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

Imię i nazwisko autora: Jakub Szpak

Numer indeksu: -

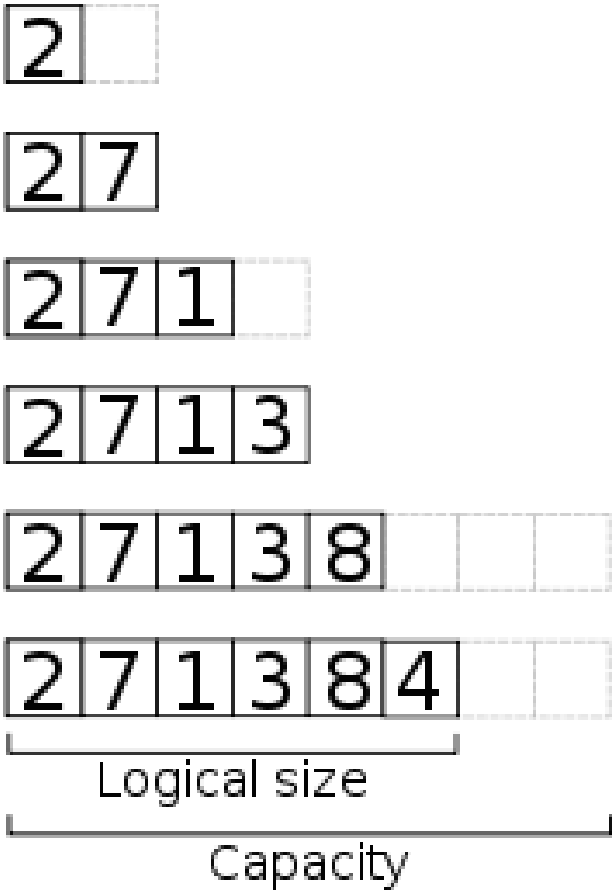
Grupa projektowa: E06-93b Poniedziałek 9:15 P

Nazwisko prowadzącego: Sterna

Wprowadzenie

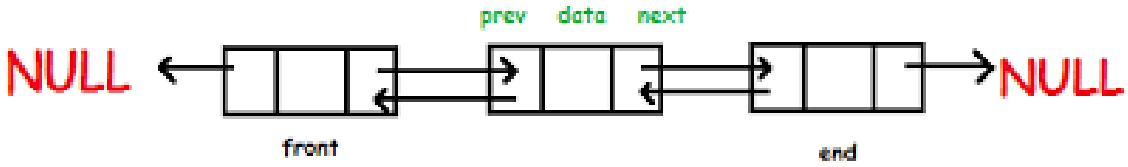
Tematem projektu było badanie efektywności operacji dodawania, usuwania oraz wyszukiwania elementów w danych strukturach danych. Zadanie wykonano na strukturach danych takich jak tablica dynamiczna, lista dwukierunkowa, kopiec binarny i binarne drzewo poszukiwań. Każdą z tych struktur cechują jej specjalne własności, których odpowiednie zrozumienie pozwala na wybranie najlepszej struktury danych dla własnych potrzeb. Kopiec binarny np. cechuje łatwy dostęp do największego lub najmniejszego elementu zależnie od wariantu (wybrany został wariant kopca typu max). Znajomość tych cech daje programiście możliwości wyboru najlepszego rozwiązania co na dużą skalę powinno przynieść ogromne korzyści w wydajności (czasie wykonywania danych operacji). Mimo to często młodzi developerzy pomijają, ten jakże ważny proces rozplanowywania swojego projektu. Nieświadomi pozbywają się tym możliwości późniejszego skalowania programu i sprzedaży czy pomocy innych programistów w postaci dalszego rozwoju Open Source. Dlatego celem tego projektu jest opis i pomiary złożoności poszczególnych operacji na strukturach. Najpierw jednak wstęp teoretyczny który pokaże jakich wyników powinniśmy się spodziewać po książkowej implementacji następujących struktur danych.

Tablica dynamiczna



Tablica dynamiczna to jedna z najbardziej podstawowych struktur danych. Powinna być ona dobrze znana przez każdego programistę. Charakteryzuje się możliwością zwiększenia jej maksymalnego rozmiaru podczas działania programu. W większości współczesnych implementacji zastosowany jest wariant, gdzie maksymalny rozmiar tablicy zwiększany jest kilku krotnie, w momencie gdy próbujemy dodać element do końca tablicy, której aktualny rozmiar jest równy jej tymczasowej maksymalnej objętości. W przypadku implementacji projektowej jednak zastosowana została strategia bez alokacji pamięci na zapas. Logicznym jest zatem, że wyniki niektórych złożoności obliczeniowych takich jak dodanie czy usunięcie elementu będą stosownie niezgodne. Dzieje się tak, ponieważ każda z tych operacji wiąże się z alokacją nowej tablicy o rozmiarze różniącym się o 1 i następnie przepisaniu starej tablicy do nowo zarezerwowanego miejsca w pamięci. W rezultacie otrzymujemy wolniejszy wynik ponieważ w wyżej opisanym wariancie wybierając współczynnik skalowalności za każdym razem zwiększaliśmy zaalokowane miejsce np. dwukrotnie. Według literatury dodanie, usunięcie oraz wyszukanie elementu w tablicy mają liniową złożoność czasową O(n) a uzyskanie dostępu do elementu mając dany indeks można wykonać w czasie stałym O(1).

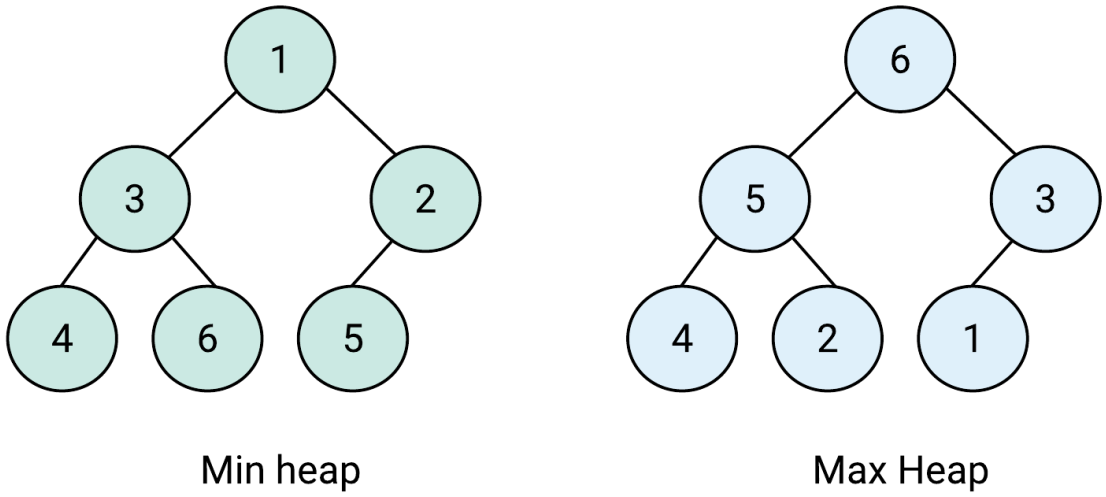
Lista dwukierunkowa



Kolejną strukturą danych którą należało zaimplementować jest lista dwukierunkowa. Lista dwukierunkowa składa się z węzłów, z których każdy z nich posiada wskaźnik na element następny i poprzedni. Jeśli element jest ostatnim elementem w liście to jego wskaźnik na następnika ustawiamy na wartość NULL, co wykorzystujemy później podczas sprawdzania czy doszliśmy już do końca listy. Podobnie wygląda to w przypadku wskaźnika na poprzednika elementu pierwszego w liście. Dwukierunkowość listy może wydawać się niepotrzebna na pierwszy rzut oka, ponieważ zajmuje ona więcej miejsca w pamięci od standardowej listy jednokierunkowej, jednak po przyjrzeniu się jej ma ona dużo zalet. Posiadając wskaźnik na głowę czyli pierwszy element listy oraz wskaźnik na ogon czyli ostatni element listy otwierają się przed programistą takie możliwości jak zaimplementowanie algorytmów szukających (w liście) czy wyświetlających listę od przodu oraz od tyłu w bardzo efektywnym czasie. W liście dwukierunkowej wyróżniłem cztery warianty dodania elementu: dodanie elementu jako nowej głowy listy, dodanie elementu jako nowego ogona listy, dodanie elementu po danym węźle, dodanie elementu przed danym węzłem. Jeśli chodzi o usuwanie to skróciło się to do trzech wariantów: usunięcia głowy, usunięcia ogona oraz usunięcia danego węzła ze środka. Literatura w tym wypadku podpowiada, że złożoność czasowa operacji na dwukierunkowej liście jest taka sama jak w liście jednokierunkowej (jak wspomniałem wcześniej wadą jest w dużej mierze dodatkowej miejsce w pamięci zajmowane przez dwukierunkową listę), wstawianie i usuwanie elementu wykonywane są w czasie stałym czyli O(n). Wynika to z tego,

że tak naprawdę te operacje to nic innego niż odpowiednie poprzestawianie odpowiednich wskaźników sąsiadujących elementów. Szukanie oraz dostęp do poszczególnych węzłów natomiast charakteryzuje się złożonością O(n).

Kopiec binarny

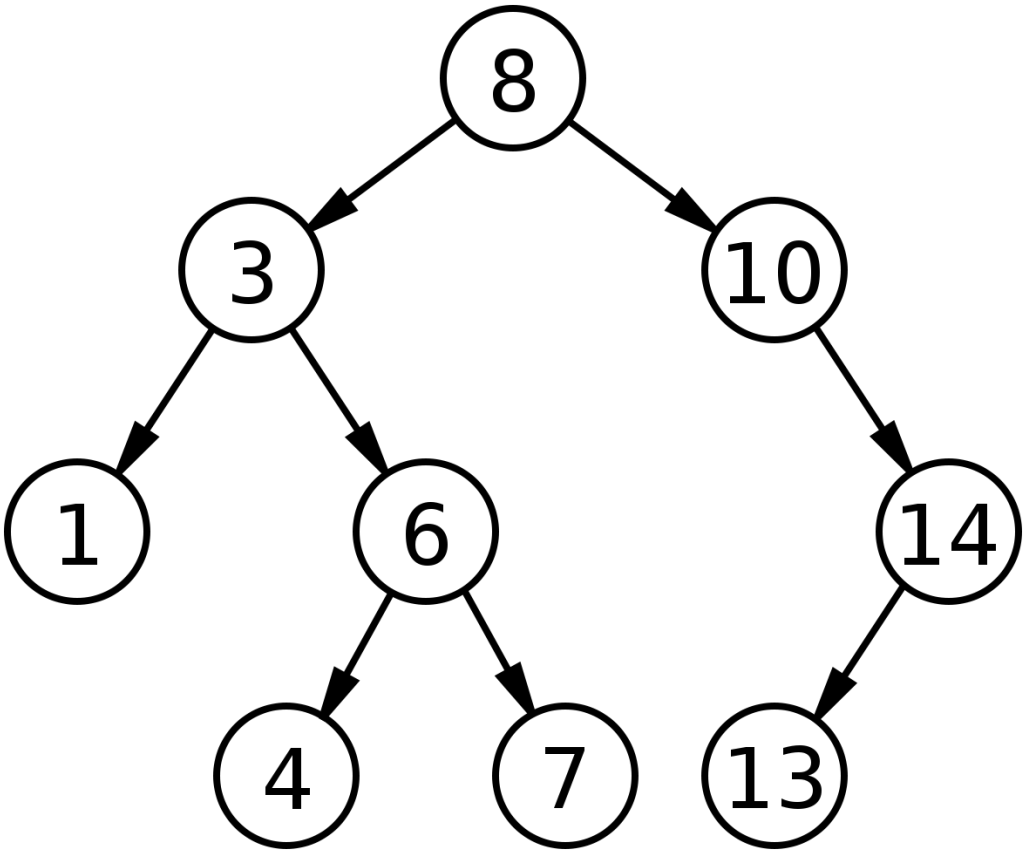


Następną strukturą danych, która również powinna należeć do zestawu każdego programisty jest kopiec binarny. Kopiec to bardzo specyficzna struktura, której odzwierciedlenia można się dopatrzyć w wielu systemach w życiu codziennym. Reprezentuje ona drzewo binarne, którego wszystkie poziomy z wyjątkiem ostatniego muszą być pełne. Wyróżnia się dwa warianty kopca binarnego. Pierwszy z nich to typ max. Klucz każdego węzła w takim kopcu musi być większy, niż klucze dzieci tego węzła. Przeciwnie wygląda to w przypadku kopca w wariancie min, co oznacza że klucz węzła ma być mniejszy niż klucze jego dzieci. Podstawowym i pierwszym przychodzącym do głowy zastosowaniem dla takiej struktury danych jest implementacja za jego pomocą kolejki priorytetowej. Dlaczego? Własności kopca pozwalają na bardzo szybki oraz łatwy dostęp do elementów z najmniejszymi lub największymi kluczami (oczywiście zależnie od wariantu kopca). Dzieje

się tak, ponieważ mając taką strukturę, w której własność kopca jest zachowana, możemy być pewni, że największy (lub najmniejszy) element będzie zawsze korzeniem. Kopiec oprócz wariantów można zaimplementować na dwa sposoby: w postaci listy oraz w postaci tablicy dynamicznej. Na potrzebę projektu wybrałem wariant max na tablicy dynamicznej z punktu pierwszego. Dowiedzieliśmy się już, że dostęp do korzenia jest bardzo szybki. Teraz o innych operacjach. Wstawianie do kopca zależy od ilości poziomów drzewa. W pesymistycznym przypadku dodanie ma złożoność obliczeniową $O(\log n)$, choć w niektórych przypadkach wstawienie wykonywane jest w czasie stałym O(1). Jeśli chodzi o wyszukiwanie elementów w kopcu to wygląda ono identycznie jak wyszukiwanie w przypadku tablicy i jego złożoność obliczeniowa to O(n), a usuwanie w tym projekcie zostało ograniczone do wariantu zdjęcia korzenia, którego złożoność obliczeniowa wynosi również $O(\log n)$. Kluczową funkcją w implementacji jest wielokrotnie wywoływana funkcja rekurencyjna heapify odnawiająca własność kopca na danym węźle. W moim projekcie wygląda ona następująco:

```
// restore heap property
void Heap::max_heapify(int index)
{
    int l = left(index);
    int r = right(index);
    int largest = index;
    if (l < array.get_size() && array[l] > array[largest])
    {
        largest = l;
    }
    if (r < array.get_size() && array[r] > array[largest])
    {
        largest = r;
    }
    if (largest != index)
    {
        std::swap(array[largest], array[index]);
        max_heapify(largest);
    }
    return;
}
```

Binarne drzewo poszukiwań



Binarne drzewo poszukiwań to dynamiczna struktura danych będąca drzewem binarnym, w którym lewe poddrzewo każdego węzła zawiera wyłącznie elementy o kluczach mniejszych niż klucz węzła a prawe poddrzewo zawiera wyłącznie elementy o kluczach większych niż klucz węzła. Możliwość przechowywania w drzewie binarnych poszukiwań węzłów o tych samych kluczach zależy od potrzeb oraz implementacji. Węzły, oprócz klucza, przechowują wskaźniki na swojego lewego i prawego syna oraz na swojego ojca (zależnie od implementacji, w projekcie został wykorzystany wariant z ojcem). Wstawianie nowego klucza do takiego drzewa podobnie jak w kopcu charakteryzuje się możliwością wstawienia go tylko w jednym miejscu. Oznacza to że po wywołaniu metody insert klucz znajduje swoje miejsce w drzewie, tworzy tam nowy węzeł i przypisuje do niego wartość klucza.

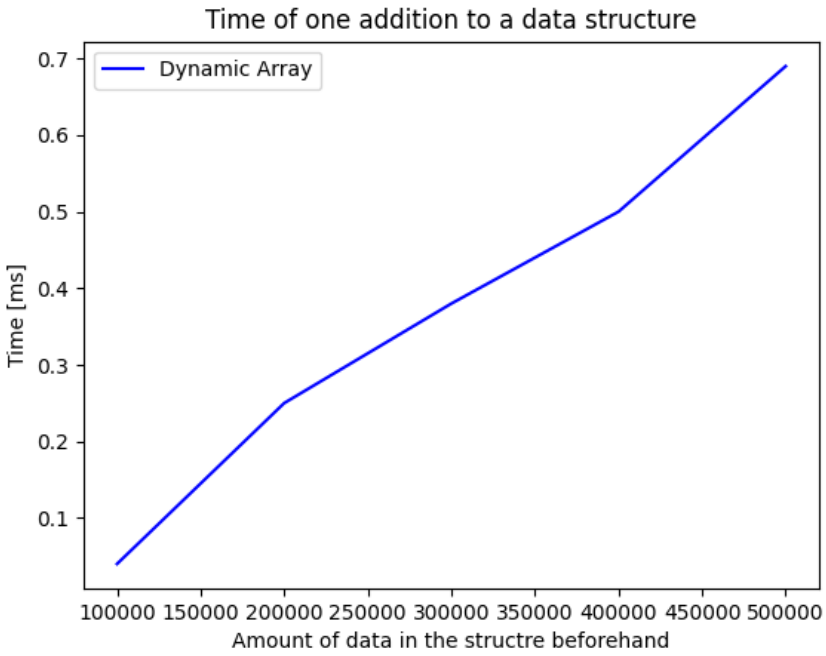
Największym czynnikiem wpływającym na złożoności obliczeniowe operacji na drzewie jest jego wysokość, ponieważ większość operacji podczas przechodzenia po drzewie porównuje klucz z kluczem węzła i na tej podstawie podejmuje decyzje czy iść dalej w jego lewe czy prawe poddrzewo. Często więc stosuje się algorytm DSW, który kosztem operacji czasowych doprowadza drzewo do jego najmniejszej możliwej wysokości. Mówimy wtedy o drzewie zrównoważonym. Na potrzebę tego projektu jednak, nie został on zaimplementowany, co przy odpowiednio dobranych danych testowych może prowadzić do dużych różnic w złożonościach obliczeniowych operacji. Dodawanie, usuwanie wyszukiwanie oraz dostęp do elementów przy zrównoważonym drzewie wykonuje się w czasie O($\log n$), jednak jeśli do drzewa kolejno dodamy elementy z posortowanej tablicy to takie drzewo zamienia się w nic innego, a listę z wieloma niepotrzebnymi wskaźnikami i porównaniami. Złożoność obliczeniowa w pesymistycznym przypadku sprowadza się więc do liniowej O(n).

Badania

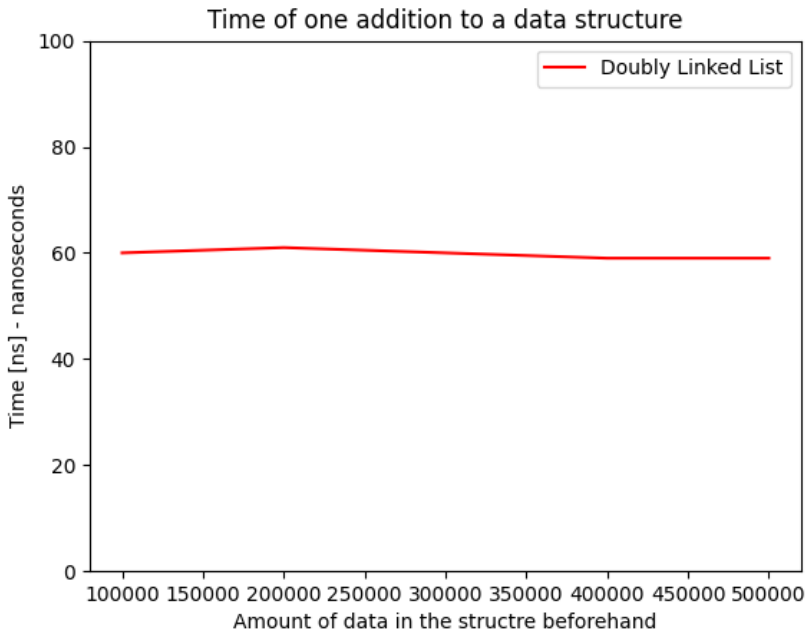
1. Dodawanie elementów do poszczególnych struktur

Gdy wyniki pomiaru czasu dodania jednego elementu do struktury były zbyt małe, postanowiłem dodawać po np. 10 lub 100 lub nawet 100 000 elementów (zależnie od czasów) do struktury (oczywiście w momencie, gdy miała ona już załadowaną odpowiednią ilość elementów) i dzielić taki wynik przez liczbę operacji. Uznałem, że np. w przypadku przesunięcie spowodowane tym, że czas dodania elementu do tablicy z 200 000'ami elementów jest mniejszy niż czas dodania elementu do tablicy z 200 100'ami elementów, jest zdecydowanie mniejsze niż zaokrąglanie zaoferowane przez kompilator. Pomiary mocno odstające od reszty zostały uznane jako błędy grube i nie były brane pod uwagę w obliczeniach.

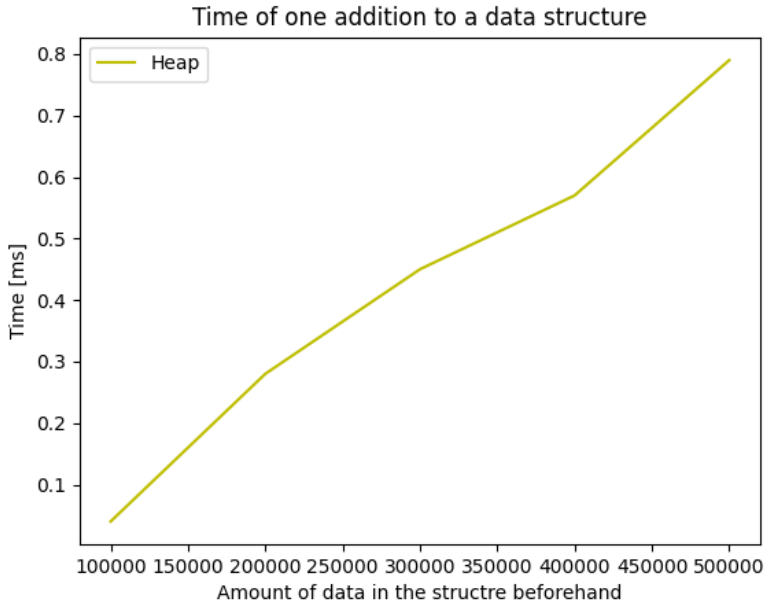
Czas jednego dodania/liczba elementów uprzednio	100 000	200 000	300 000	400 000	500 000	
Tablica dynamiczna	0.04ms [0.03-0.04]	0.26ms [0.25-0.27]	0.38ms [0.37-0.40]	0.50ms [0.50-0.51]	0.69ms [0.64-0.76]	
Lista dwukierunkowa [nanosekundy]	60ns [60-60]	61ns [56-65]	60ns [57-63]	59ns [58-60]	59ns [58-59]	
Kopiec binarny	0.04ms [0.03-0.04]	0.28ms [0.26-0.29]	0.43ms [0.40-0.45]	0.55ms [0.53-0.58]	0.77 [0.75-0.77]	
Drzewo binarne poszukiwań [nanosekundy]	250ns [200-300]	380ns [300-400]	520ns [500-600]	640ns [600-700]	850ns [800-900]	



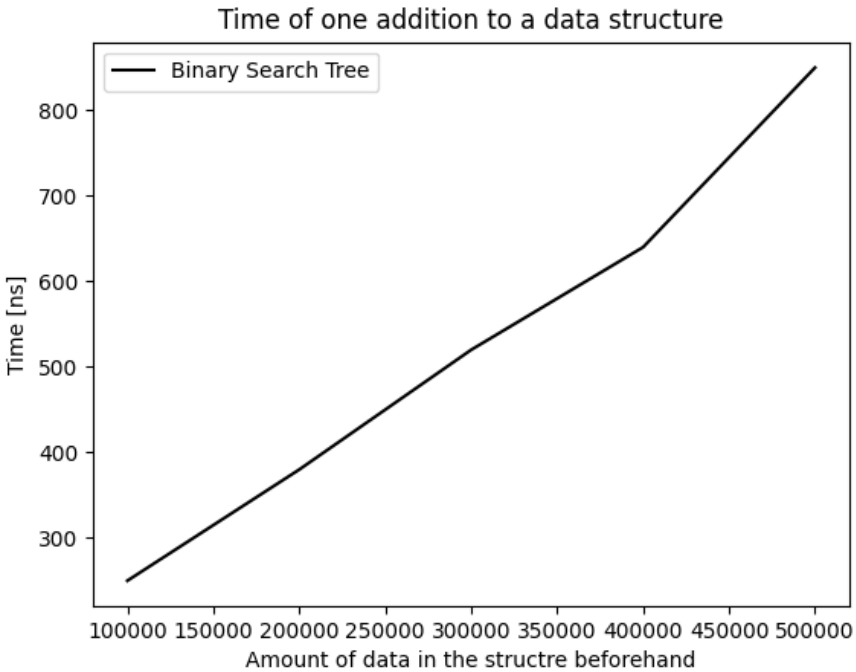
Wykres przedstawiający zmienność czasu dodania jednego elementu do tablicy dynamicznej w zależności od liczby elementów przechowywanych w niej przed dodaniem



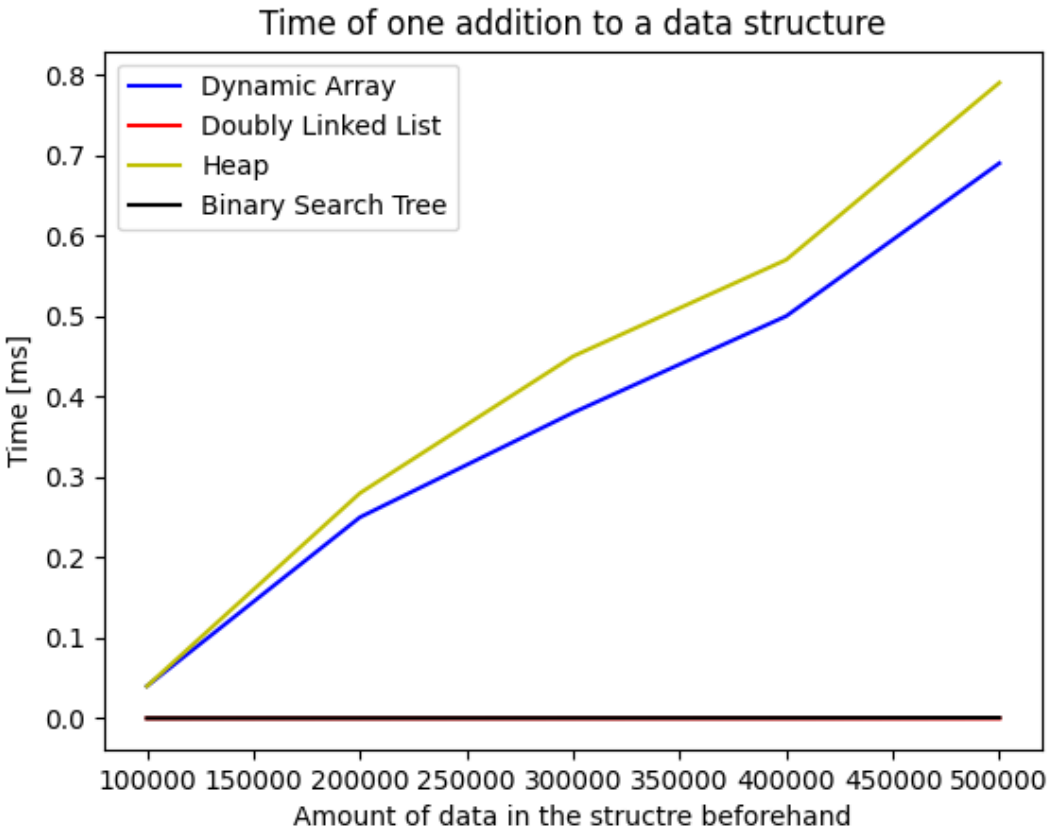
Wykres przedstawiający zmienność czasu dodania jednego elementu do listy dwukierunkowej w zależności od liczby elementów przechowywanych w niej przed dodaniem



Wykres przedstawiający zmienność czasu dodania jednego elementu do kopca binarnego w zależności od liczby elementów przechowywanych w nim uprzednio



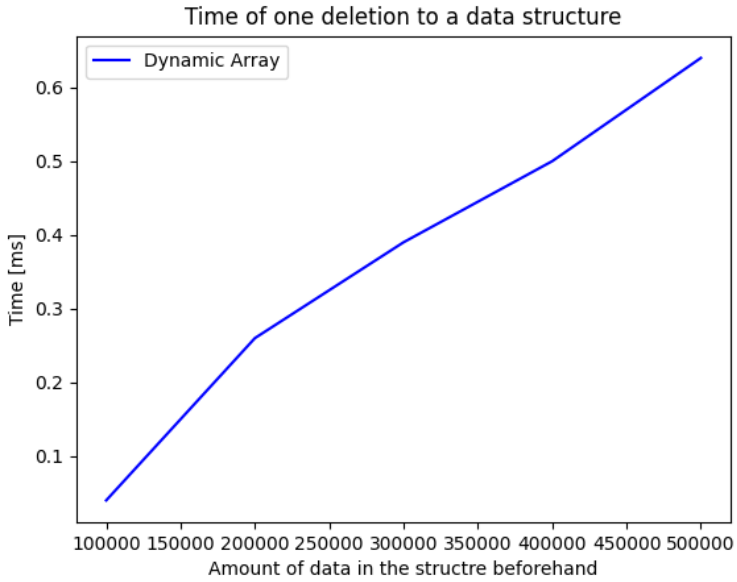
Wykres przedstawiający zmienność czasu dodania jednego elementu do drzewo binarnego poszukiwań w zależności od liczby elementów przechowywanych w nim uprzednio



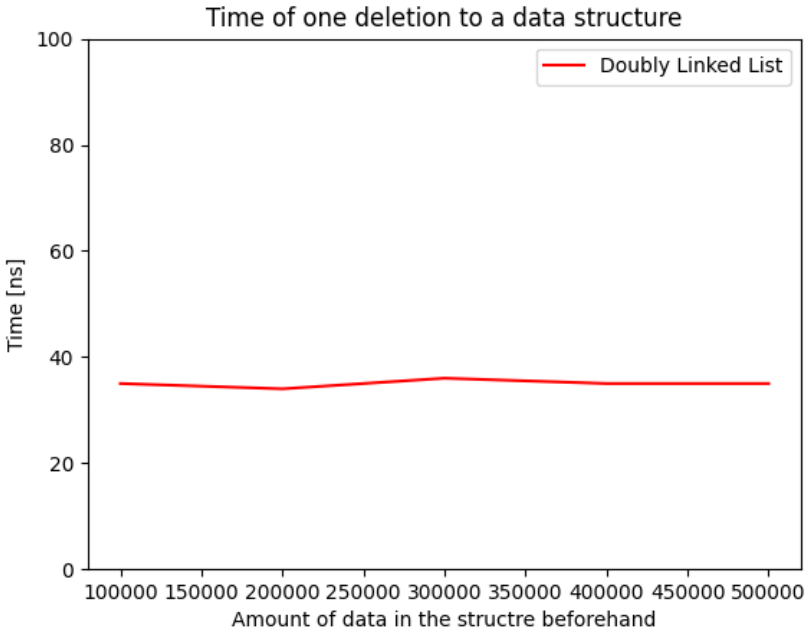
Wykres pokazujący jak przedstawiają się złożoności czasowe dodania jednego elementu, struktur zaimplementowanych na potrzebę projektu

2. Usuwanie elementów z poszczególnych struktur

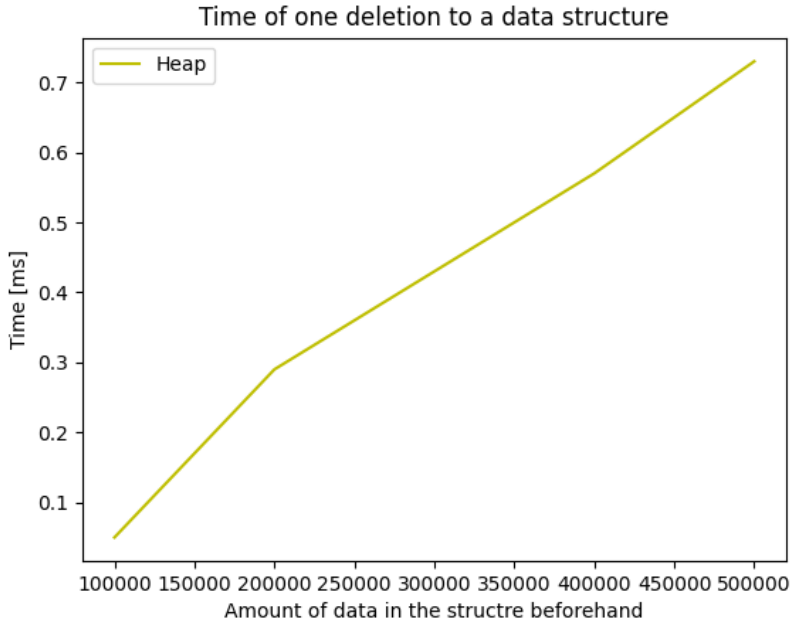
Czas jednego usunięcia/liczba elementów uprzednio	100 000	200 000	300 000	400 000	500 000	
Tablica dynamiczna	0.04ms [0.03-0.04]	0.26ms [0.26-0.27]	0.39ms [0.38-0.41]	0.50ms [0.49-0.52]	0.62ms [0.60-0.73]	
Lista dwukierunkowa [nanosekundy]	35ns [30-40]	34ns [30-40]	36ns [30-40]	35ns [30-40]	35ns [30-40]	
Kopiec binarny [delete_max()]	0.05ms [0.05-0.07]	0.29ms [0.28-0.30]	0.43ms [0.40-0.46]	0.57ms [0.55-0.60]	0.73ms [0.70-0.75]	
Drzewo binarne poszukiwań [nanosekundy]	125ns [120-130]	145ns [140-150]	150ns [145-155]	160ns [150-180]	165ns [160-170]	



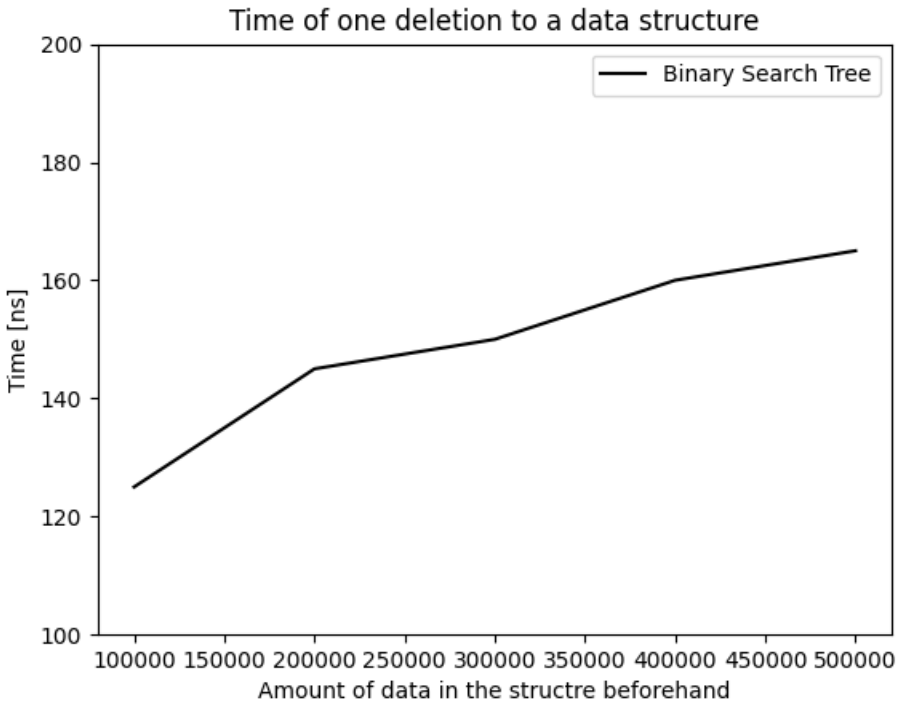
Wykres przedstawiający zmienność czasu usunięcia jednego elementu z tablicy dynamicznej w zależności od liczby elementów przechowywanych w niej uprzednio



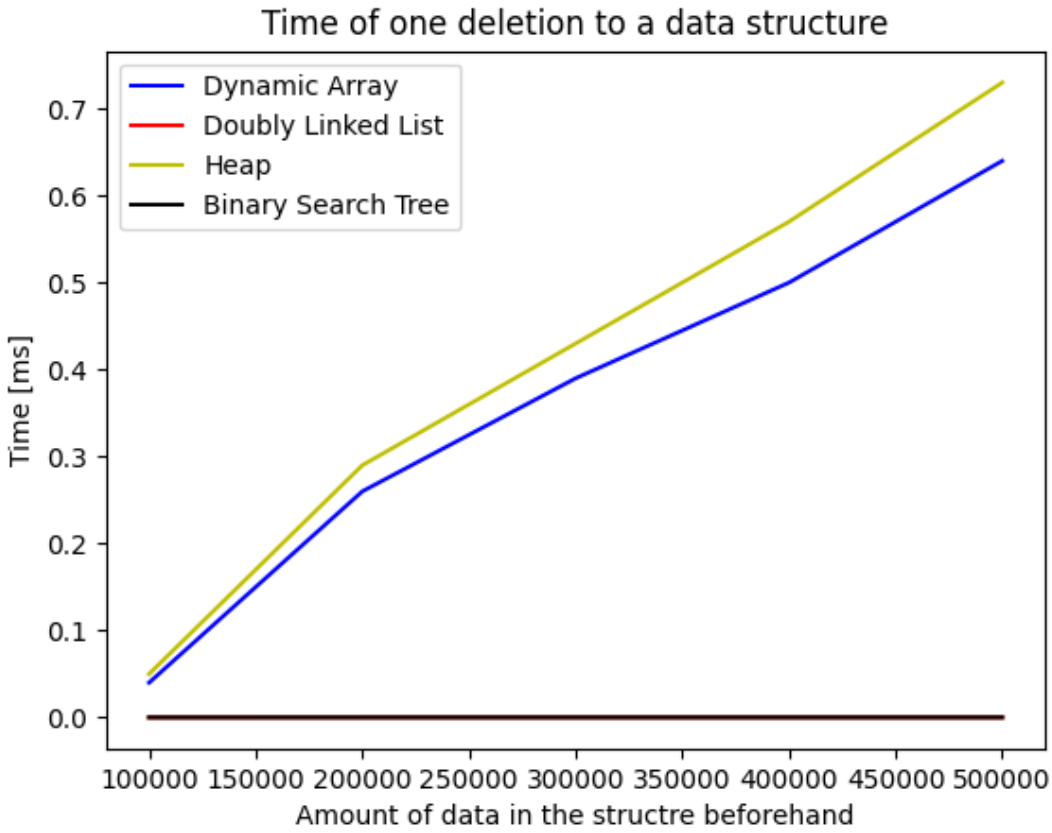
Wykres przedstawiający zmienność czasu usunięcia jednego elementu z listy dwukierunkowej w zależności od liczby elementów przechowywanych w niej uprzednio



Wykres przedstawiający zmienność czasu usunięcia jednego elementu z kopca binarnego w zależności od liczby elementów przechowywanych w nim uprzednio



Wykres przedstawiający zmienność czasu usunięcia jednego elementu z drzewa binarnego poszukiwań w zależności od liczby elementów przechowywanych w nim uprzednio



Wykres pokazujący jak przedstawiają się złożoności czasowe usunięcia jednego elementu, struktur zaimplementowanych na potrzebę projektu

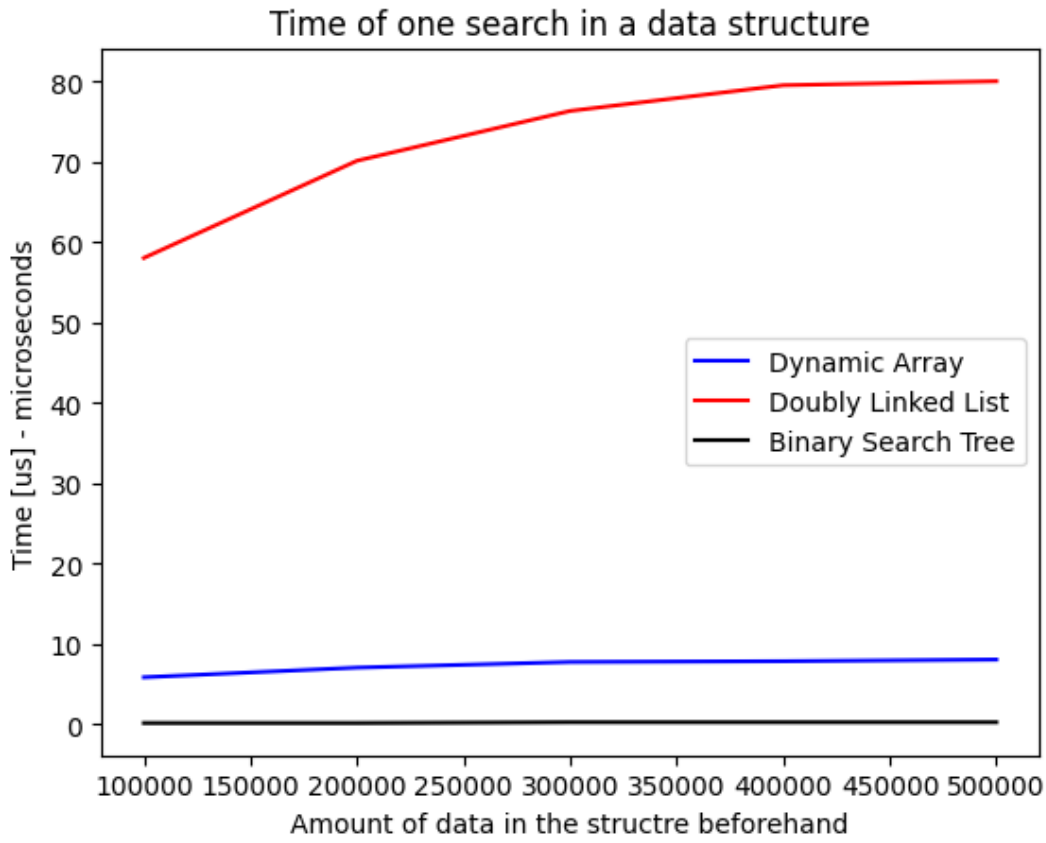
3. Wyszukiwanie jednego elementu

W pomiarze złożoności czasowej wyszukiwania elementu w strukturze pominięty został kopiec binarny, ponieważ jego wyniki pokrywały się z wynikami tablicy dynamicznej. Jest to spowodowane tym, że kopiec został zaimplementowany w wariancie na tablicy i metoda search w kopcu wywołuje po prostu metodę search z tablicy.

Średni czas jednego wyszukania/liczba elementów uprzednio	100 000	200 000	300 000	400 000	500 000	
Tablica dynamiczna [mikrosekundy]	5.8µs [5.7-5.8]	7µs [6.8-7.2]	7.7µs [7.5-8.1]	7.8µs [7.8-7.9]	8µs [7.8-8]	
Lista dwukierunkowa [mikrosekundy]	58µs [57.6-59.0]	70.1µs [69.0-71.2]	76.3µs [75.8-77.9]	79.5µs [77.3-82.7]	80µs [77.8-83.7]	
Drzewo binarne poszukiwań [mikrosekundy]	0.1µs [0.1-0.2]	0.1µs [0.1-0.2]	0.2µs [0.1-0.2]	0.2µs [0.1-0.2]	0.2µs [0.1-0.2]	

Niezgodność wyników z literaturą

Wyszukanie elementu w strukturze danych to nic innego niż znalezienie pierwszego napotkanego elementu, którego klucz równa się zadanemu przez nas kluczowi i zwrócenie jego pozycji lub wskaźnika na dany element. Przy bardzo dużych strukturach napotykamy na problem związany z wiarygodnością pomiarów, ponieważ na czas pomiaru bardzo duży wpływ ma to jak daleko w strukturze znajduje się szukany przez nas element. Jeśli wylosujemy szukanie węzła który znajduje się na pierwszym miejscu w tablicy dynamicznej, czas operacji będzie zdecydowanie krótszy, niż znalezienie elementu z końca takiej tablicy. W literaturze podany jest czas pesymistyczny. Na to nakłada się jeszcze powtarzalność elementów. Jako, że wszystkie struktury zostały zaimplementowane tak, aby mogły posiadać dwa węzły lub miejsca z tymi samymi wartościami. Prawie zawsze więc szukając losowej wartości w strukturze o bardzo dużych wymiarach, natrafimy na sytuacje, w której szukany klucz znajduje się w niej przynajmniej dwukrotnie a pomiar skończy się na pierwszej natrafionej przez algorytm instancji. Stąd właśnie we wszystkich pomiarach tendencja do wyrównywania się średniego czasu wyszukania wraz z wzrostem liczby elementów.



Wykres pokazujący jak przedstawiają się złożoności czasowe wyszukania pierwszego elementu ze wskazanym kluczem, struktur zaimplementowanych na potrzebę projektu

Wnioski, uwagi, refleksja

- większość operacji na zastosowanym wariancie tablicy bez alokacji pamięci na zapas okazało się mocno odstawać od innych struktur pod względem czasu jako że przy każdym dodaniu lub usunięciu elementu konieczne jest przeniesienie całej tablicy w nowe miejsce w pamięci (podczas projektu próbowałem również zaimplementować wersję z dwukrotnym zwiększaniem objętości z każdym przekroczeniem aktualnego limitu i ta wersja sprawowała się zdecydowanie lepiej)

- niezgodność w wyszukiwaniu opisana na poprzedniej stronie