Nikita Stepanenko
Nr.albumu 245816

Sprawozdanie
Struktury Danych i Złożoność Obliczeniowa

Zadanie projektowe 1:

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

Spis treści

1. Opis zadania projektowego

Celem zadania projektowego było zaimplementowanie różnego typu struktur danych, oraz funkcji wykonujących na nich operacje dodawania elementów, usuwania elementów, oraz wyszukiwania elementu zawierającego podaną wartość. Zaimplementowane zostały:

- Lista
- Tablica
- Kopiec
- Drzewo czerwono-czarne

Przy implementacji podanych struktur wykorzystane zostały zasady podane w zadaniu projektowym:

- Podstawową liczbą używaną przy tworzeniu struktur jest 32 bitowy integer
- Wszystkie struktury danych alokowane są dynamicznie, oraz realokowane po dodaniu/usunięciu elementu
- Dla tablicy oraz listy rozpatrzone osobno zostały przypadki dotyczące dodania lub usunięcia elementu zależnie od jego pozycji w danej strukturze
- Program napisany został w języku C++, z użyciem **vector** wyłącznie dla uproszczenia transakcji ze źródeł(konsol, plik, oraz rand()) do zaimplementowanych przez mnie struktur danych.
- Dane testowe wygenerowane zostały losowo, w różnych ilościach, oraz w pełnym zakresie liczb na jakich operuje program, przy użyciu funkcji rand() zawartej w standardowej bibliotece języka C++. Losowość oparta została o czas procesora.
- Wyniki zostały uśrednione po wykonaniu 10 pomiarów dla każdej z przyjętych grup testowych

2. Złożoności obliczeniowe operacji

2.1. Opis ogólny złożoności obliczeniowej

Złożoność obliczeniową definiujemy jako ilość zasobów komputerowych koniecznych do wykonania programu realizującego algorytm. Wyrażana jest jako funkcja parametru, od którego zależna jest jej wartość. Bazując na tej definicji możemy więc wyróżnić dwa podstawowe typy złożoności obliczeniowej:

1) Złożoność pamięciowa, czyli ilość pamięci, jaka zostanie wykorzystana w celu realizacji danego algorytmu, bądź przechowana określonej liczby danych.

2) Złożoność czasowa, czyli czas potrzebny na wykonanie danego algorytmu, wyrażany zazwyczaj w podstawowych jednostkach czasu, lub liczbie cykli procesora. Podobnie jak w przypadku złożoności pamięciowej, jest ona ściśle zależna od ilości danych jakimi operujemy.

Dla wszystkich algorytmów, w których złożoności są zależne od zbioru danych mamy

Złożoność średnia, czyli typowe zużycie zasobów dla losowego zbioru danych. Ustalana jest jako średnia uzyskiwana z dokonanych pomiarów

2.2. Złożoności operacji wykonywanych na tablicy

Funkcja	Średnia
Dodanie wartości	O(n)
Usunięcie wartości	O(n)
Wyszukanie wartości	O(n)
Dostęp do elementu	O(1)

2.3. Złożoność operacji wykonywanych na liście

Funkcja	Średnia
Dodanie wartości	O(-)
Usunięcie wartości	O(-)
Wyszukanie wartości	O(n)
Dostęp do elementu	O(n)

2.4. Złożoność operacji wykonywanych na kopcu

Funkcja	Średnia
Dodanie wartości	O(1)
Usunięcie wartości	O(1)
Wyszukanie wartości	O(n)
Dostęp do elementu	O(n)

3. Pomiar czasu

Wykorzystana została biblioteka „Chrono.h” do pomiaru czasu, zawarta w zbiorze standardowych bibliotek języka C++, w standardzie C++11. Biblioteka ta operuje na czasie procesora, co pozwala na uzyskanie dokładności czasowej równej jednemu okresowi przebiegu procesora. Na potrzeby uproszczenia pomiaru, przyjęta została jednak dokładność rzędu jednej nanosekundy. Cała klasa pomiarowa zawarta w programie, została umieszczona w pliku „TimeCheck.h” oraz „TimeCzeck.cpp”, a jej implementacja prezentuje ma taką postać:

TimeCheck.h

```
#pragma once
#include <iostream>
#include <chrono>
#include <fstream>

class TimeCheck
{
public:
	std::chrono::high_resolution_clock::time_point timeStart;
	std::chrono::high_resolution_clock::time_point timeEnd;

	void time_Start();
	void time_End();
	int getTime();
	void printResult();
};
```

TimeCheck.cpp

```
#include "pch.h"
#include "TimeCheck.h"

void TimeCheck::time_Start()
{
	timeStart = std::chrono::high_resolution_clock::now();
}

void TimeCheck::time_End()
{
	timeEnd = std::chrono::high_resolution_clock::now();
}

int TimeCheck::getTime()
{
	return std::chrono::duration_cast<std::chrono::nanoseconds>(timeEnd - timeStart).count;
}

void TimeCheck::printResult()
{
	std::ofstream fout;
	fout.open("TIME.txt", std::ofstream::app);
	fout << getTime() << std::endl;
	fout.close();
}
```

Funkcja zwraca różnicę w czasie przed i po wykonaniu funkcji. Program pobiera czas przed wykonaniem funkcji, po jej wykonaniu, a następnie oblicza różnicę czasu bazując na ilości okresów przebiegu procesora, które miały miejsce pomiędzy oboma zarejestrowanymi czasami. Funkcja printResult() wpisuje wynik do pliku dla uproszczenia podwumowania.

5. Implementacja tablicy

Zaimplementowałem tablicę na 2 sposoby: MyArrayList i MyArrayListSlow

Tablica – kontener uporządkowanych danych takiego samego typu, w którym poszczególne elementy dostępne są za pomocą kluczy (indeksu). Indeks najczęściej przyjmuje wartości numeryczne. Rozmiar tablicy jest albo ustalony z góry (tablice statyczne), albo może się zmieniać w trakcie wykonywania programu (tablice dynamiczne). W naszym przypadku mamy tablice dynamiczne.

MyArrayListSlow – ta wersja zajmuje tylie pamięci, ile elementów. Rozszerza się i zmniejsza się o 1 co powoduje konieczność kopiowania z starej tabeli do nowej co tranzakcję.

Ważna jest relokacja pamięci – swoiste zbudowanie tablicy od nowa po wykonaniu operacji aby ‘odświeżyć’ zawartość tablicy.

- Dodawanie na początku

Najpierw wymagane jest zwiększenie rozmiaru tablicy o jeden, a następnie przesunięcie wszystkich elementów o adres wyżej, tak żeby pierwsze miejsce (początek) było puste. W to miejsce wstawiamy zadaną wartość.

- Dodawanie na końcu

Podobnie jak w dodawaniu na początek, rozmiar zostaje zwiększony o jeden, ale przesuwanie elementów nie potrzebne, bo nowo utworzony indeks jest „pusty”– tam wstawiamy zadaną wartość.

- Dodawanie w wybranym miejscu

Zwiększenie rozmiaru tablicy o jeden, a następnie szukamy podanego indeksu – dalej robimy operację podobną do dodawania na początek, ale zaczynając od wyznaczonego indeksu.

- Usuwanie z początku tablicy

Operacja ta wpierw usuwa wartość z pierwszej komórki tablicy, a następnie przesuwa wszystkie pozostałe elementy wstecz. Wtedy ostatnia komórka jest pusta i zostaje usunięta, zmniejszając rozmiar o jeden.

- Usuwanie z końca tablicy

Usuwa ostatni element z tablicy, zmniejszając rozmiar tablicy o jeden.

- Usuwanie z wybranego miejsca

Szukamy zadanego indeksu, a następnie usuwamy z niego wartość. Potem wszystkie elementy powyżej są przesuwane do tyłu, a ostatnia komórka jest usuwana zmniejszając rozmiar tablicy.

- Wyszukiwanie

Przechodzi po wszystkich indeksah od początku i sprawdza ich wartość z podaną. Jeśli element został znaleziony – daje komunikat i zwraca indeks elementu. Jeśli element nie istnieje lub jest na końcu złożoność czasowa jest równa O(n) gdzie n to liczba elementów tablicy. W innym przypadku jest zależna od pozycji szukanego klucza(indeksu).

MyArrayList – ta wersja zajmuje pamięci najczęściej więcej niż ilość elementów. Rozszerza się w 2 razy, kiedy ilość elementów jest równa rozmiarowi, co powoduje konieczność kopiowania z starej tabeli do nowej co raz rzadziej z wzrostem ilości elementów.

Ważna jest relokacja pamięci – swoiste zbudowanie tablicy od nowa po wykonaniu operacji aby ‘odświeżyć’ zawartość tablicy.

- Dodawanie na początku

Najpierw wymagane jest zwiększenie rozmiaru tablicy w 2 razy, jeżeli rozmiar jest równy ilości elementów, a następnie przesunięcie wszystkich elementów o adres wyżej, tak żeby pierwsze miejsce (początek) było puste. W to miejsce wstawiamy zadaną wartość.

- Dodawanie na końcu

Podobnie jak w dodawaniu na początek, rozmiar zostaje zwiększony w 2 razy, jeżeli rozmiar jest równy ilości elementów, ale przesuwanie elementów nie potrzebne, bo nowo utworzony indeks jest „pusty”– tam wstawiamy zadaną wartość.

- Dodawanie w wybranym miejscu

Zwiększenie rozmiaru tablicy w 2 razy, jeżeli rozmiar jest równy ilości elementów, a następnie szukamy podanego indeksu – dalej robimy operację podobną do dodawania na początek, ale zaczynając od wyznaczonego indeksu.

- Usuwanie z początku tablicy

Operacja ta wpierw usuwa wartość z pierwszej komórki tablicy, a następnie przesuwa wszystkie pozostałe elementy wstecz. Usunięcia komórki nie ma.

- Usuwanie z końca tablicy

Usuwa ostatni element z tablicy, zmniejszając rozmiaru nie ma.

- Usuwanie z wybranego miejsca

Szukamy zadanego indeksu, a następnie usuwamy z niego wartość. Potem wszystkie elementy powyżej są przesuwane do tyłu, rozmiar tablicy nie zmniejsza się.

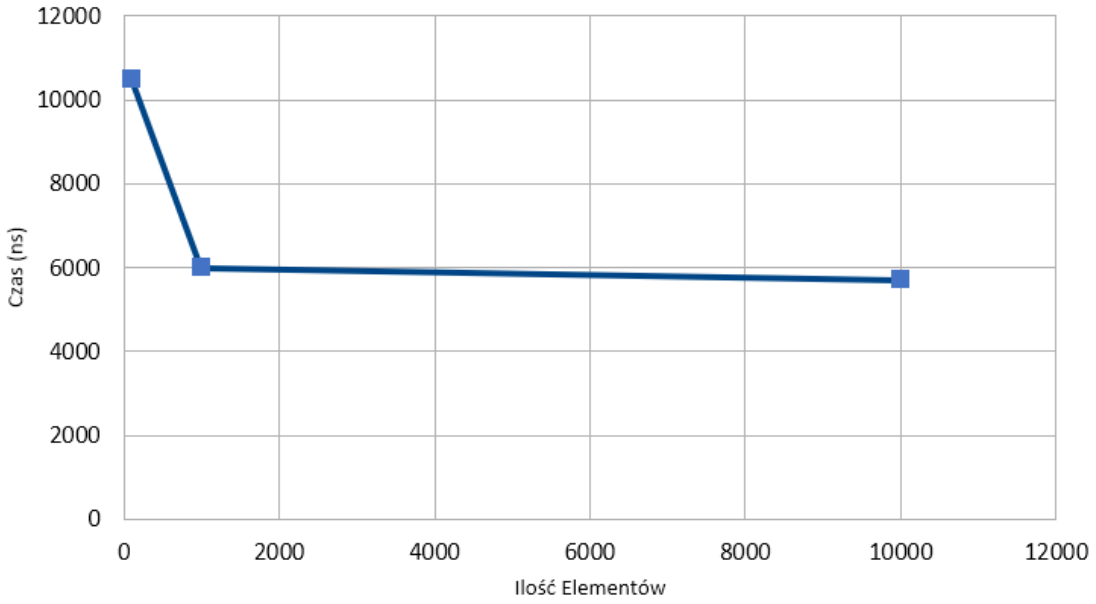
- Wyszukiwanie

Przechodzi po wszystkich indeksah od początku i sprawdza ich wartość z podaną. Jeśli element został znaleziony – daje komunikat i zwraca indeks elementu. Jeśli element nie istnieje lub jest na końcu złożoność czasowa jest równa O(n) gdzie n to liczba elementów tablicy. W innym przypadku jest zależna od pozycji szukanego klucza(indeksu).

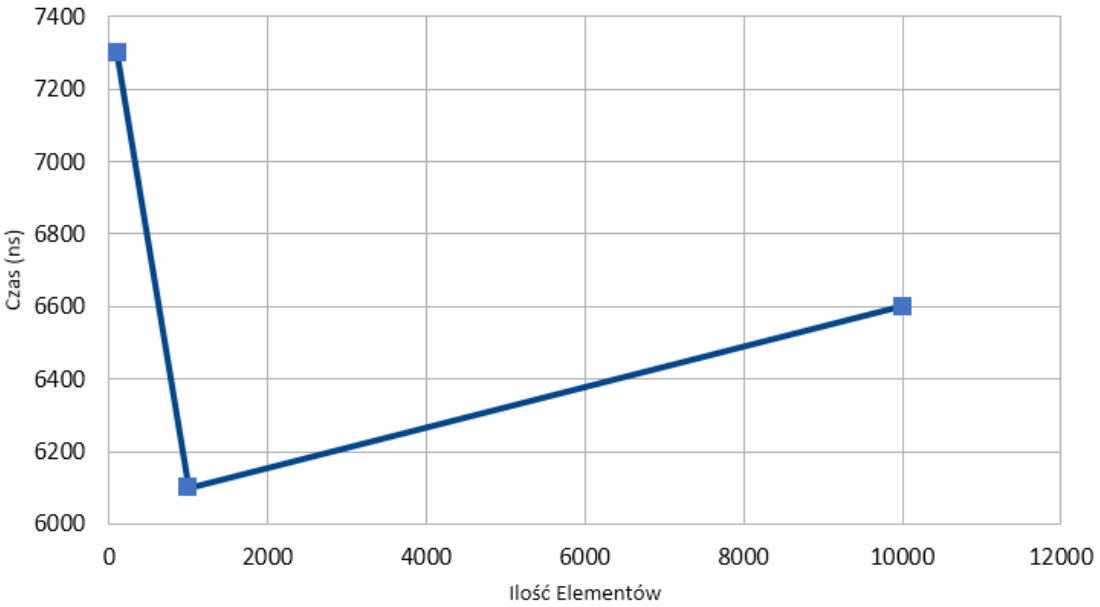
5. Pomiary funkcji tablicy

- Dodawanie na początku

  ArrayList

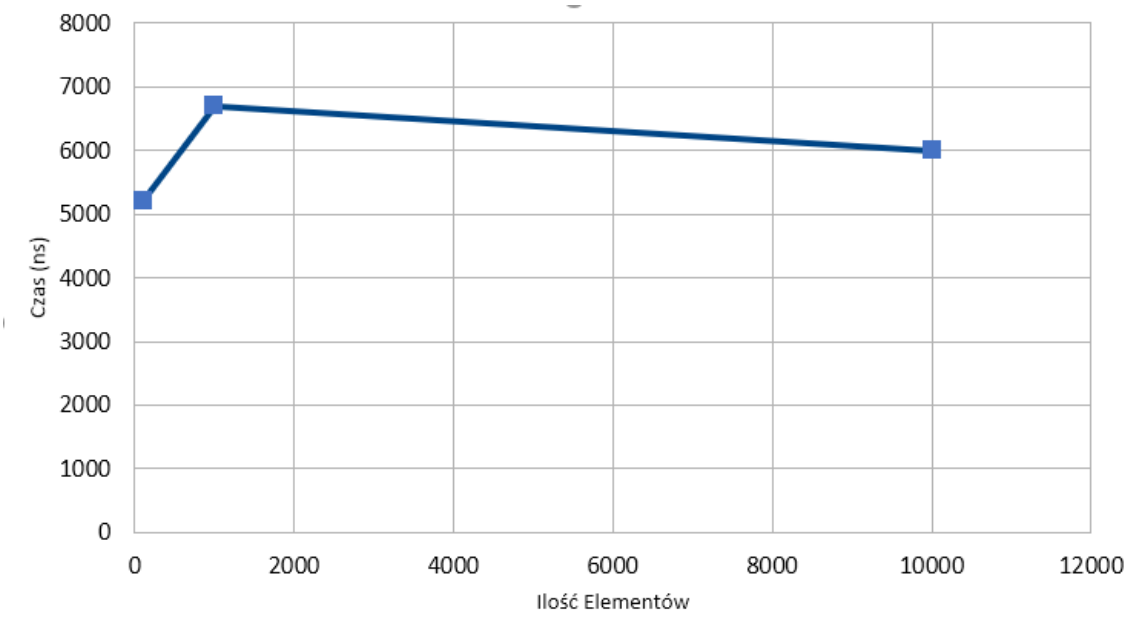


  ArrayListSlow

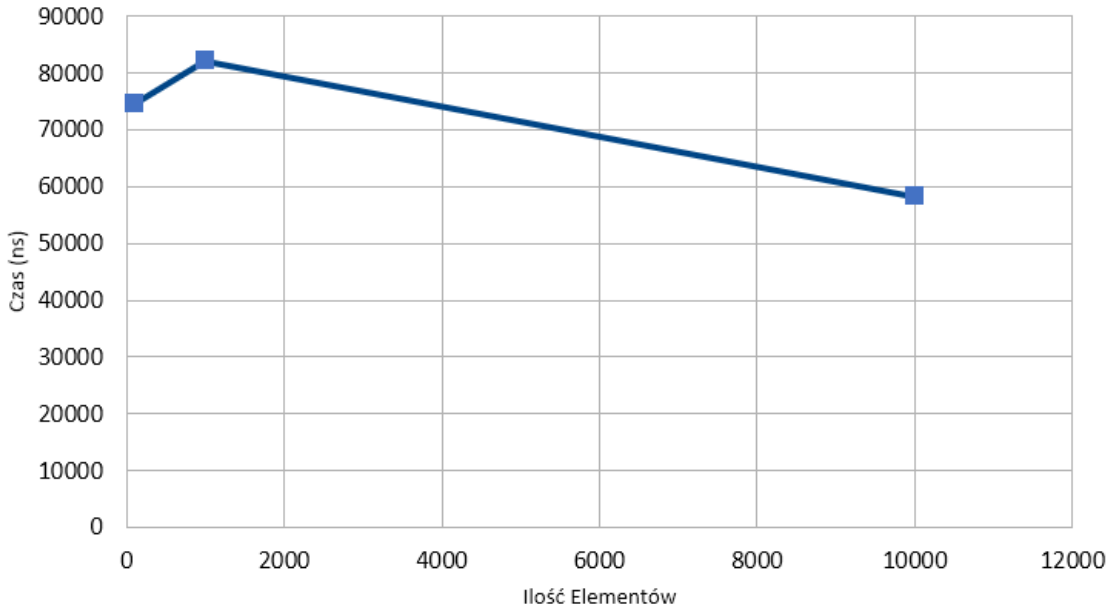


- Dodawanie na końcu

  ArrayList

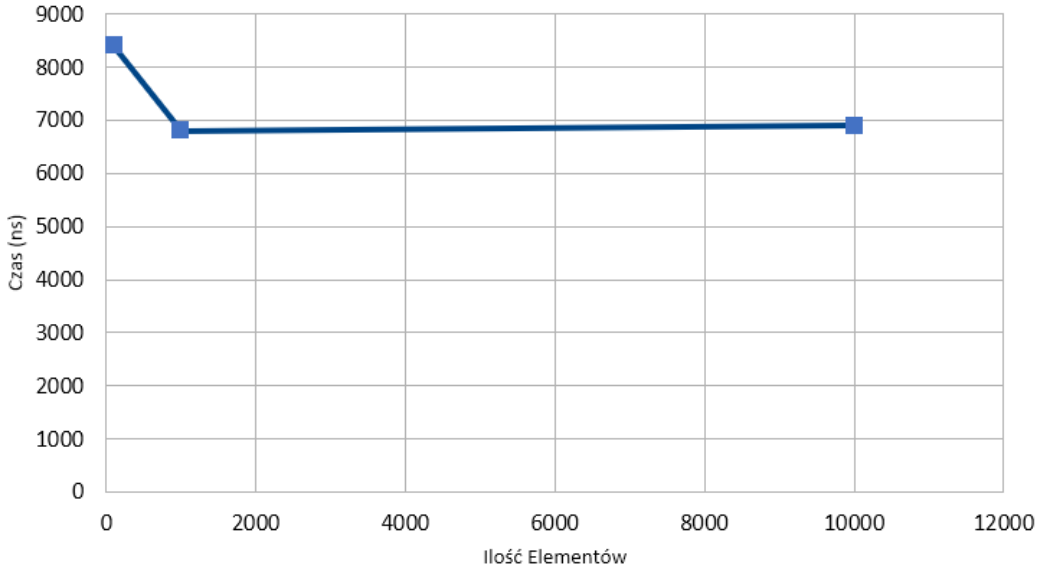


ArrayListSlow

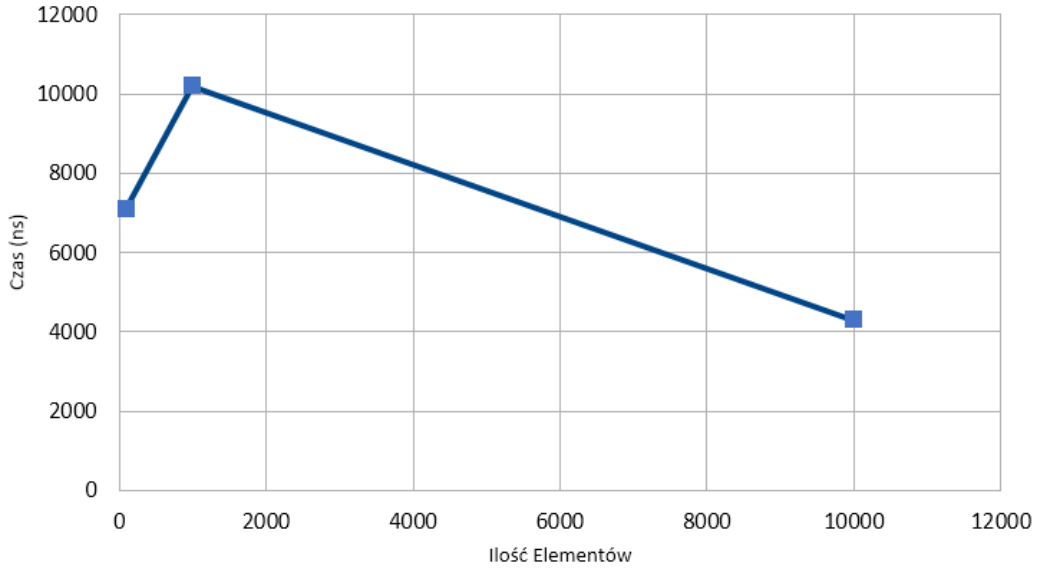


- Dodawanie na dowolnej pozycji

ArrayList

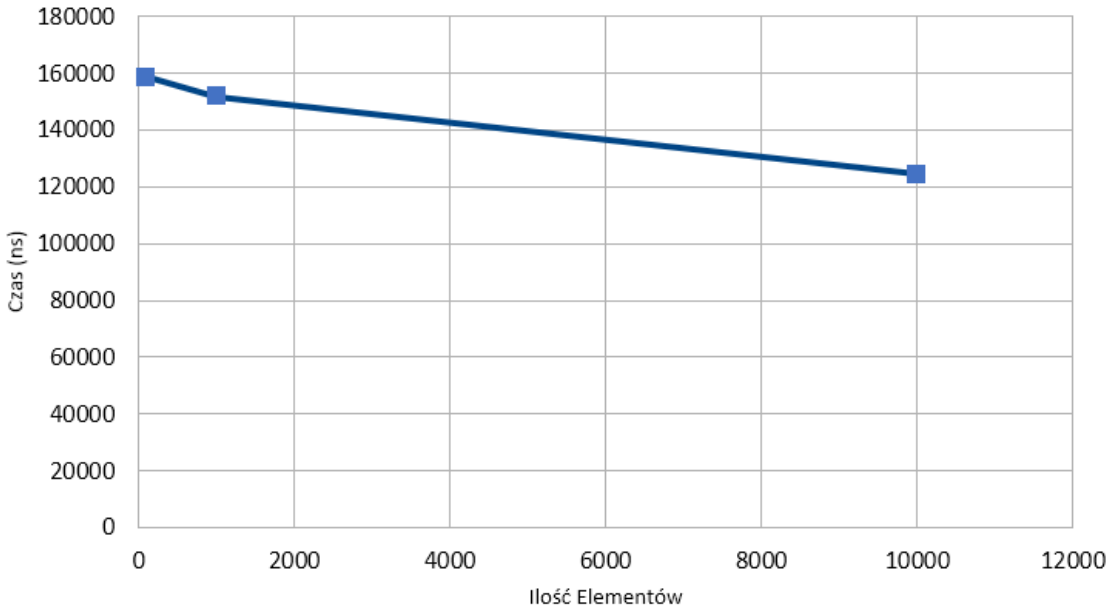


ArrayListSlow

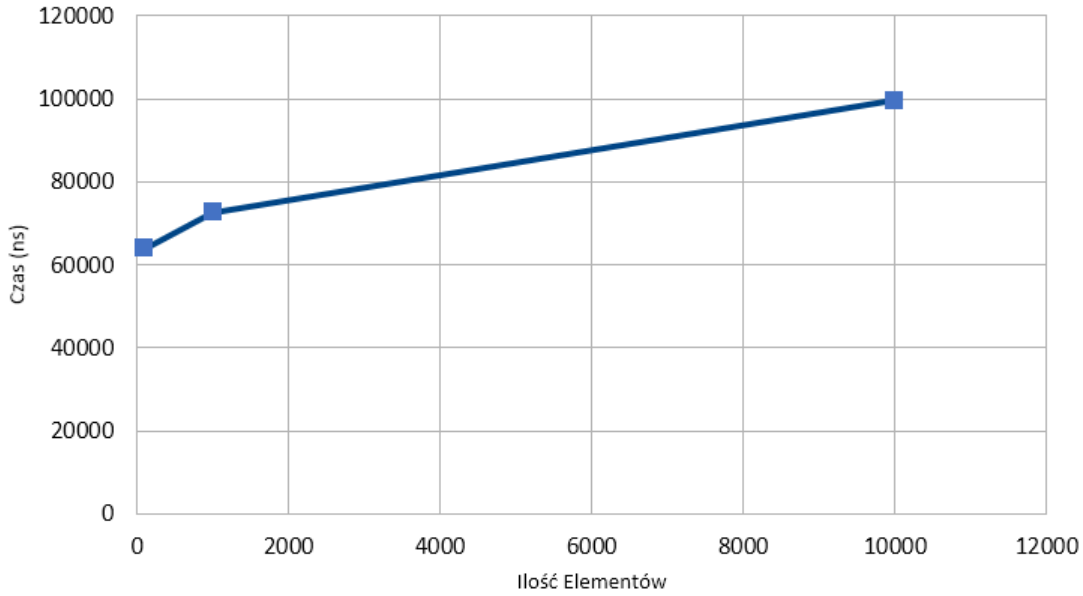


- Usuwanie z początku

ArrayList

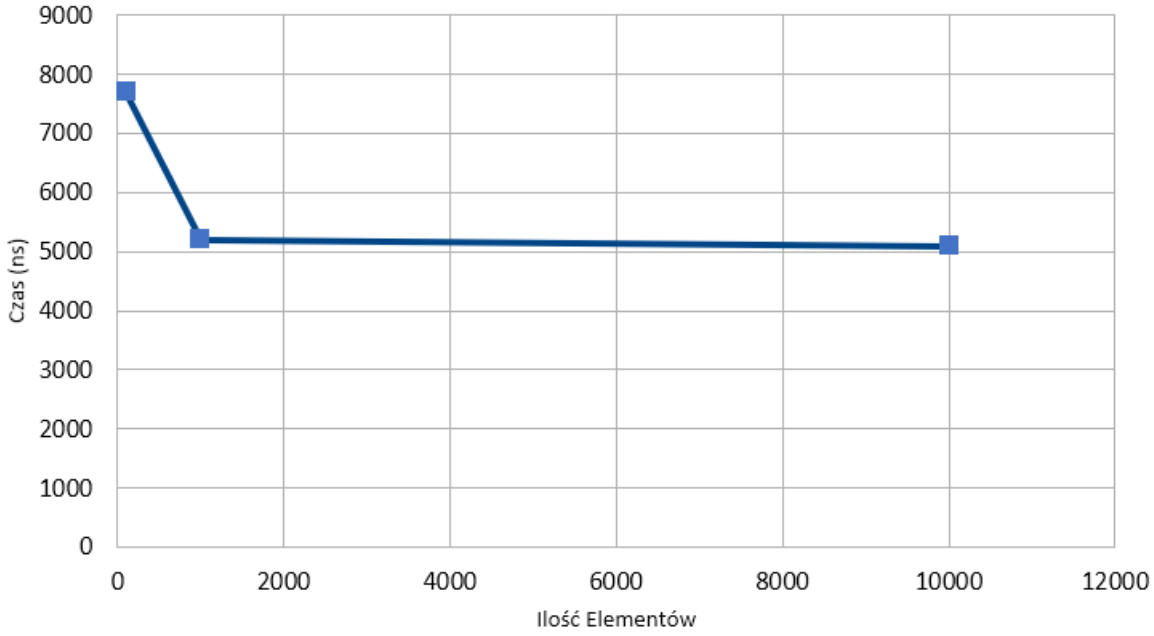


ArrayListSlow

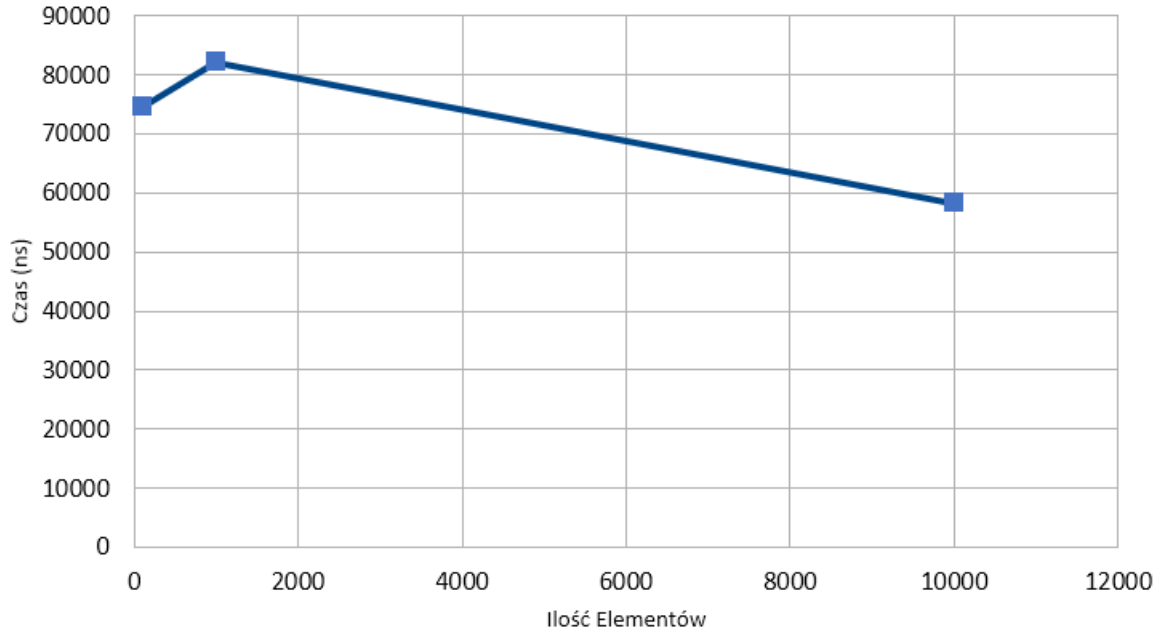


- Usuwanie z końca

ArrayList

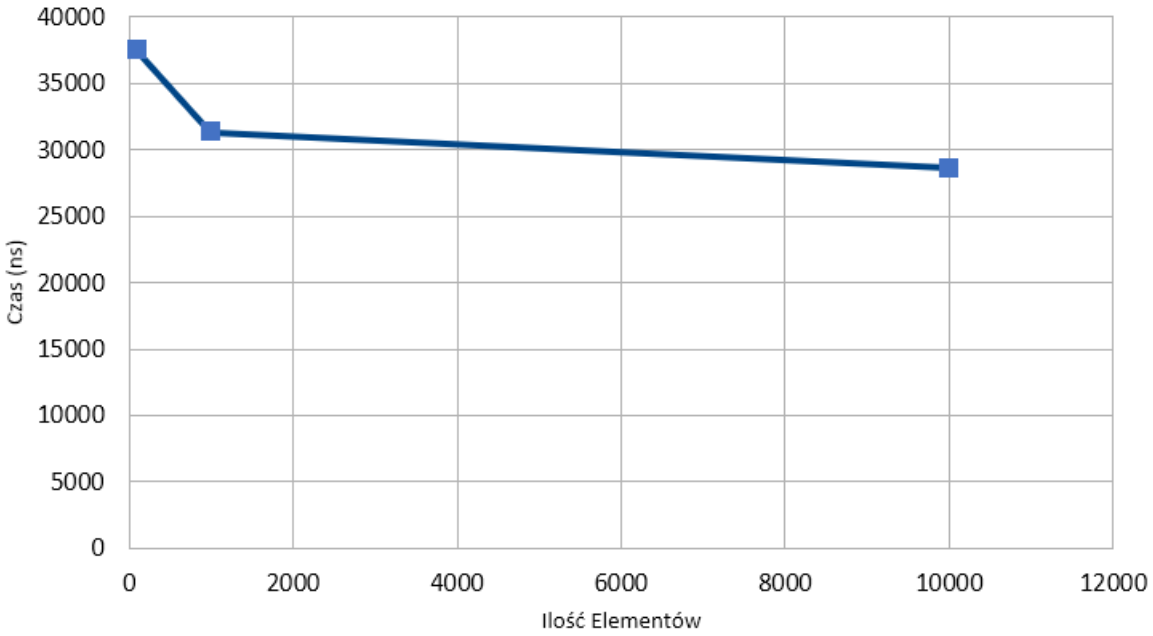


ArrayListSlow:

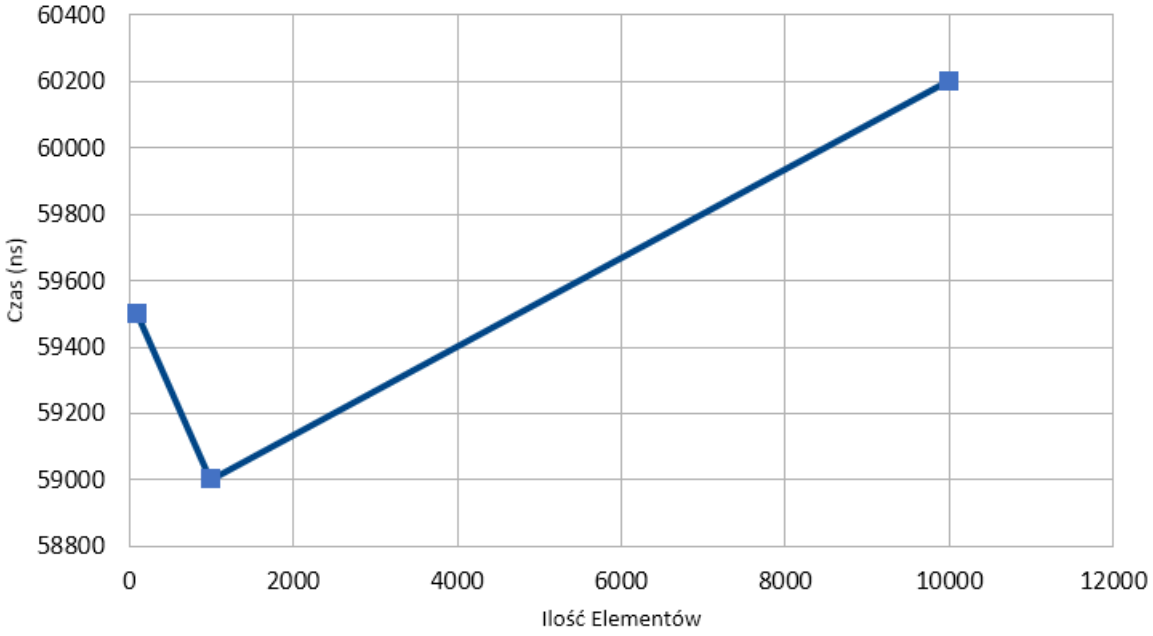


- Usuwanie z dowolnej pozycji

ArrayList

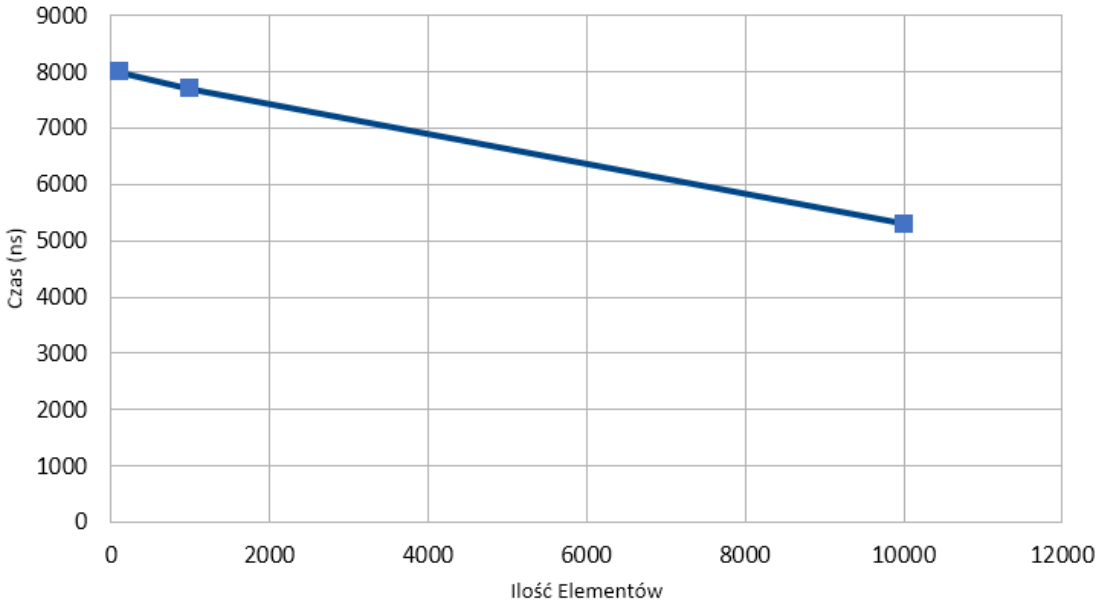


ArrayListSlow

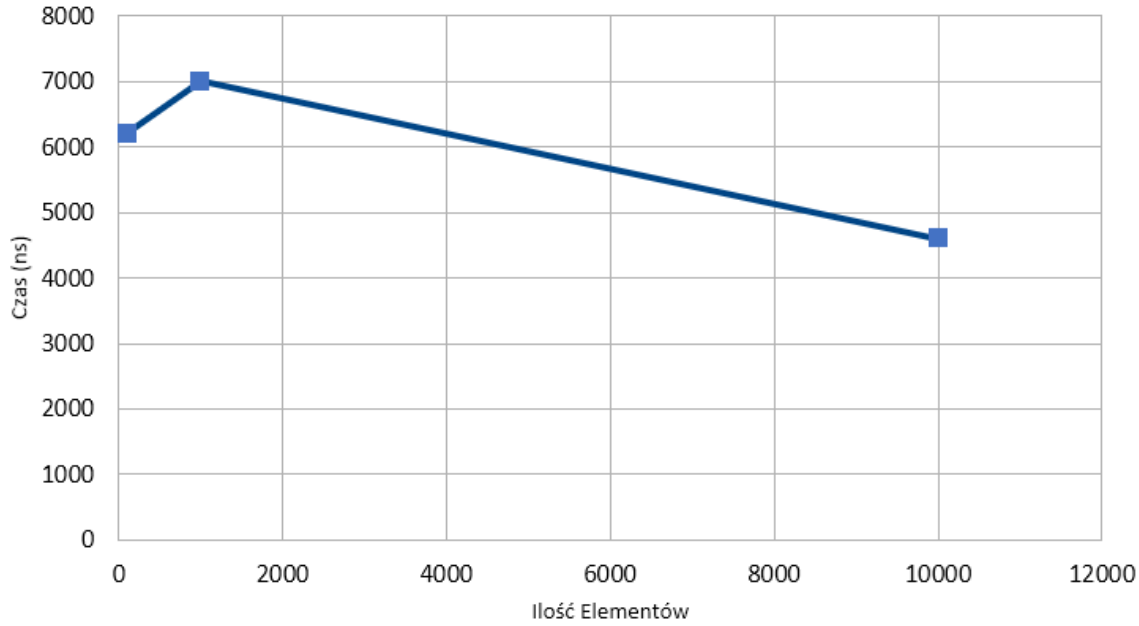


- Wyszukiwanie

ArrayList



ArrayListSlow



6. Implementacja listy

Lista – struktura danych służąca do reprezentacji zbiorów dynamicznych, w której elementy ułożone są w liniowym porządku. Rozróżniane są dwa podstawowe rodzaje list: lista jednokierunkowa w której z każdego elementu możliwe jest przejście do jego następnika oraz lista dwukierunkowa w której z każdego elementu możliwe jest przejście do jego poprzednika i następnika. W naszym przypadku mamy dwukierunkową. Lista jest podobna do tablicy

Każdy element tablicy posiada dwa wskaźniki – na element znajdujący się przed i za nim w liście. Pierwszy element (zwany głową) ma wskaźnik na poprzedni element warty NULL a więc wskazuje na 'nic' , tak samo jest w przypadku ostatniego elementu listy (zwanego ogonem) którego wskaźnik na następny element też jest równy NULL.

- Dodawanie na początek listy

Tworzony jest nowy element o zadanej wartości, do którego wskaźnika na element poprzedni wstawiamy NULL, a dla obecnej 'głowy' do wskaźnika na element poprzedni wstawiamy adres nowego elementu, by wreszcie przypisać nowemu elementowi status 'głowy'. Zwiększamy rozmiar listy(liczbę elementów w liście). Jeśli lista była pusta to nowy element jest jednocześnie głową i ogonem więc nadajemy mu też taki status.

- Dodawanie na koniec listy

Dodawanie elementu na koniec listy jest podobne do dodawana na początek. Tworzymy nowy element i umieszczamy w nim dane. We wskaźniku na element następny wstawiamy adres zerowy (NULL), w polu wskaźnika na poprzedni element wstawiamy adres obecnego ogona. Nowemu elementowi nadajemy status ogona i zwiększamy rozmiar listy o jeden (ilość elementów w liście) oraz do pola następnego elementu w poprzednim ogonie wstawiamy adres dodanego elementu.

- Dodawanie na wybrane miejsce

Jeśli wybrany element jest pierwszym elementem listy (lub podano zły adres) , to zadanie sprowadza się do dodania nowego elementu na początku listy. Tak samo jeżeli jest to ostatni adres listy – wykonywana jest operacja dodawania na ostatnie miejsce. W innym przypadku tworzymy nowy element i w adresie poprzednika wstawiamy adres elementu będącego na podanym adresie, a do adresu następnika – adres elementu będącego następcą elementu o zadanym adresie. Potem zmieniamy adresy następnika (dla elementu o wybranym adresie) i poprzednika (dla elementu

będącym następnikiem nowego elementu) aby wskazywały na nowy element i zwiększamy rozmiar(liczbę elementów).

- Usuwanie z początku listy

Zamieniamy status ‘głowy’ z elementu pierwszego na element następny. Do adresu poprzednika nowej głowy wstawiamy adres NULL. Usuwamy niepotrzebny element z pamięci i zmniejsamy liczbę elementów.

- Usuwanie z końca listy

Podobnie jak w przypadku początku listy – elementowi poprzedzającemu ogon nadajemy status nowego ‘ogona’ a do jego adresu następcy wstawiamy NULL. Usuwamy niepotrzebny element i zmniejszamy licznik wskazujący liczbę elementów.

- Usuwanie z wybranego miejsca

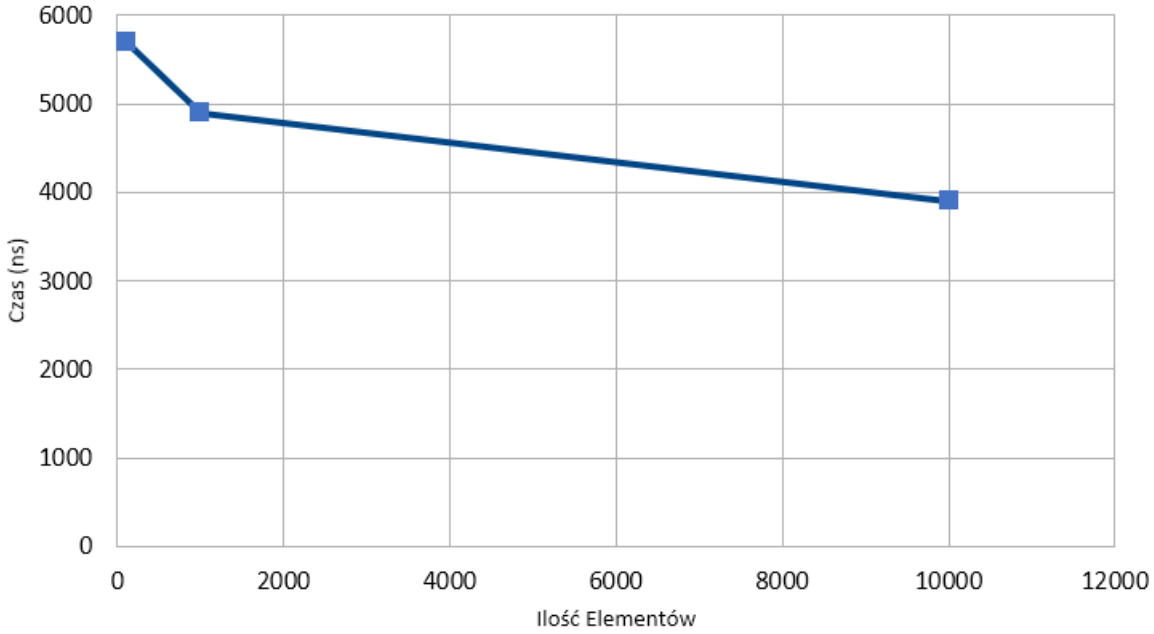
Jeśli wybrane miejsce to ogon lub głowa – wywołujemy odpowiednie operacje. W przeciwnym wypadku do adresu następnika elementu o wybranym elemencie wstawiamy adres poprzednika wybranego elementu. Do adresu poprzednika elementu występującego po wybranym wstawiamy taki sam adres jaki posiadał usuwany element. Potem usuwamy z pamięci niepotrzebny element i zmniejszamy licznik elementów.

- Wyszukiwanie

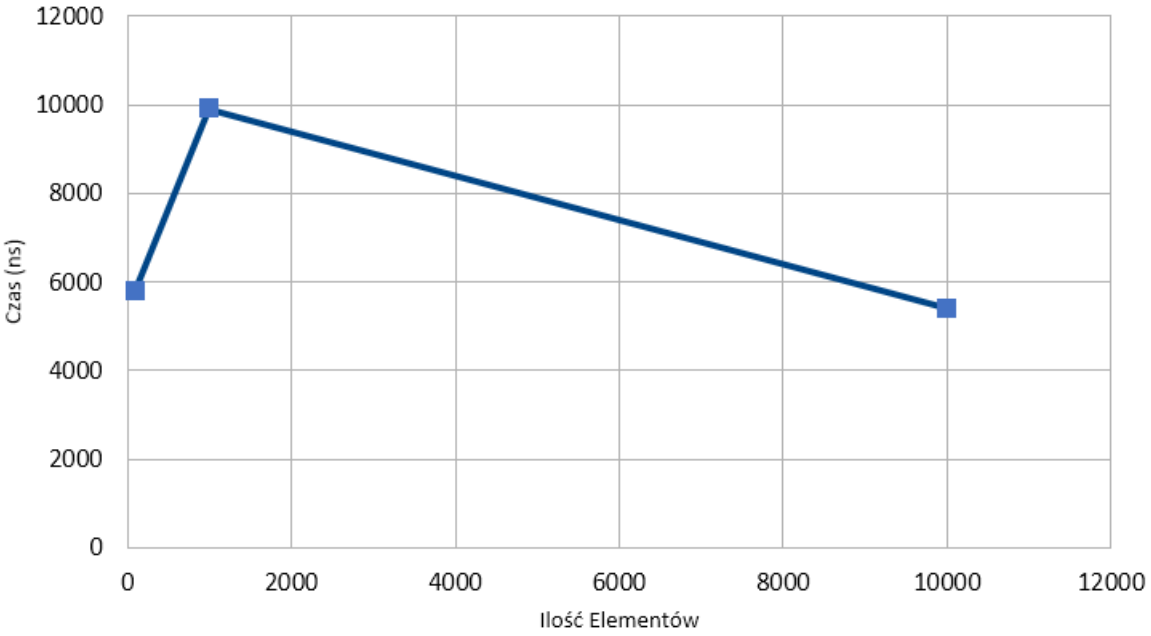
Poczynając od głowy przeszukujemy zawartości komórek (klucze) pod kątem szukanej wartości. Przechodząc z elementu na element posługujemy się wskaźnikami zawierającymi adres następnego elementu. Operacja zwraca wartość boolowską.

7. Pomiar funkcji listy

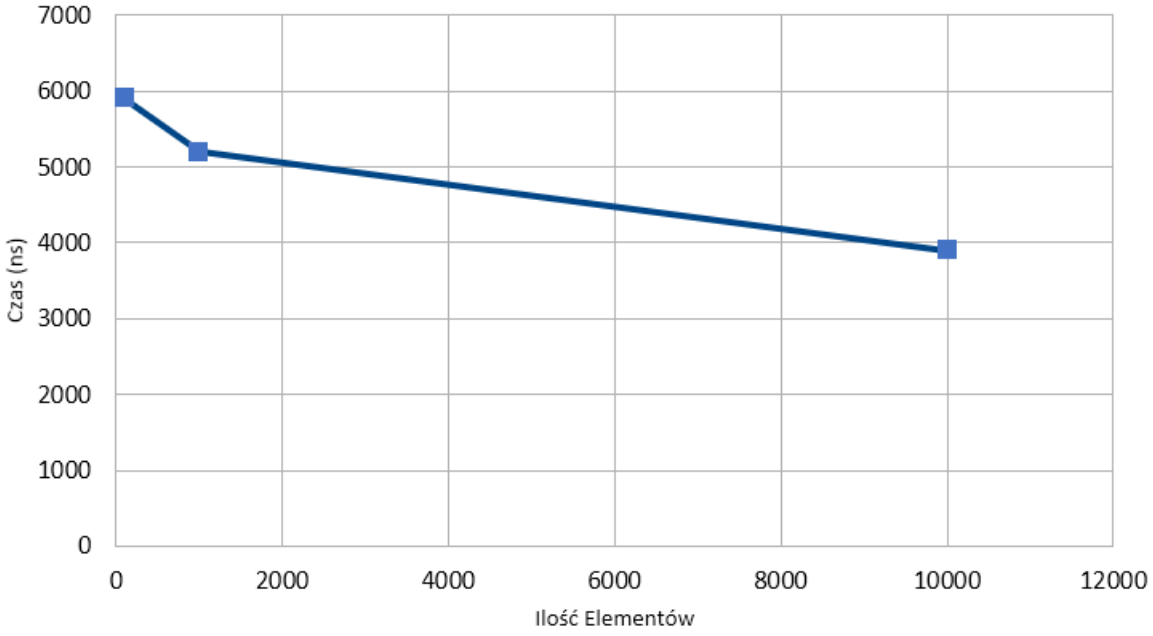
- Dodawanie na początku



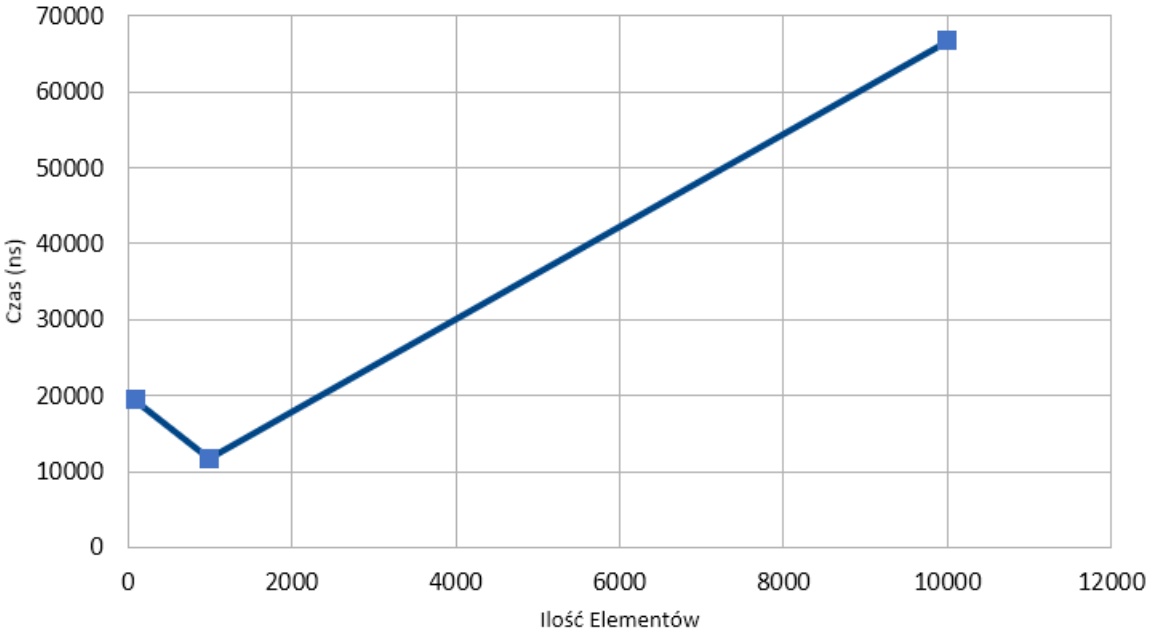
- Dodawanie na końcu



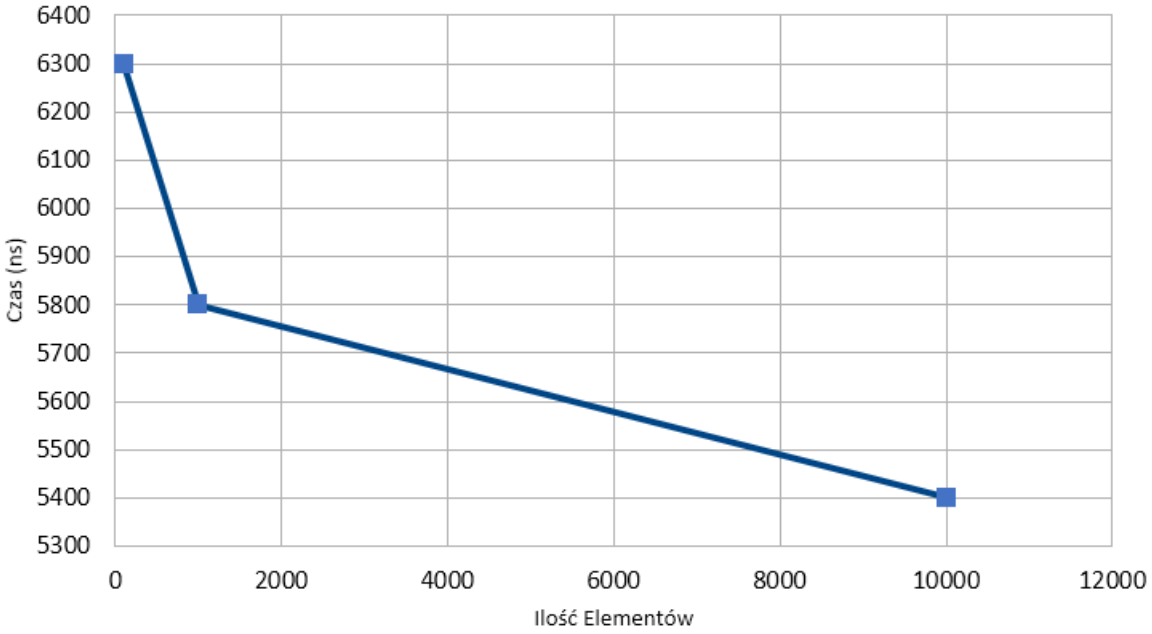
- Dodawanie na dowolnej pozycji



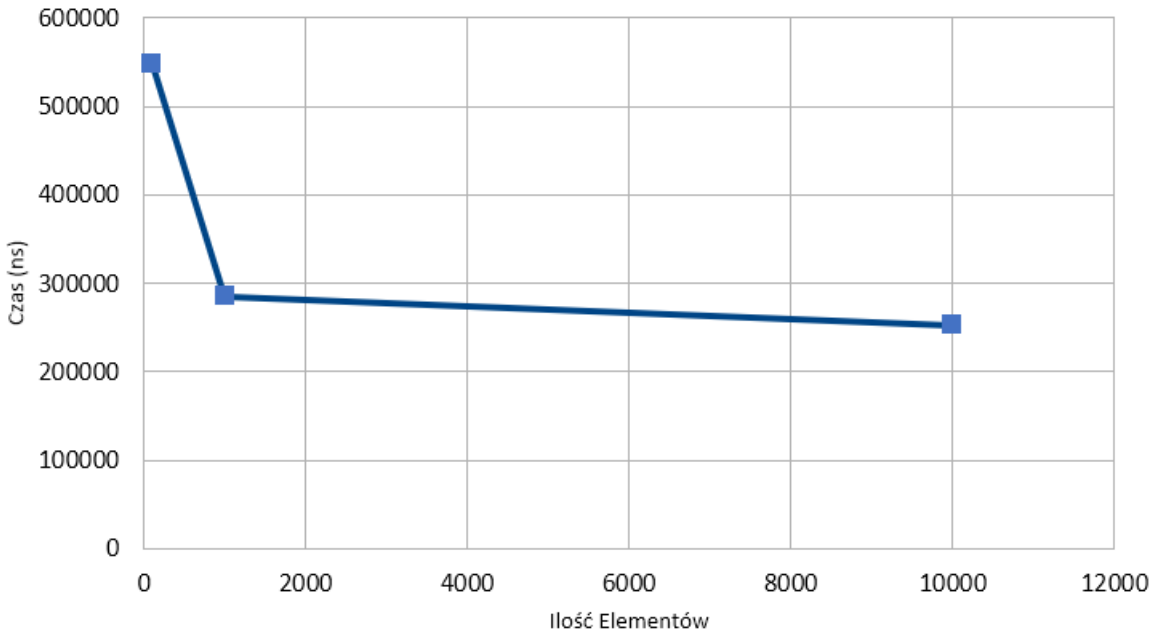
- Usuwanie z początku



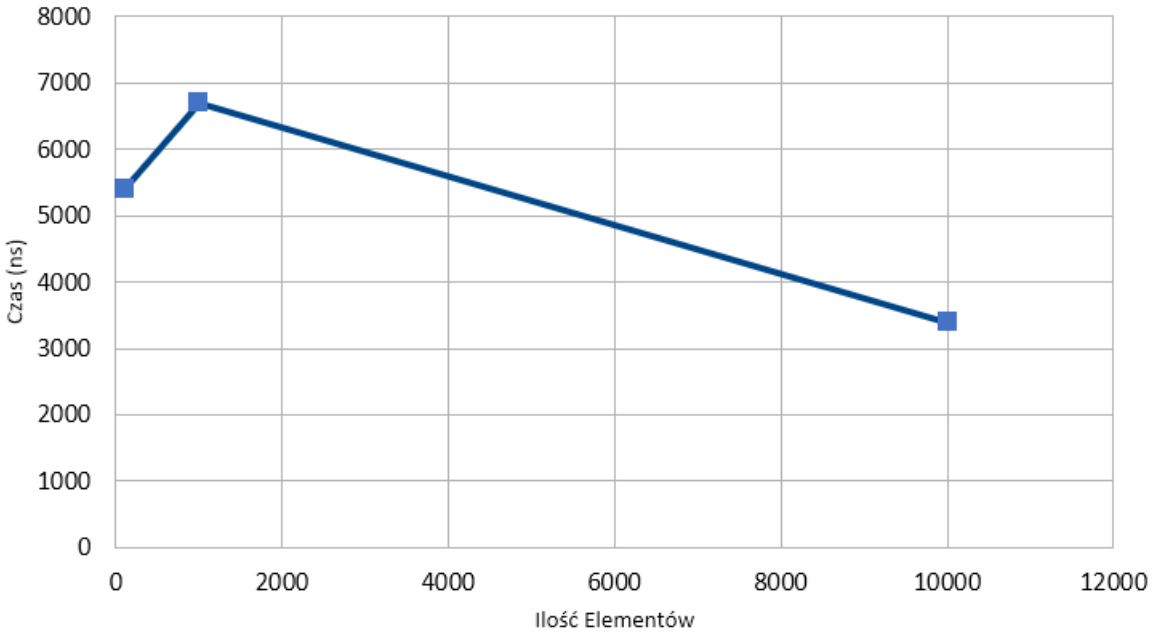
- Usuwanie z końca



- Usuwanie z dowolnej pozycji



- Wyszukiwanie



8. Implementacja kopca

Kopiec binarny jest kompletnym drzewem binarnym, co oznacza, że posada zapełnione wszystkie poziomy za wyjątkiem ostatniego, a ostatni poziom jest zapełniany bez przerw, poczynając od strony lewej do prawej. Ideą kopca jest drzewo którego ‘korzeń’ – a więc węzeł na samej górze posiadający jedynie ‘dzieci’ – ma w sobie największą wartośc wybraną z struktury kopca. Każdy węzeł ma w sobie wartośc która jest większa od wartości z węzłach będących jego dziećmi oraz mniejszą od swojego ‘rodzica’. Węzły nie posiadające dzieci (a więc zestaw najmniejszych wartości ze struktury kopca) nazywamy liściami. W kodzie jednak kopiec jest nadal tabelą w której każdy element ma indeks, zawartość i wskaźniki. W tym prypadku jednak wskaźniki są adresowawne na rodzica i dzieci danego węzła/elementu. Aby oddać ‘pozycję’ elementu na drzewie bianrnym przypisywany jest mu dany indeks. Tak więc element zawierający największą wartość ma indeks 1 i jest korzeniem, jego dzieci mają indekszy 2 i 3 itd. W kopcu dodawanie i usuwane elementów jest bardziej skomplikowaną operacją i wymaga zbudowania drzewa praktycznie od nowa – tak właśnie działa relokacja pamięci w tej strukturze.

- Dodawanie elementu

Dodawany element jest umieszczany na ostatniej pozycji w tablicy reprezentującej kopiec, a więc jako najmniejszy ‘liść’. Potem jego wartość jest sprawdzana z wartością rodzica i jeślie jest od niej większy – zamieniamy te elementy miejscami. Idziemy tak długo aż warunek nie zostanie spełniony. Budujemy kopiec od nowa korzystając z nowej tablicy.

- Usuwawnie elementu

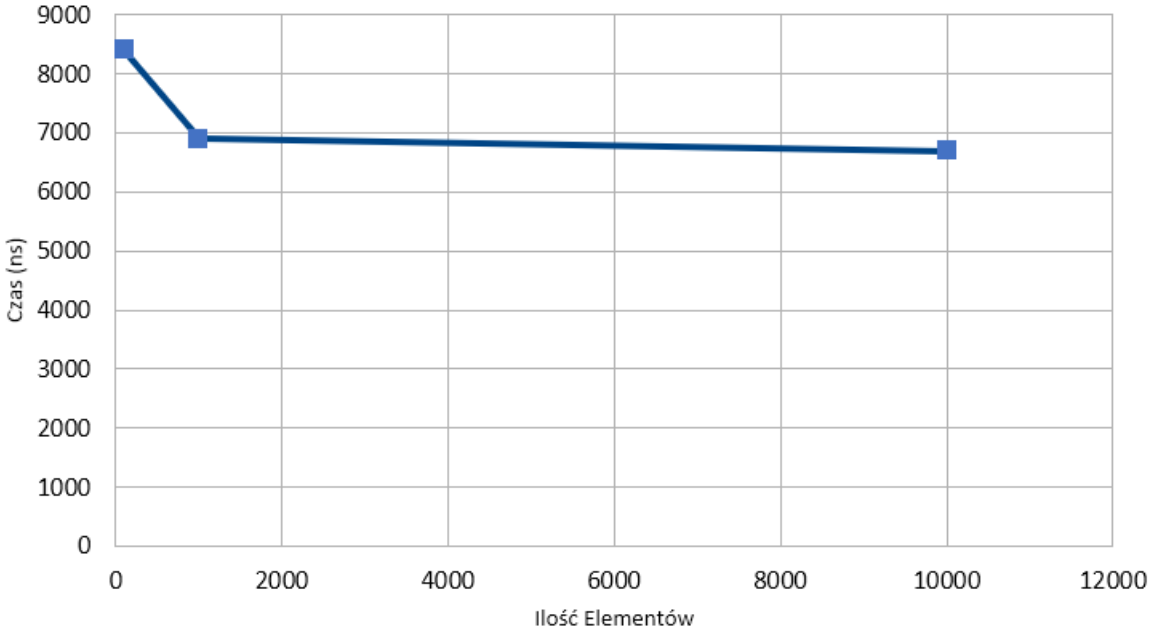
Wyrzucamy dany element z tabeli i ponownie budujemy kopiec z nowych wartości (przypisanych do nowych indeksów). W najgorszym przypdaku jest to wyrzucenie korzenia co oznacza zamianę pozycji wszystkich elementów.

- Wyszukiwanie zadanej wartości

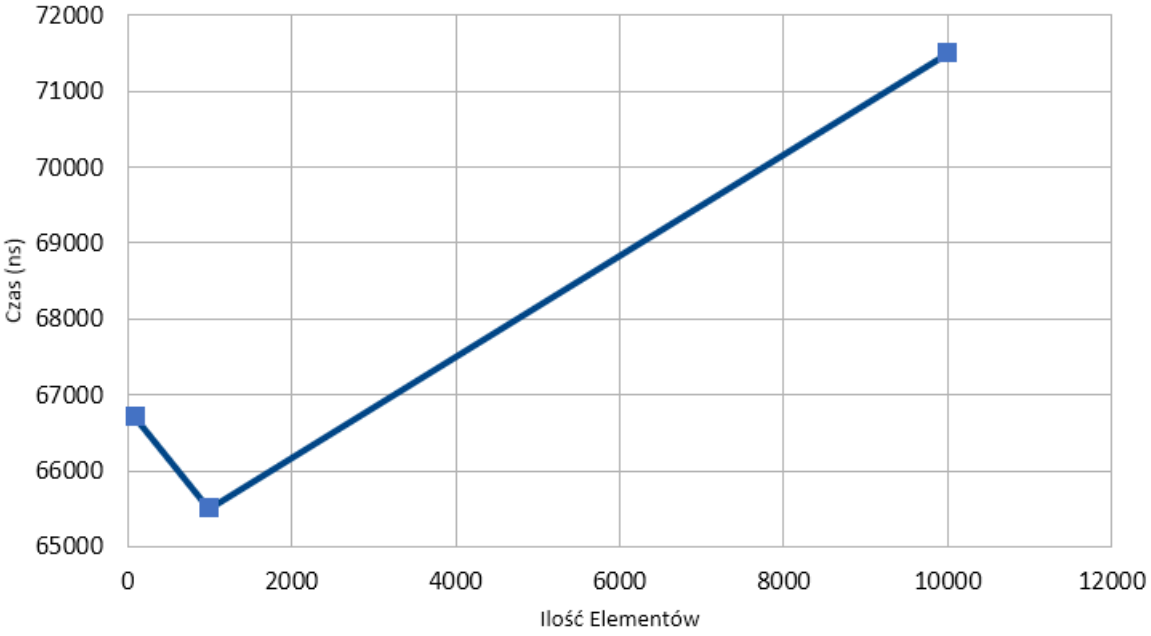
Wyszukiwanie elementu nie różni się niczym od innych struktur opartych na tabeli - Operacja zwraca wartość boolowską.

9. Pomiar funkcji kopca

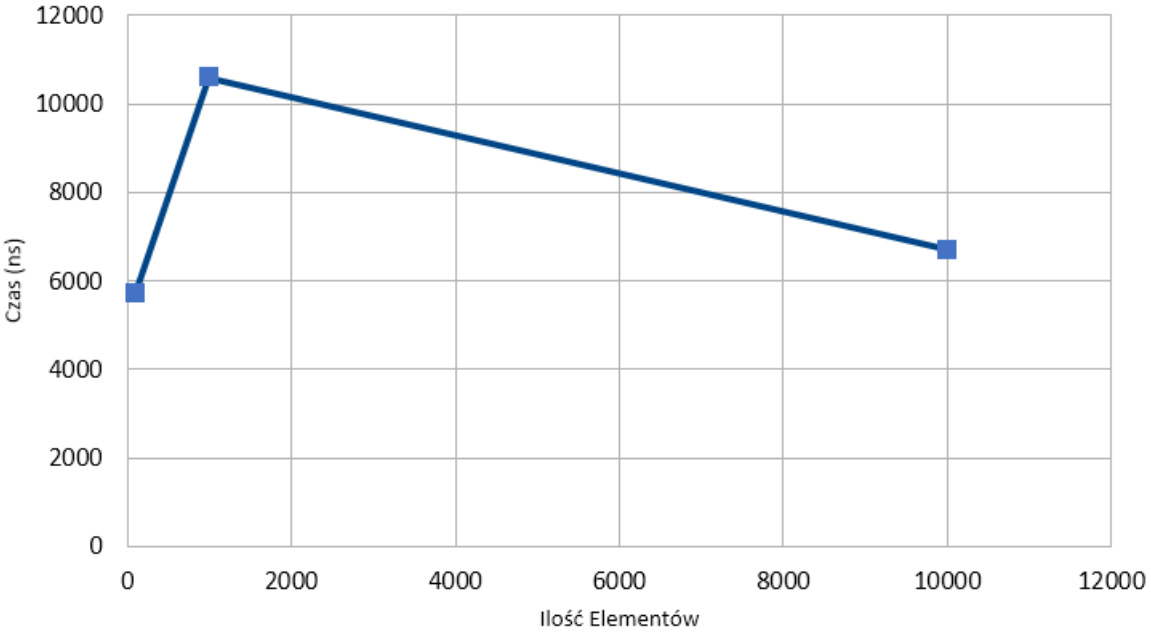
- Dodawanie



- Usuwanie



- Wyszukanie


12000
10000
8000
6000
4000
2000
0
Czas (ns)
0
2000
4000
6000
8000
10000
12000
Ilość Elementów

10. Implementacja drzewa czerwono-czarnego

Drzewa czerwono-czarne są odmianą samoorganizujących się drzew binarnych. W liściach drzew czerwono-czarnych nie przechowuje się danych. Liście te nawet nie muszą znajdować się bezpośrednio w pamięci komputera. Puste wskazanie w polu syna węzła może być interpretowane jako liść. Jednakże używanie rzeczywistych liści upraszcza niektóre algorytmy na drzewach czerwono-czarnych. W celu zaoszczędzenia pamięci często wybiera się pojedynczy węzeł, nazywany potocznie strażnikiem. W takim przypadku węzły wewnętrzne w drzewie czerwono-czarnym w polach synów-liści przechowują wskazania do tego węzła strażnika. Puste drzewo zawiera jedynie węzeł strażnika.

- Dodanie elementu

W odróżnieniu od drzew BST, wstawiając element musimy pamiętać, aby zachować zrównoważenie drzewa. Wstawienie elementu w dowolnym miejscu może powodować zaburzenie struktury kolorystycznej drzewa. Aby uniknąć pomyłek należy zastosować następujący algorytm:

1. Początkowo wstawiamy element tak, jak do standardowego drzewa BST.
2. Kolor każdego nowo dodanego elementu jest czerwony.
3. Jeżeli rodzic wstawionego węzła jest czarny to własność drzewa została zachowana.
4. Jeżeli rodzic wstawionego węzła jest czerwony to własność 3 została zaburzona (rodzic i syn mają kolor czerwony). Aby przywrócić własność należy przekolorować wybrane węzły i zmienić relację między konfliktującymi węzłami.

- Usunięcie elementu

Podobnie, jak w przypadku wstawiania, usuwanie wymaga dodatkowej uwagi w celu zachowania zrównoważenia drzewa. Tym razem, zamiast martwić się o rodzica wstawianego elementu, skupić należy uwagę na kolorze usuwanego węzła. Należy pamiętać, że jeżeli usuwany wierzchołek jest czerwony, czarna wysokość drzewa nie jest zakłócona, natomiast jeżeli usuwany wierzchołek jest czarny należy naprawić wysokość dla każdej ścieżki w drzewie.

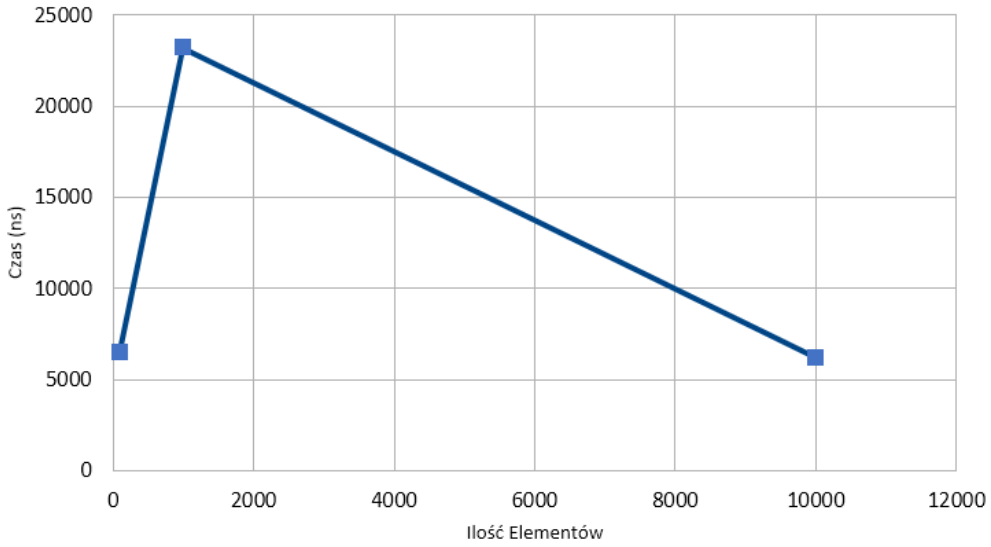
- Wyszukanie wartości

Wyszukiwanie elementu o kluczu k odbywa się tak samo jak wyszukiwanie w standardowym drzewie BST. Funkcja wyszukująca jako parametry przyjmuje wskaźnik do korzenia drzewa oraz

wartość do znalezienia. Funkcja boolowska zwraca prawdę, jeżeli element został znaleziony, lub fałsz, jezeli nie wystepuje w drzewie.

11. Pomiary drzewa czerwono-czarnego

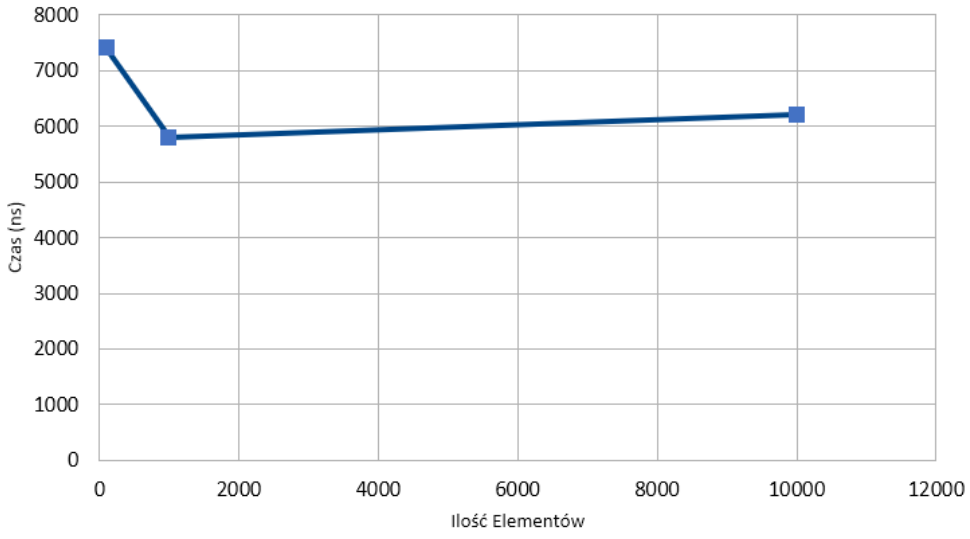
- Dodawanie



- Usuwanie

Usuwanie nie udało się całkiem zrealizować.

- Wyszukanie



12. Wnioski

W większości przypadków czas zatracony jest podobny do oczekiwanego przez funkcję, ale, czasem to nie jest prawdą, ponieważ czas zależy jeszcze i od tego, ile było potrzebne na relokację pamięci. Najlepiej to widać w liście (LinckedList).

Też można zauważyć, że w tablice i liście wartość elementu nie ma wpływu na czas operacji. Natomiast ma ona znaczenie w prawie każdej funkcji związanej z kopcem lub dzewem. To jest spowodowane tym, że ArrayList'a oraz LinkedList'a nie uwzględniają wartości które są dodawane, w odróżnieniu od Heap'a i drzewa czerwono-czarnego.

Można wywnioskować, że tablica posiada jedynie adres i wartości które mogą być łatwo wczytane i zmienione, po znajomości indeksu.

Lista zawiera wskaźniki na poprzednika i następnika co pozwala na lepszą orientację w zawartości lecz wydłuża czas operacj gdyż musi ona też zmieniać wartości wskaźników oraz dostęp do samego elementu jest dłuższy bo znajomość indeksu tu nie pomaga, bo trzeba iterować się od początku lub końcu.

Kopiec ustawia elementy zależąc od ich wartości, to pozwala na szybkie wyszukiwania największego/najmniejszego elementu, natomiast modyfikacja jest utrudniona.

Drzewo czerwono-czarne pozwala szybko wyszukiwać elementu po ich wartościach, bo nie mamy potrzeby przechodzić przez całość, a w najgorszym przypadku ilość poziomów dzewa.

Podsumowując, mogę powiedzieć, że ten projekt był dla mnie ciekawym oraz dobrym doświadczeniem. Było ciekawym porównania czasów różnych operacji w różnych strukturach. Chociaż nie udało mi się napisać całkiem działające drzewo czerwono-czarne.