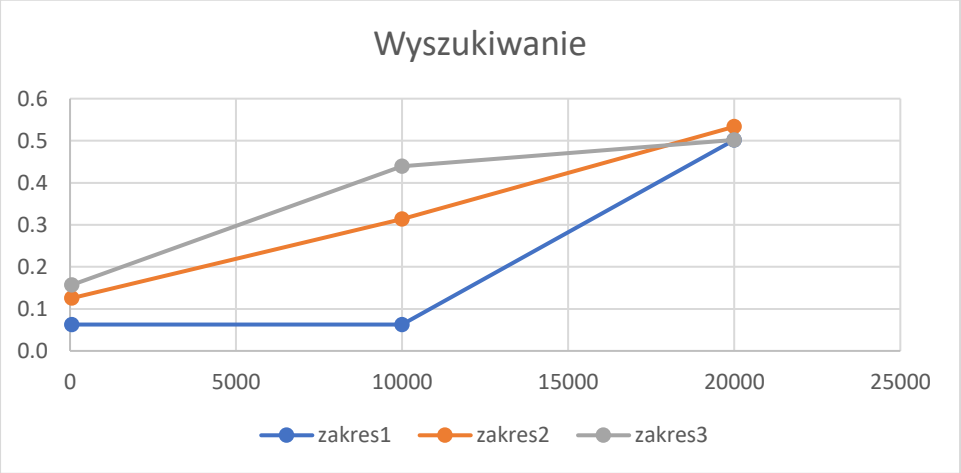
d. Drzewo BST



Nie jestem w stanie wyjaśnić, dlaczego złożoność w przypadku mojej implementacji drzewa BST nie spełnia złożoności O(n). Mam wrażenie, że powodem jest nieoptymalna implementacja algorytmu DSW.



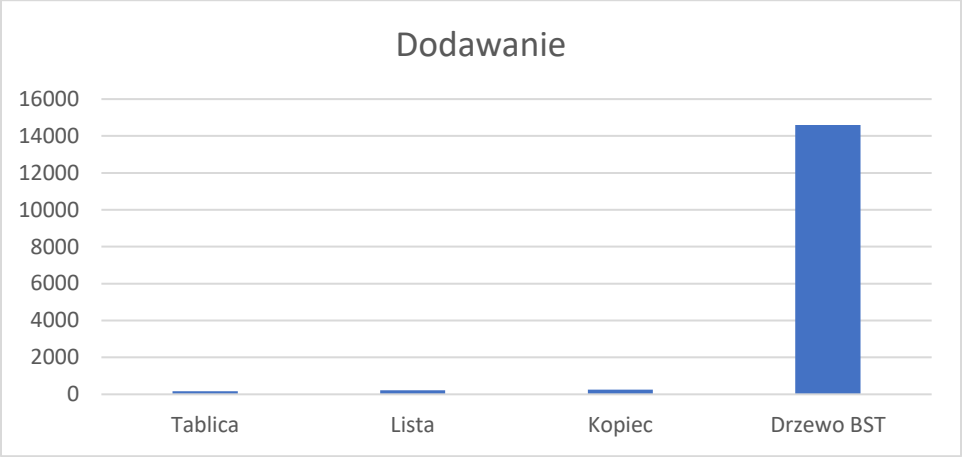
Jako, że tutaj też po usunięciu uruchamiany jest algorytm DSW, w tym przypadku jest taka sama sytuacja jak na górze.



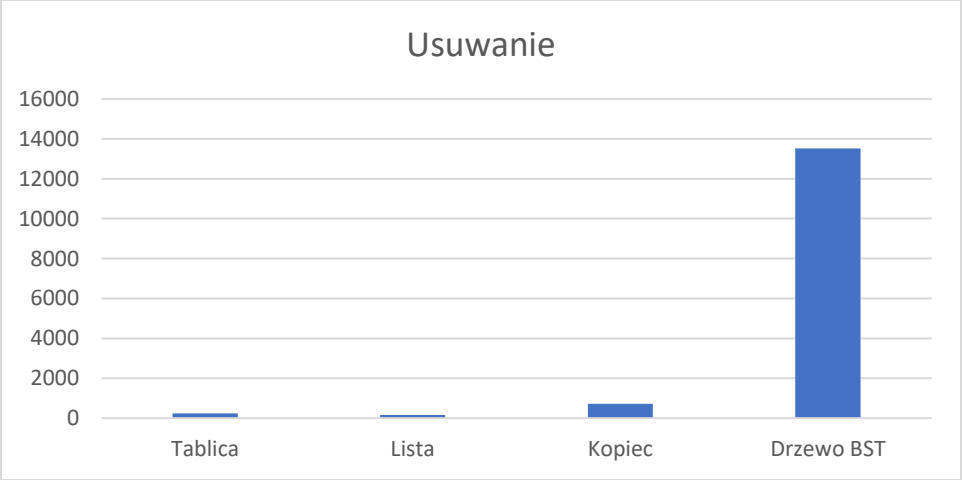
Wyszukiwanie powinno mieć złożoność O(log n). Wykres jednak tego nie oddaje z powodu niedokładności pomiarów licznika. Wyszukiwanie odbywało się tak szybko, że w wielu przypadkach licznik zwracał wartość 0.

5. Porównanie struktur

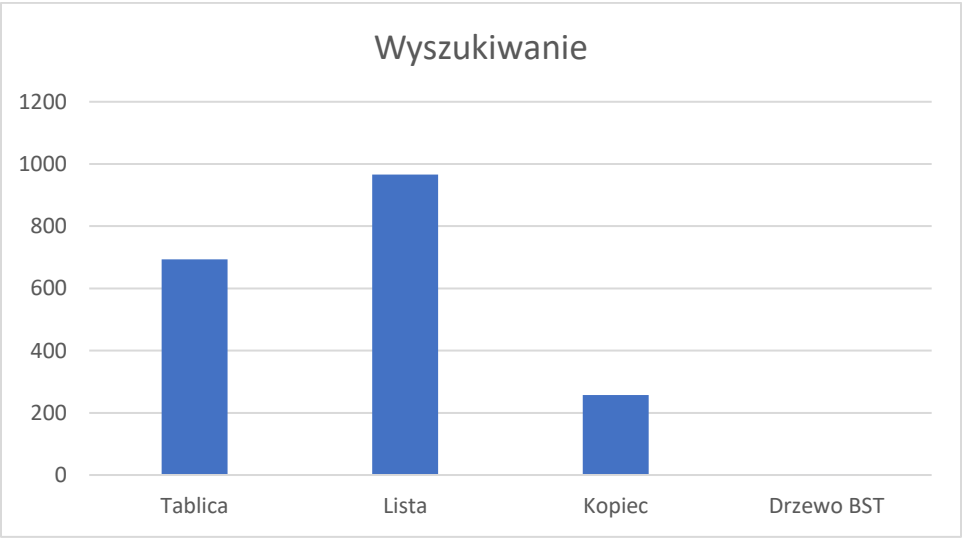
a. Dodawanie



b. Usuwanie



c. Wyszukiwanie



6. Wnioski

W trakcie pisania sprawozdania zauważyłem kilka znaczących błędów w mojej implementacji struktur, jak i w trakcie testowania.

Mam wrażenie, że mam źle zaimplementowany kopiec, co spowodowało lekko zakłamane wyniki.

Dodatkowo bardzo wielkim problemem okazał się algorytm wyszukiwania, który niepotrzebnie wypisywał wartości indeksów, gdzie znajduję się szukany element.

Najwygodniejszą i najbardziej uniwersalną strukturą wg. mnie jest lista. Patrząc na wyniki, nie jest też tak bardzo w tyle za innymi.

Najprostszą nadal jednak uważam tablicę i do małych ilości elementów jest wystarczająca.

Drewa BST to bardzo przydatna struktura do przechowywania danych, które często będą przeszukiwane.

Dla kopca nie znalazłem żadnego zastosowania podczas pracy nad tym projektem. Nie widzę sytuacji (innej niż sortowanie), gdy potrzeba szybkiego dostępu do największego elementu w strukturze.