2020

# Struktury Danych I
# Złożoność Obliczeniowa

PROJEKT

ALIAKSEI TOKARAU

# Spis Treści

Zadanie projektowe nr. 1 ..... 2
Wstęp teoretyczny ..... 3
Tablica dynamiczna ..... 4
Dodawanie elementu ..... 5
Usuwanie elementu ..... 8
Szukanie elementu o zdananym kluczu ..... 11
Wnioski ..... 12
Lista dwukierunkowa ..... 13
Dodawanie elementu ..... 14
Usuwanie elementu ..... 17
Szukanie elementu o losowym kluczu ..... 20
Wnioski ..... 21
Kopiec ..... 22
Dodawanie elementu ..... 23
Usuwanie elementu ..... 24
Wyszukiwanie elementu ..... 25
Wnioski ..... 26
Drzewo czerwono-czarne ..... 27
Dodawanie elementu ..... 28
Usuwanie elementu ..... 29
Poszukiwanie elementu ..... 30
Wnioski ..... 31
Literatura ..... 31

# Zadanie projektowe nr. 1

**Data wykonania projektu:** 02.02.2020

**Data oddania sprawozdania:** 04.02.2020

**Grupa zajęciowa:** cz/TP+1/2 11:15-13:00

**Skład grupy:** Aliaksei Tokarau 250930

**Prowadzący:** Dr inż. Jarosław Mierzwa

## Cel projektu

Implementacja następujących struktur danych:

- Tablica
- Lista
- Kopiec binarnny
- Drzewo czerwono-czarne

Pomiar czasu wykonania następujących operacji:

- Dodawaie elementu
- Usuwanie elementu
- Wyszukiwanie elementu

## Przyjęte założenia

- Podstawowym elementem struktur danych jest 4 bajtowa liczba całkowita ze znakiem
- Wszystkie struktury danych są alokowane dynamicznie
- Ponieważ pojedynczy pomiar czasu jest obarczony dużym błędem, pomiary wykonane są wielokrotnie (100 razy) i otrzymana wartość jest uśredniona
- Środowiskiem programistycznym jest Clion 2019.3

# Wstęp teoretyczny

W informatyce struktura danych jest to organizacja danych, zarządzanie i format przechowywania danych, który umożliwia efektywny dostęp i modyfikację danych. Struktura danych jest zbiorem wartości danych, zależności między nimi oraz funkcji lub operacji, które mogą być stosowane do danych.

Różne rodzaje struktur danych są dostosowane do różnych rodzajów aplikacji, a niektóre są wyspecjalizowane do konkretnych zadań. Na przykład relacyjne bazy danych często używają indeksów binarnego drzewa poszukiwań, podczas gdy implementacje kompilatorów zwykle używają hash tabel do wyszukiwania identyfikatorów. Struktury danych umożliwiają efektywne zarządzanie dużymi ilościami danych do zastosowań takich jak duże bazy danych i usługi indeksowania internetu. Zazwyczaj efektywne struktury danych są kluczem do projektowania wydajnych algorytmów. Niektóre formalne metody projektowania i języki programowania podkreślają struktury danych, a nie algorytmy, jako kluczowy czynnik organizacyjny w projektowaniu oprogramowania.

Złożoność obliczeniowa lub po prostu złożoność algorytmu to ilość zasobów potrzebnych do jego uruchomienia. Szczególny nacisk kładzie się na wymagania czasu i pamięci. Ponieważ ilość zasobów wymagana do uruchomienia algorytmu zasadniczo różni się w zależności od wielkościdanych wejściowych, złożoność wyraża się zwykle jako funkcję n → f(n), gdzie n jest wielkością wejścia, F(N) jest albo najbardziej pesymistyczną złożonością (maksymalną ilością zasobów, które są potrzebne na wszystkich danych o rozmiarze N), albo złożonością przeciętną (średnią ilością zasobów na wszystkich danych o rozmiarze n). Złożoność czasowa jest ogólnie wyrażona jako liczba wymaganych operacji elementarnych na wejściu o rozmiarze n, gdzie zakłada się, że operacje elementarne zajmują stałą ilość czasu na danym komputerze i zmieniają się tylko o stały współczynnik przy uruchamianiu na innym komputerze. Złożoność przestrzeni wyraża się zwykle jako ilość pamięci wymaganą przez algorytm na wejściu o rozmiarze N.

Różne struktury mają różną złożoność obliczeniową(czasową i pamięciową), różnych operacji, dlatego programista wybierając strukturę do przechowywania danych kieruje się najczęściej złożonością obliczeniową operacji(algorytmu), która będzie najczęściej wykonywana i potrzebna, dla danej wielkości struktury danych. Czasem w zależności od dostępnego sprzętu i oprogramowania programista musi wybrać algorytm o jak najniższej pamięciowej złożoności obliczeniowej, kosztem znacznego zwiększenia czasowej złożoności obliczeniowej.

# Tablica dynamiczna

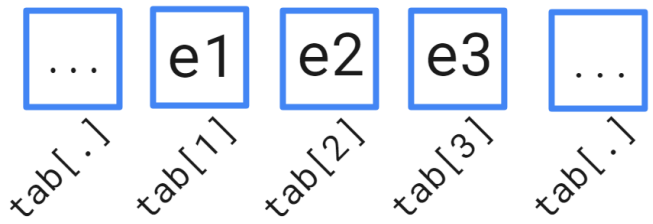
Tablica, to struktura danych składająca się ze zbioru elementów (wartości lub zmiennych), z których każdy jest identyfikowany co najmniej jednym indeksem tablicy lub kluczem. Tablica jest przechowywana w taki sposób, że położenie każdego elementu można obliczyć z jego krotki indeksu za pomocą wzoru matematycznego. Najprostszym typem struktury danych jest tablica liniowa, zwana też tablicą jednowymiarową.

W tym przypadku relokujemy tablicę co każde dodanie/usunięcie co oznacza nizką pamięciową złożoność obliczeniową.

Zastosowania tablicy dynamicznej:

- Macierze 2D, zwane na ogół macierzami, są stosowane głównie w przetwarzaniu obrazów
- Przetwarzanie mowy, gdzie każdy sygnał mowy jest układem amplitud sygnału
- Obraz RGB to tablica N*N*3
- Tekstowy typ danych jest tablicą symboli

Przykład:




Teoretyczna czasowa złożoność oblizeniowa tablicy:

| Operacja | Średnia złożoność | Pesymistyczna złożoność |
|---|---|---|
| Dostęp (nie sprawdzamy) | O(1) | O(1) |
| Dodawanie elementu | O(n) | O(n) |
| Usuwanie elementu | O(n) | O(n) |
| Poszuk elementu | O(n) | O(n) |

## Dodawanie elementu

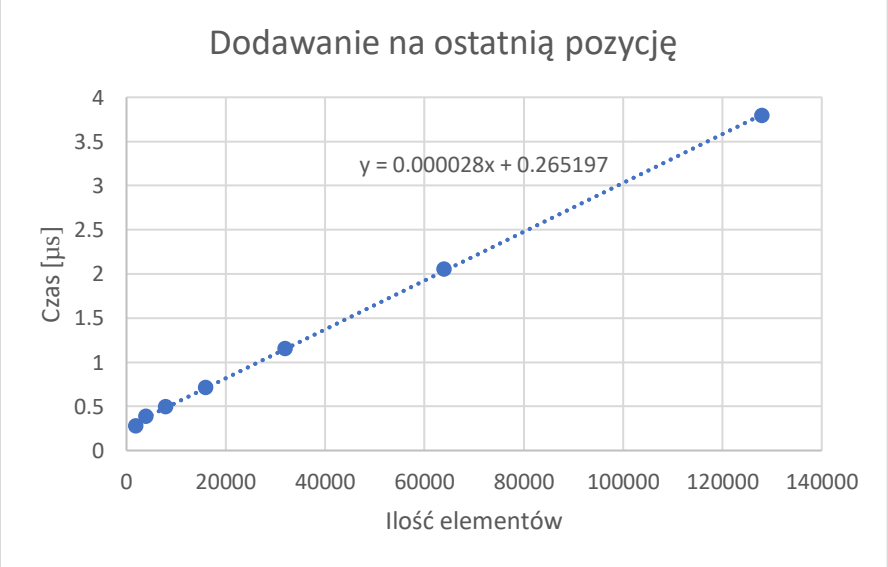
W przypadku tablicy operacja ta polega na alokowaniu nowego obszaru pamięci o rozmiarze *n+1* dla tablicy o rozmiarze *n*. W nowy obzar pamięci przepisujemy wartości z poprzedniej tablicy wraz z nowym elementem i zwalniamy wykorzystywany dotąd obszar pamięci. Dla danej operacji dobiramy 7 różnych punktów pomiarowych (2000, 4000, 8000, 16000, 32000, 64000, 128000 elementów) w każdym z nich dodajemy 200 elementów 100 razy. Rezultaty testu dzielimy na 20000 aby uzyskać czas dodawania 1 elementu:

1. Dodawanie elementu na ostatnią pozycję

   Do nowej tablicy dodejemy element na ostatnią pozycję. Czasowa złożoność obliczeniowa tego algorytmu to O(n) – liniowa zależna od ilości elementów w tablicy.

```
-----------------------------------------------------
Size: 2000, cache: 200, operation: adding element on last position, result(microseconds): 5626.43
Size: 4000, cache: 200, operation: adding element on last position, result(microseconds): 7744.91
Size: 8000, cache: 200, operation: adding element on last position, result(microseconds): 9943.79
Size: 16000, cache: 200, operation: adding element on last position, result(microseconds): 14281.19
Size: 32000, cache: 200, operation: adding element on last position, result(microseconds): 23056.39
Size: 64000, cache: 200, operation: adding element on last position, result(microseconds): 41075.85
Size: 128000, cache: 200, operation: adding element on last position, result(microseconds): 75885.89
-----------------------------------------------------
```

| L.p. | Ilość elementów | Uśredniony czas 100 pomiarów [µs] |
|---|---|---|
| 1 | 2000 | 0.281322 |
| 2 | 4000 | 0.387246 |
| 3 | 8000 | 0.49719 |
| 4 | 16000 | 0.71406 |
| 5 | 32000 | 1.15282 |
| 6 | 64000 | 2.053793 |
| 7 | 128000 | 3.792795 |

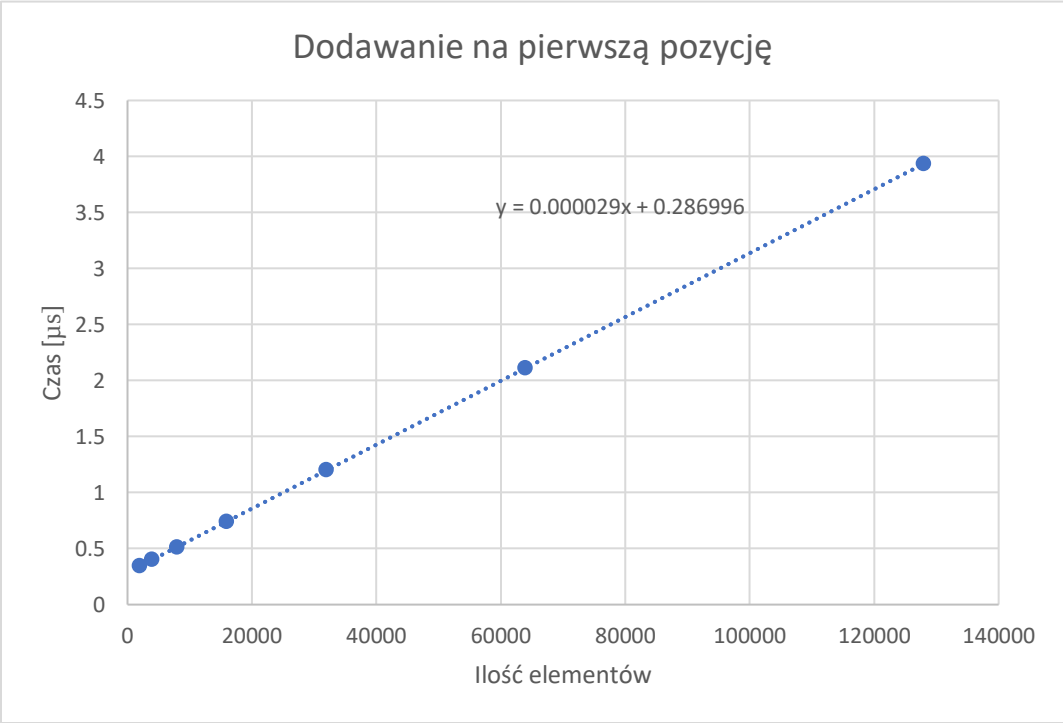


2. Dodawanie elementu na początek tablicy

Do nowej tablicy dodejemy element na pierwszą pozycję. Czasowa złożoność obliczeniowa tego algorytmu to O(n) – liniowa zależna od ilości elementów w tablicy.

```
-----------------------------------------------------
Size: 2000, cache: 200, operation: adding element on first position, result(microseconds): 6915.55
Size: 4000, cache: 200, operation: adding element on first position, result(microseconds): 8005.03
Size: 8000, cache: 200, operation: adding element on first position, result(microseconds): 10273.61
Size: 16000, cache: 200, operation: adding element on first position, result(microseconds): 14831.05
Size: 32000, cache: 200, operation: adding element on first position, result(microseconds): 24035.72
Size: 64000, cache: 200, operation: adding element on first position, result(microseconds): 42195.16
Size: 128000, cache: 200, operation: adding element on first position, result(microseconds): 78704.02
-----------------------------------------------------
```

| L.p. | Ilość elementów | Uśredniony czas 100 pomiarów [µs] |
|---|---|---|
| 1 | 2000 | 0.345778 |
| 2 | 4000 | 0.400252 |
| 3 | 8000 | 0.513681 |
| 4 | 16000 | 0.741553 |
| 5 | 32000 | 1.201786 |
| 6 | 64000 | 2.109758 |
| 7 | 128000 | 3.935201 |

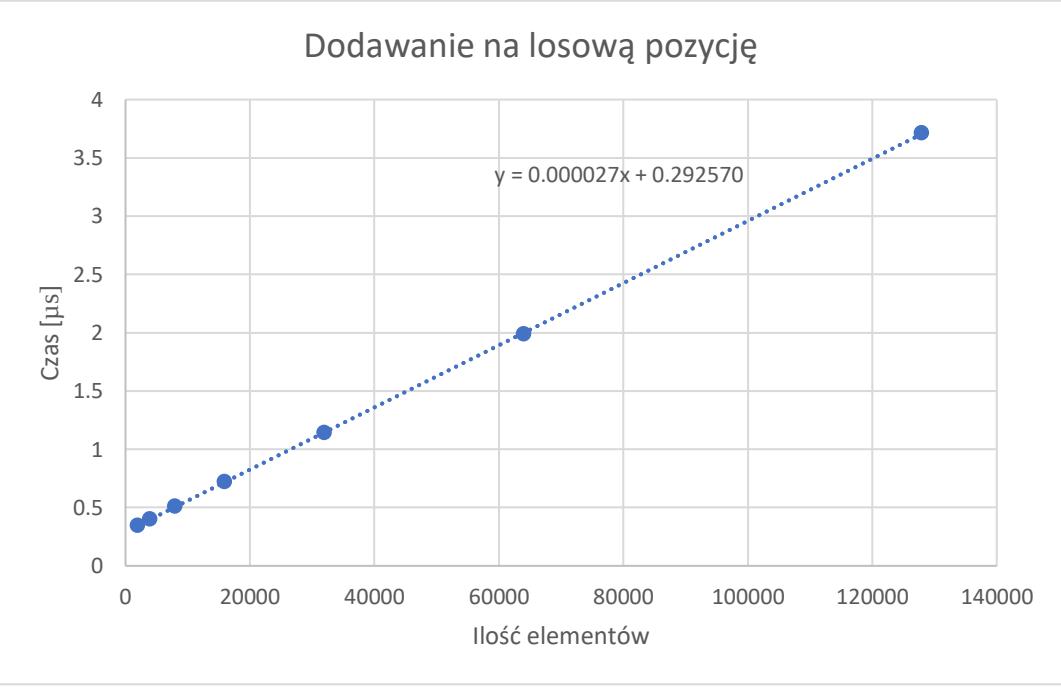


3. Dodawanie elementu na losową pozycję

Tworymy nowy obszar pamięci, wpisujemy część tablicy do miejsca do którego ma być zapisany nowy element, wpisujemy nowy element, zapisujemy pozostałą cześć tablicy. Czasowa złożoność obliczeniowa tego algorytmu to O(n) – liniowa zależna od ilości elementów w tablicy.

```
-------------------------------------------------------
Size: 2000, cache: 200, operation: adding element on random position, result(microseconds): 6925.55
Size: 4000, cache: 200, operation: adding element on random position, result(microseconds): 8044.9
Size: 8000, cache: 200, operation: adding element on random position, result(microseconds): 10203.65
Size: 16000, cache: 200, operation: adding element on random position, result(microseconds): 14411.21
Size: 32000, cache: 200, operation: adding element on random position, result(microseconds): 22846.39
Size: 64000, cache: 200, operation: adding element on random position, result(microseconds): 39746.63
Size: 128000, cache: 200, operation: adding element on random position, result(microseconds): 74236.69
-------------------------------------------------------
```

| L.p. | Ilość elementów | Uśredniony czas 100 pomiarów [µs] |
|---|---|---|
| 1 | 2000 | 0.346278 |
| 2 | 4000 | 0.402245 |
| 3 | 8000 | 0.510183 |
| 4 | 16000 | 0.720561 |
| 5 | 32000 | 1.14232 |
| 6 | 64000 | 1.987332 |
| 7 | 128000 | 3.711835 |



## Usuwanie elementu

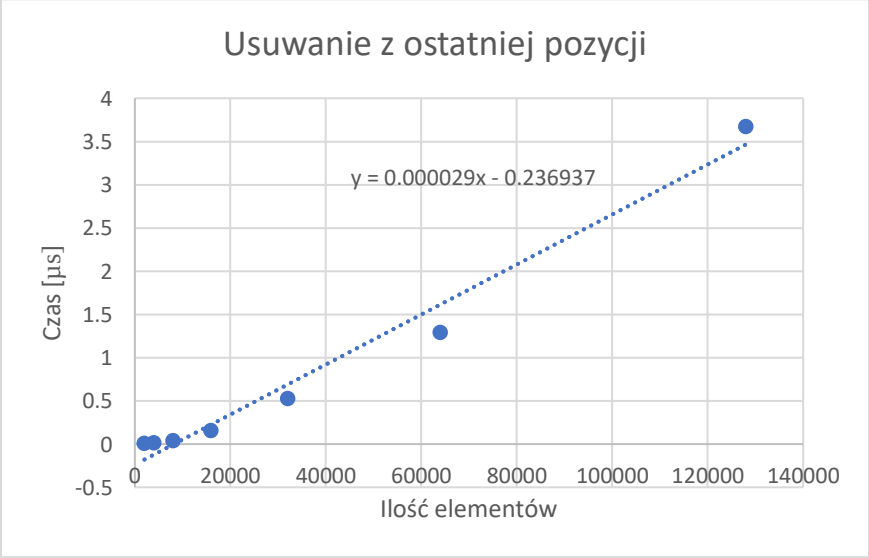
Podobno jak w dodawaniu alokujemy nowy obszar pamięci dla nowej tablicy tym razem o rozmiarze *n-1* dla tablicy o rozmiarze *n*. Generalnie działamy na tych samych zasadach. Mamy 7 różnych punktów pomiarowych (2000, 4000, 8000, 16000, 32000, 64000, 128000 elementów) w każdym z nich usuwamy 200 elementów 100 razy. Rezultaty testu dzielimy na 20000 aby uzyskać czas usunięcia 1 elementu:

1. Usuwanie elementu z ostatniej pozycji

   Usuwamy element z ostatniej pozycji danej tablicy i przenosimy pozostałą część tablicy w nowy obszar pamięci. Czasowa złożoność obliczeniowa tego algorytmu to O(n) – liniowa zależna od ilości elementów w tablicy.

```
-------------------------------------------------------
Size: 2000, cache: 200, operation: removing last element, result(microseconds): 59.97
Size: 4000, cache: 200, operation: removing last element, result(microseconds): 189.76
Size: 8000, cache: 200, operation: removing last element, result(microseconds): 789.37
Size: 16000, cache: 200, operation: removing last element, result(microseconds): 3067.89
Size: 32000, cache: 200, operation: removing last element, result(microseconds): 10513.51
Size: 64000, cache: 200, operation: removing last element, result(microseconds): 25784.7
Size: 128000, cache: 200, operation: removing last element, result(microseconds): 73357.06
-------------------------------------------------------
```

| L.p. | Ilość elementów | Uśredniony czas 100 pomiarów [µs] |
|---|---|---|
| 1 | 2000 | 0.002999 |
| 2 | 4000 | 0.009488 |
| 3 | 8000 | 0.039469 |
| 4 | 16000 | 0.153395 |
| 5 | 32000 | 0.525676 |
| 6 | 64000 | 1.289235 |
| 7 | 128000 | 3.667853 |

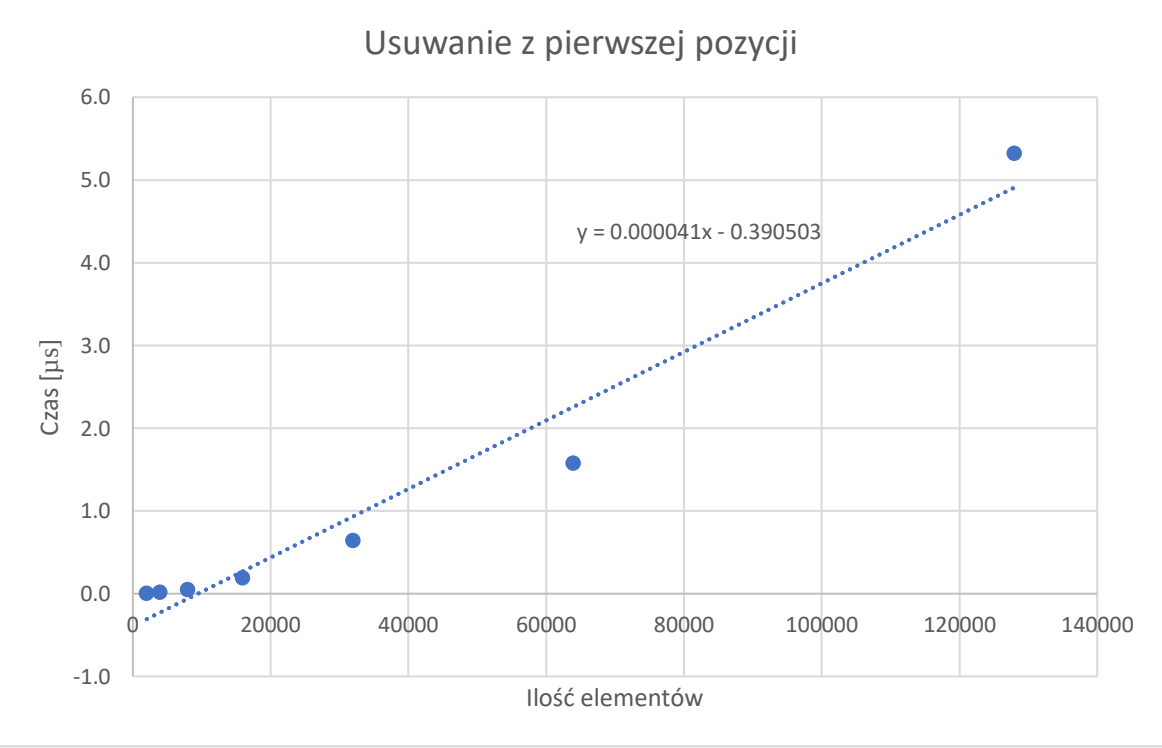


2. Usuwanie elementu z pierwszej pozycji

Usuwamy element z pierwszej pozycji i przenosimy pozostałą część tablicy w nowy obszar pamięci. Czasowa złożoność obliczeniowa tego algorytmu to O(n) – liniowa zależna od ilości elementów w tablicy.

```
-----------------------------------------------------------
Size: 2000, cache: 200, operation: removing first element, result(microseconds): 59.93
Size: 4000, cache: 200, operation: removing first element, result(microseconds): 249.78
Size: 8000, cache: 200, operation: removing first element, result(microseconds): 939.28
Size: 16000, cache: 200, operation: removing first element, result(microseconds): 3757.67
Size: 32000, cache: 200, operation: removing first element, result(microseconds): 12862.11
Size: 64000, cache: 200, operation: removing first element, result(microseconds): 31451.26
Size: 128000, cache: 200, operation: removing first element, result(microseconds): 106338.14
-----------------------------------------------------------
```

| L.p. | Ilość elementów | Uśredniony czas 100 pomiarów [µs] |
|---|---|---|
| 1 | 2000 | 0.002997 |
| 2 | 4000 | 0.012489 |
| 3 | 8000 | 0.046964 |
| 4 | 16000 | 0.187884 |
| 5 | 32000 | 0.643106 |
| 6 | 64000 | 1.572563 |
| 7 | 128000 | 5.316909 |

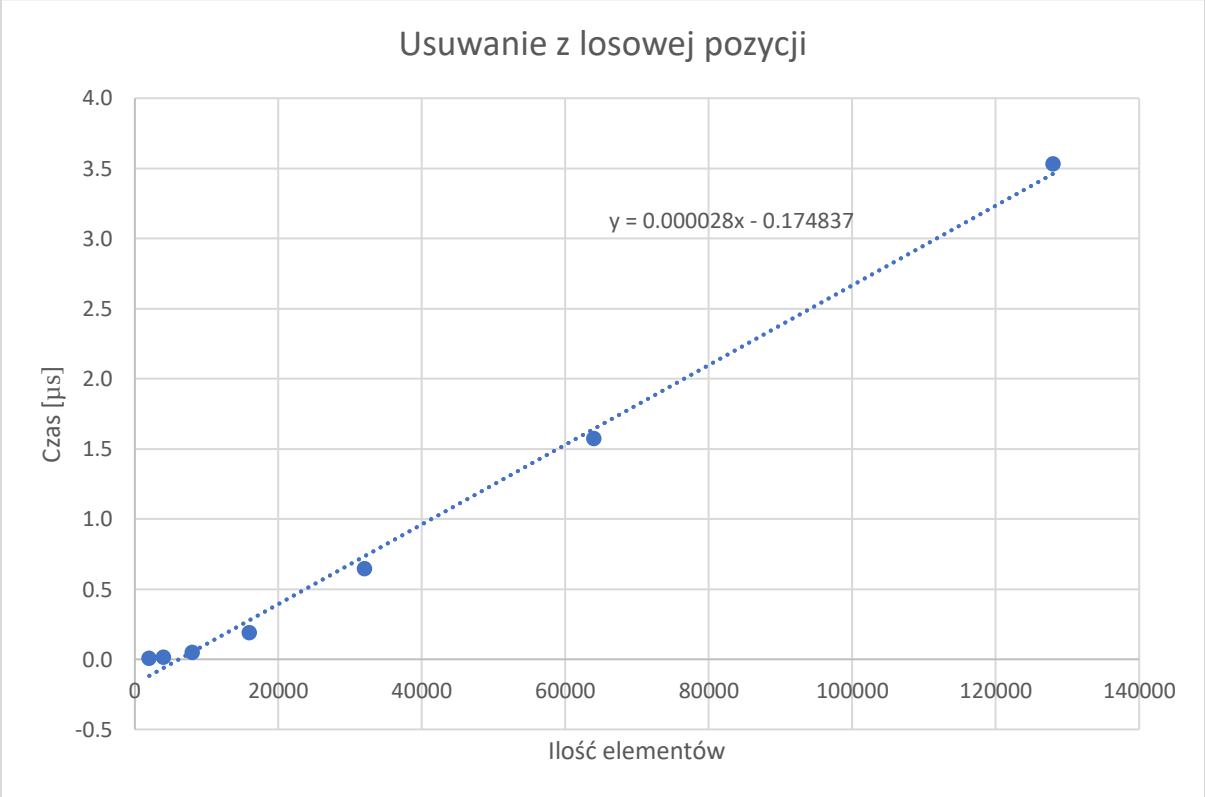


1. Usuwanie elementu z losowej pozycji

Prepisujemy cześć tablicy do nowego obszaru, ignoryjemy element który chcemy usunąć i kopijujemy pozostałą część tablicy. Czasowa złożoność obliczeniowa tego algorytmu to O(n) – liniowa zależna od ilości elementów w tablicy.

```
-----------------------------------------------------------
Size: 2000, cache: 200, operation: removing random element, result(microseconds): 59.94
Size: 4000, cache: 200, operation: removing random element, result(microseconds): 239.78
Size: 8000, cache: 200, operation: removing random element, result(microseconds): 939.49
Size: 16000, cache: 200, operation: removing random element, result(microseconds): 3757.47
Size: 32000, cache: 200, operation: removing random element, result(microseconds): 12842.05
Size: 64000, cache: 200, operation: removing random element, result(microseconds): 31441.43
Size: 128000, cache: 200, operation: removing random element, result(microseconds): 70548.85
-----------------------------------------------------------
```

| L.p. | Ilość elementów | Uśredniony czas 100 pomiarów [µs] |
|---|---|---|
| 1 | 2000 | 0.002997 |
| 2 | 4000 | 0.011989 |
| 3 | 8000 | 0.046975 |
| 4 | 16000 | 0.187874 |
| 5 | 32000 | 0.642103 |
| 6 | 64000 | 1.572072 |
| 7 | 128000 | 3.527443 |

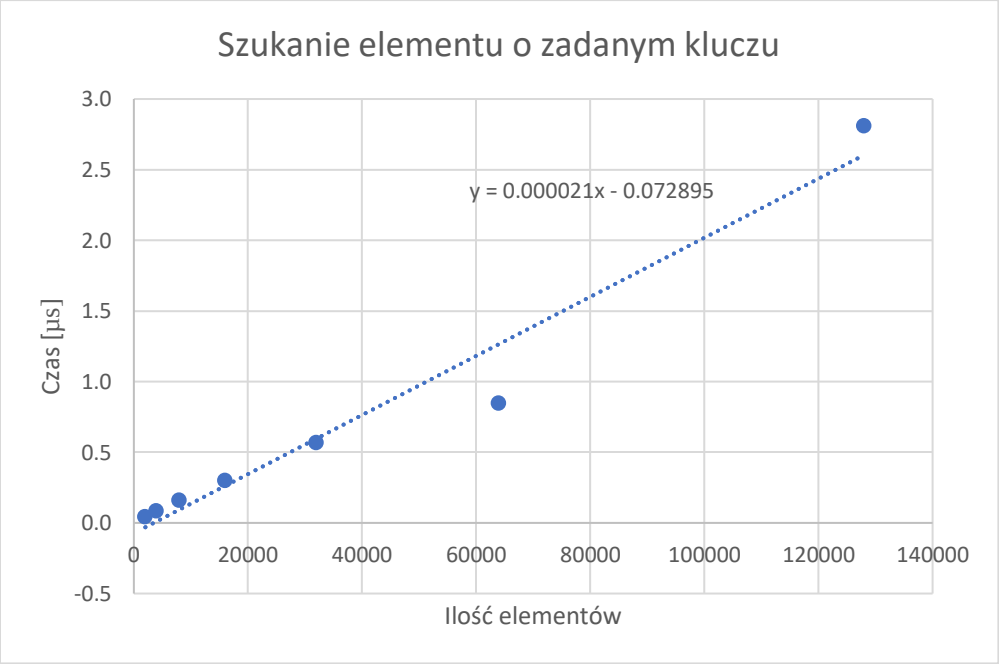


## Szukanie elementu o zdananym kluczu

Algorytm w tym punkcie polega na przeszukiwaniu tabeli od początkowego indeksu i sprawdzaniu czy wartość pod tym indeksem zgodna jest z zadanym kluczem. Ten program zwraca -1 jeżeli wartość nie została znaleziona albo numer indeksu pierwszego pojawienia tego elementu w tablice. Punkty pomiarowe zostawiamy te same: n = 2000, 4000, 8000, 16000, 32000, 64000, 128000. Czasowa złożoność tego algorytmu jest linowa O(n).

```
-------------------------------------------------------
Size: 2000, cache: 200, operation: searching for element, result(microseconds): 849.16
Size: 4000, cache: 200, operation: searching for element, result(microseconds): 1628.55
Size: 8000, cache: 200, operation: searching for element, result(microseconds): 3137.64
Size: 16000, cache: 200, operation: searching for element, result(microseconds): 5965.97
Size: 32000, cache: 200, operation: searching for element, result(microseconds): 11332.99
Size: 64000, cache: 200, operation: searching for element, result(microseconds): 16929.96
Size: 128000, cache: 200, operation: searching for element, result(microseconds): 56167.06
-------------------------------------------------------
```

| L.p. | Ilość elementów | Uśredniony czas 100 pomiarów [µs] |
|---|---|---|
| 1 | 2000 | 0.042458 |
| 2 | 4000 | 0.081428 |
| 3 | 8000 | 0.156882 |
| 4 | 16000 | 0.298299 |
| 5 | 32000 | 0.56665 |
| 6 | 64000 | 0.846498 |
| 7 | 128000 | 2.808353 |



## Wnioski

Każda operacja przebigła w sposób spodziwany. Każda z nich ma złożoność obliczniową O(n), co wyraźnie widać na wykresach. Generalnie tablica – jest jedna z najprostrzych struktur danych. Podstawowe działania na nij się wykonują dość wolno w porównaniu z innymi strukturami danych. Ale ze względu na pamięć jest ona dość efektywna, bo w każdym momencie zajmuje jak najmniej miejsca oraz każdy element nie potrzebuje przechowywania dodatkowych danych jak na przykład adres do nagłówka następnego elementu w listach. To robi tablicę dość przydatną w wielu przypadkach, jak n.p. w wymienionych wyżej.

Dynamiczna alokacja pamięci w przypadku struktur tablicowych jest istotna w prawidłowym działaniu programu. Nie korzystanie z niej skutkuje niestabilnością programu, który po wykonaniu wielu operacji może zakończyć swoje działanie w wyniku błędu pamięci. Jednak korzystanie z relokacji po każdym usunięciu/dodaniu elementu wydłuża czas operacji, a dodatkowo uniemożliwia zaobserwowanie rzeczywistej efektywności danego algorytmu.

# Lista dwukierunkowa

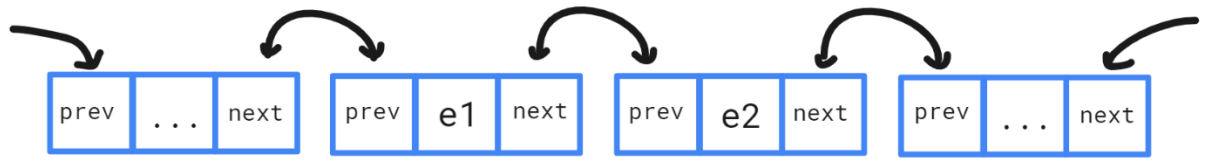
Lista lub sekwencja jest abstrakcyjnym typem danych, który reprezentuje policzalną liczbę uporządkowanych wartości, gdzie ta sama wartość może wystąpić więcej niż jeden raz. Instancja listy jest komputerową reprezentacją matematycznego pojęcia krotki lub skończonego ciągu. Nieskończonym analogiem listy jest strumień. Listy są podstawowym przykładem kontenerów, ponieważ zawierają inne wartości. Jeżeli ta sama wartość występuje wielokrotnie, każde zdarzenie jest traktowane jako odrębny element.

Każdy element przechowuje dane (w tym przypadku liczba 4-bajtowe ze znakiem) i wskaznik do nagłówka następnego elementu. Ale w tym przypadku rozpatrujemy listę dwukierunkową, co oznacza że też przechowujemy wskaznik do nagłówka poprzedniego elementu. Jeżeli poprzedniego (element jest głową listy) lub następnego (element jest ogonem listy) elementu nie instnieje to wskaznik wynosi NULL.

Zastosowania listy dwukierunkowej:

- Lista dwukierunkowa może być stosowana w systemach nawigacyjnych, w których wymagana jest zarówno nawigacja z przodu, jak i z tyłu.
- Jest ona używana przez przeglądarki do implementacji nawigacji do tyłu i do przodu odwiedzanych stron internetowych.
- Jest również używana przez różne aplikacje do implementacji funkcji undo i redo.
- Może być używana do reprezentowania talii kart w grach.

Przykład:




Tablica złożoności poszczególnych operacji na liscie dwukierunkowej:

| Operacja | Średnia złożoność | Pesymistyczna złożoność |
|---|---|---|
| Dostęp (nie sprawdzamy) | O(n) | O(n) |
| Dodawanie elementu | O(1) | O(1) |
| Usuwanie elementu | O(1) | O(1) |
| Poszuk elementu | O(n) | O(n) |

## Dodawanie elementu

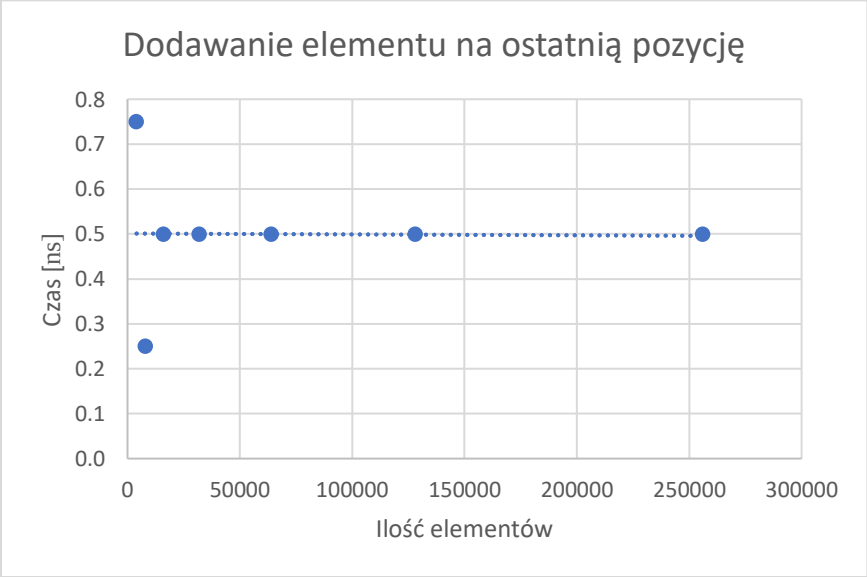
Jest to dośc szybka operacja jeżeli mamy do czynienia z listą chociaż czasowa złożoność będzie się zmieniała w zależnośći od tego na którą pozycję wstawiamy element, bo nie każda z nich nie zależy od ilości danych wejściowych. Z tego powodu, że oczekuje szybzejsze działanie algorytmów, trochę zmianiamy punkty obserwacji. Teraz działamy na listach o pomiarach: 4000, 8000, 16000, 32000, 64000, 128000, 256000 elementów. Za każdym razem dodajemy 400 elementów. Powtarzamy operację 100 razy i uśredniamy wynik.

1. Dodawanie elementu na ostatnią pozycję

   Aby dodać element na koniec listy należy stworzyć obiekt lub strukturę 1 elementu listy i ustawić jej wskaźnik elementu następnego na NULL, a elementu poprzednigo na ten, które aktualnie jest ogonem. Wskaznik „poprzedniego” ogona następnego elementu ustawiamy na „nowy” ogon. Czasowa złożoność tego algorytmu jest stała O(1), gdyż nie zależy od ilości danych wejściowych.

```
-----------------------------------------------------
Size: 4000, cache: 400, operation: adding element on last position, result(nanoseconds): 29984
Size: 8000, cache: 400, operation: adding element on last position, result(nanoseconds): 9980
Size: 16000, cache: 400, operation: adding element on last position, result(nanoseconds): 19987
Size: 32000, cache: 400, operation: adding element on last position, result(nanoseconds): 19986
Size: 64000, cache: 400, operation: adding element on last position, result(nanoseconds): 19986
Size: 128000, cache: 400, operation: adding element on last position, result(nanoseconds): 19992
Size: 256000, cache: 400, operation: adding element on last position, result(nanoseconds): 19982
-----------------------------------------------------
```

| L.p. | Ilość elementów | Uśredniony czas 100 pomiarów [ns] |
|---|---|---|
| 1 | 4000 | 0.7496 |
| 2 | 8000 | 0.2495 |
| 3 | 16000 | 0.499675 |
| 4 | 32000 | 0.49965 |
| 5 | 64000 | 0.49965 |
| 6 | 128000 | 0.4998 |
| 7 | 256000 | 0.49955 |

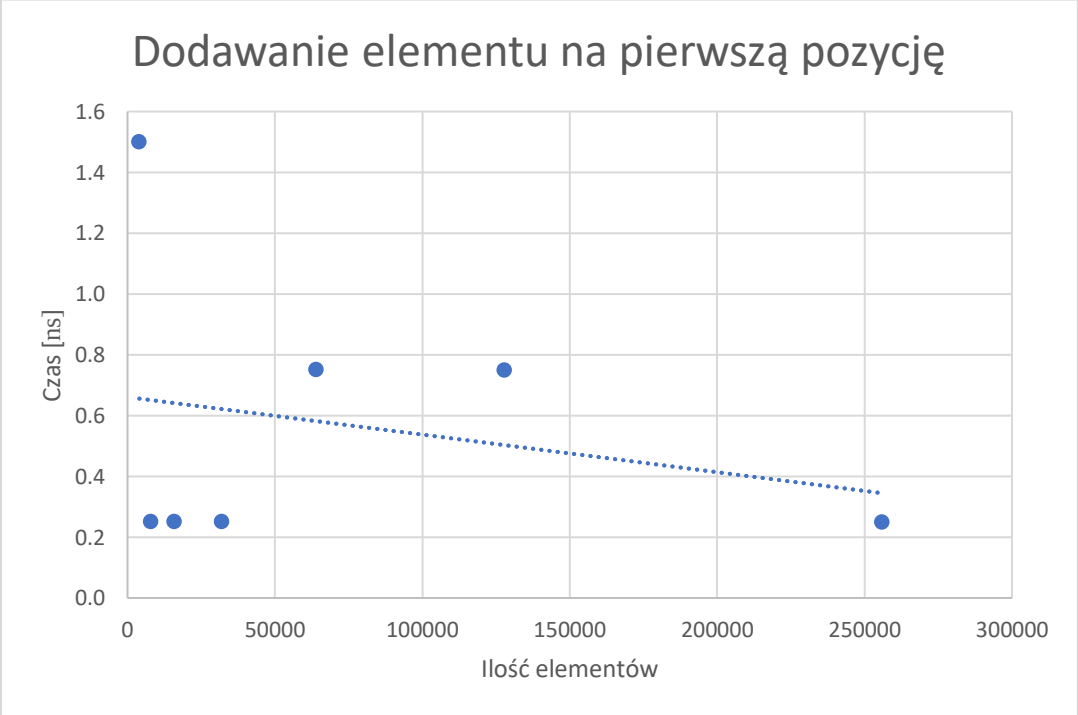


2. Dodawanie elementu na pierwszą pozycję

Aby dodać element na początek listy należy stworzyć obiekt lub strukturę 1 elementu listy i ustawić jej wskaźnik elementu poprzedzającego na NULL, a elementu następnego na ten, które aktualnie jest głową. Wskaznik „poprzedniej" głowy poprzedniego elementu ustawiamy na „nową" głowe. Czasowa złożoność tego algorytmu jest stała O(1), gdyż nie zależy od ilości danych wejściowych.

```
-------------------------------------------------------
Size: 4000, cache: 400, operation: adding element on first position, result(nanoseconds): 59978
Size: 8000, cache: 400, operation: adding element on first position, result(nanoseconds): 9995
Size: 16000, cache: 400, operation: adding element on first position, result(nanoseconds): 9994
Size: 32000, cache: 400, operation: adding element on first position, result(nanoseconds): 9995
Size: 64000, cache: 400, operation: adding element on first position, result(nanoseconds): 30011
Size: 128000, cache: 400, operation: adding element on first position, result(nanoseconds): 29968
Size: 256000, cache: 400, operation: adding element on first position, result(nanoseconds): 9979
-------------------------------------------------------
```

| L.p. | Ilość elementów | Uśredniony czas 100 pomiarów [ns] |
|---|---|---|
| 1 | 4000 | 1.49945 |
| 2 | 8000 | 0.249875 |
| 3 | 16000 | 0.24985 |
| 4 | 32000 | 0.249875 |
| 5 | 64000 | 0.750275 |
| 6 | 128000 | 0.7492 |
| 7 | 256000 | 0.249475 |

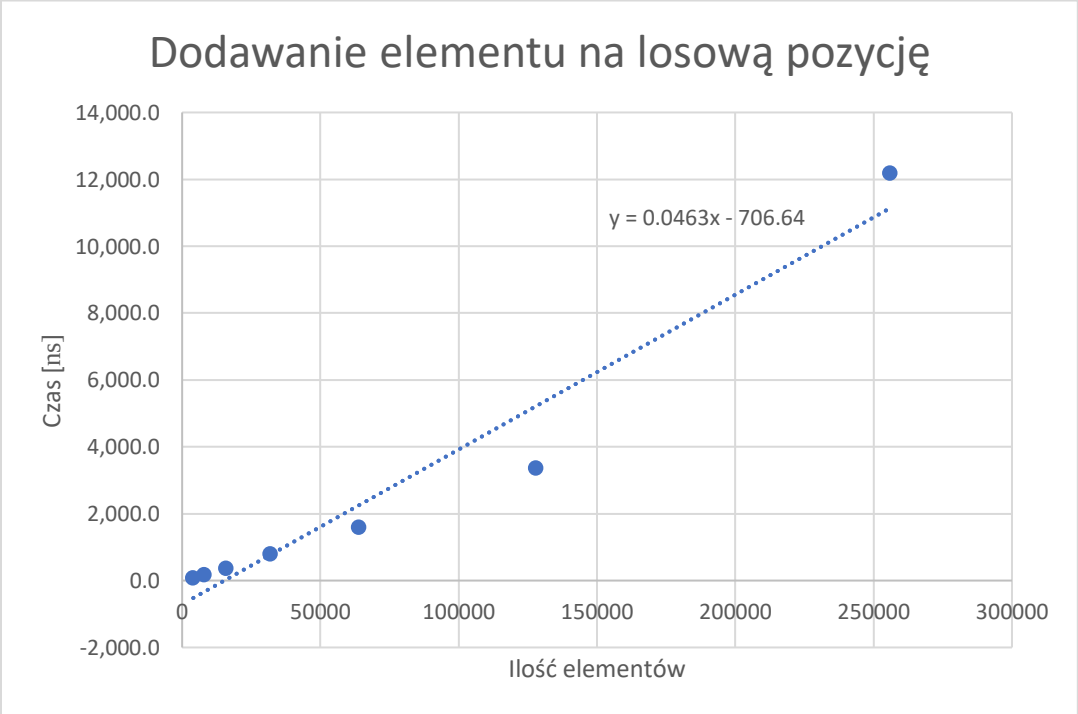


3. Dodawanie elementu na losową pozycję

Aby dodać element na losową pozycje listy należy stworzyć obiekt lub strukturę 1 elementu listy i w sposób iteracyjny znaleźć miejsce w które ma być zapisany nowy element i zmienić wskazniki. Czasowa złożoność tego algorytmu jest stała O(n), gdyż zależy od ilości danych wejściowych.

```
-----------------------------------------------------
Size: 4000, cache: 400, operation: adding element on random position, result(nanoseconds): 3359093
Size: 8000, cache: 400, operation: adding element on random position, result(nanoseconds): 6765910
Size: 16000, cache: 400, operation: adding element on random position, result(nanoseconds): 14801736
Size: 32000, cache: 400, operation: adding element on random position, result(nanoseconds): 31751796
Size: 64000, cache: 400, operation: adding element on random position, result(nanoseconds): 63933403
Size: 128000, cache: 400, operation: adding element on random position, result(nanoseconds): 134672884
Size: 256000, cache: 400, operation: adding element on random position, result(nanoseconds): 487280533
-----------------------------------------------------
```

| L.p. | Ilość elementów | Uśredniony czas 100 pomiarów [ns] |
|---|---|---|
| 1 | 4000 | 83.97733 |
| 2 | 8000 | 169.1478 |
| 3 | 16000 | 370.0434 |
| 4 | 32000 | 793.7949 |
| 5 | 64000 | 1598.335 |
| 6 | 128000 | 3366.822 |
| 7 | 256000 | 12182.01 |



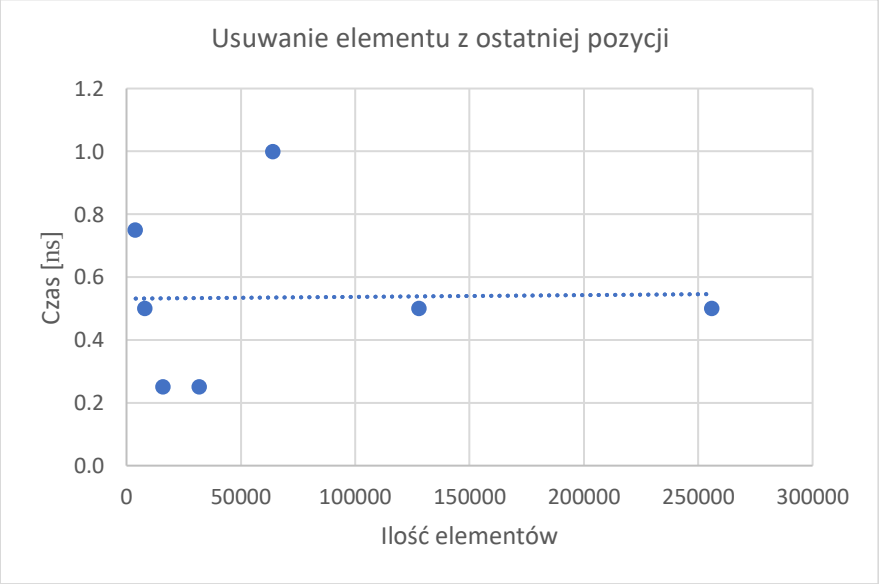
## Usuwanie elementu

Usuwanie elementu nie bardzo się różni od dodawania elementu do listy ze względu na złożoność obliczeniową. Punkty obserwacji zostawiamy takie same: 4000, 8000, 16000, 32000, 64000, 128000, 256000 elementów. Za każdym razem dodajemy 400 elementów. Powtarzamy operację 100 razy i uśredniamy wynik.

1. Usuwanie elementu z ostatniej pozycji
   Aby usunąć ostatni element, mamy się zwrócić do ogona listy. Jeżeli istnieje poprzedni element (ogon nie jest głową) to robimy go teraz ogonem i „poprzedni" ogon usuwamy. Wskaznik następnego elementu „nowego" ogona ustawiamy na NULL. . Czasowa złożoność tego algorytmu jest stała O(1), gdyż nie zależy od ilości danych wejściowych.

```
-------------------------------------------------------
Size: 4000, cache: 400, operation: removing last element, result(nanoseconds): 29982
Size: 8000, cache: 400, operation: removing last element, result(nanoseconds): 20007
Size: 16000, cache: 400, operation: removing last element, result(nanoseconds): 9993
Size: 32000, cache: 400, operation: removing last element, result(nanoseconds): 9995
Size: 64000, cache: 400, operation: removing last element, result(nanoseconds): 39974
Size: 128000, cache: 400, operation: removing last element, result(nanoseconds): 19985
Size: 256000, cache: 400, operation: removing last element, result(nanoseconds): 19992
-------------------------------------------------------
```

| L.p. | Ilość elementów | Uśredniony czas 100 pomiarów [ns] |
|---|---|---|
| 1 | 4000 | 0.74955 |
| 2 | 8000 | 0.500175 |
| 3 | 16000 | 0.249825 |
| 4 | 32000 | 0.249875 |
| 5 | 64000 | 0.99935 |
| 6 | 128000 | 0.499625 |
| 7 | 256000 | 0.4998 |

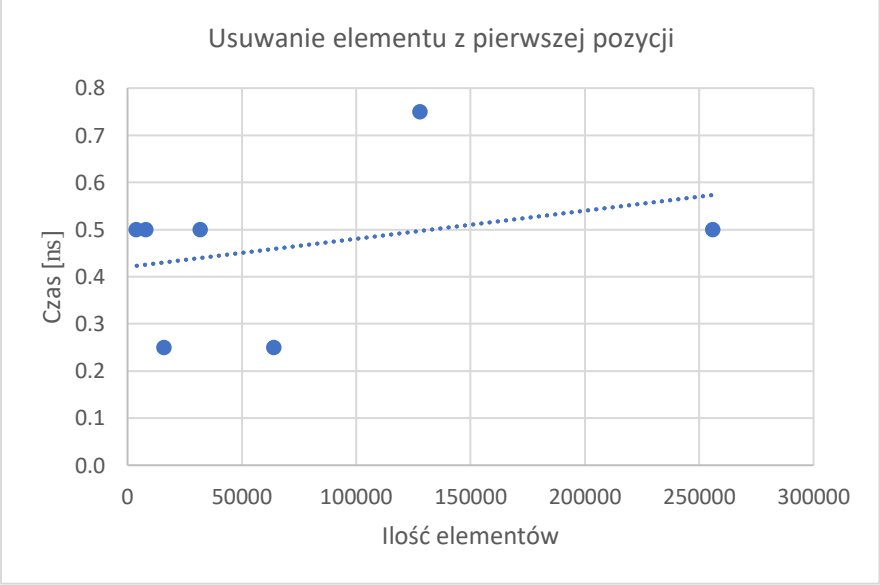


2. Usuwanie elementu z pierwszej pozycji

Aby usunąć pierwszy element, mamy się zwrócić do głowy listy. Jeżeli istnieje następny element element (głowa nie jest ogonem) to robimy ją traz głową i „poprzednią” głowę usuwamy. Wskaznik poprzedniego elementu „nowej” głowy ustawiamy na NULL. Czasowa złożoność tego algorytmu jest stała O(1), gdyż nie zależy od ilości danych wejściowych.

```
-----------------------------------------------------
Size: 4000, cache: 400, operation: removing first element, result(nanoseconds): 19986
Size: 8000, cache: 400, operation: removing first element, result(nanoseconds): 19988
Size: 16000, cache: 400, operation: removing first element, result(nanoseconds): 9992
Size: 32000, cache: 400, operation: removing first element, result(nanoseconds): 19989
Size: 64000, cache: 400, operation: removing first element, result(nanoseconds): 9994
Size: 128000, cache: 400, operation: removing first element, result(nanoseconds): 29978
Size: 256000, cache: 400, operation: removing first element, result(nanoseconds): 19984
-----------------------------------------------------
```

| L.p. | Ilość elementów | Uśredniony czas 100 pomiarów [ns] |
|---|---|---|
| 1 | 4000 | 0.49965 |
| 2 | 8000 | 0.4997 |
| 3 | 16000 | 0.2498 |
| 4 | 32000 | 0.499725 |
| 5 | 64000 | 0.24985 |
| 6 | 128000 | 0.74945 |
| 7 | 256000 | 0.4996 |

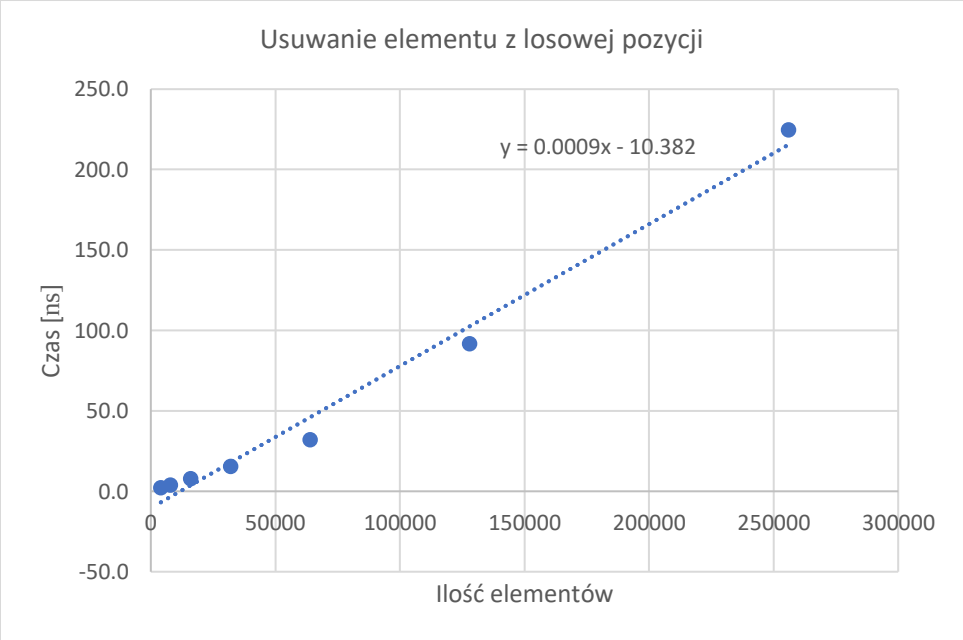


3. Usuwanie elementu z losowej pozycji

Aby usunąć element z losowej pozyj trzeba dotrzeć do niej w sposób iteracyjny, zmienić nagłówki i usunąc niepotrzebny element. Czasowa złożoność tego algorytmu jest stała O(n), gdyż zależy od ilości danych wejściowych.

```
-------------------------------------------------------
Size: 4000, cache: 400, operation: removing random element, result(nanoseconds): 69959
Size: 8000, cache: 400, operation: removing random element, result(nanoseconds): 139919
Size: 16000, cache: 400, operation: removing random element, result(nanoseconds): 299835
Size: 32000, cache: 400, operation: removing random element, result(nanoseconds): 609659
Size: 64000, cache: 400, operation: removing random element, result(nanoseconds): 1269273
Size: 128000, cache: 400, operation: removing random element, result(nanoseconds): 3657883
Size: 256000, cache: 400, operation: removing random element, result(nanoseconds): 8975001
-------------------------------------------------------
```

| L.p. | Ilość elementów | Uśredniony czas 100 pomiarów [ns] |
|---|---|---|
| 1 | 4000 | 1.748975 |
| 2 | 8000 | 3.497975 |
| 3 | 16000 | 7.495875 |
| 4 | 32000 | 15.24148 |
| 5 | 64000 | 31.73183 |
| 6 | 128000 | 91.44708 |
| 7 | 256000 | 224.375 |

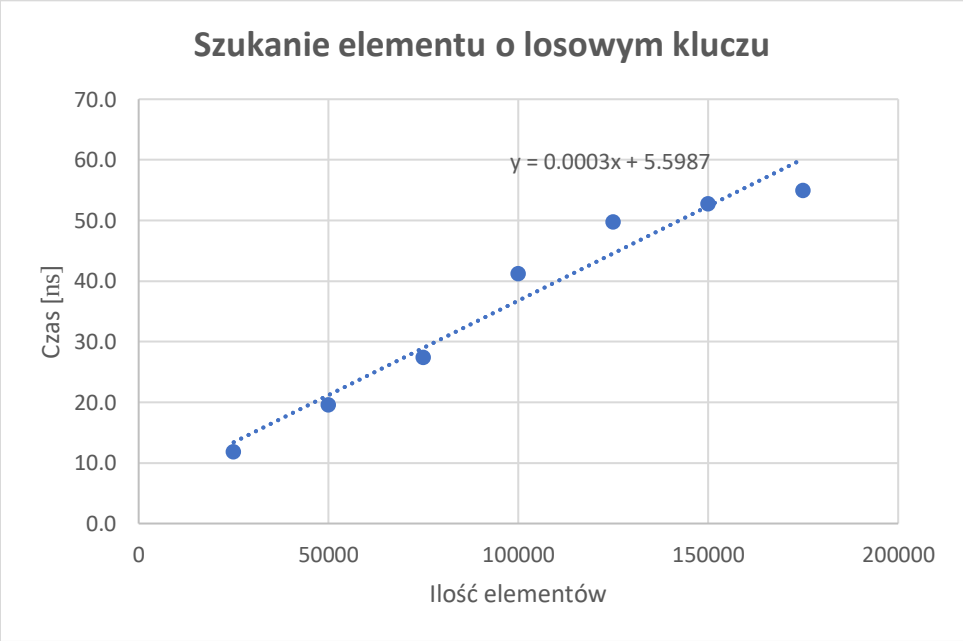


## Szukanie elementu o losowym kluczu

Szukanie elementu o zadanym kluczu ogranicza się tylko do przeszukiwania listy(skakania po wskaźnikach) i sprawdzania czy wartość danego elementu równa jest kluczowi. Jeśli tak funkcja zwraca true, jeśli nie znaleziona żadnego elementu spełniającego równanie i sprawdzono całą listę zwraca false. W tej operacji zmieniam punkty obserwacji, ustawiając ich na 25000, 50000, 75000, 100000, 150000, 175000, 200000 elementów. Ilość szukanych elementów: 500, powtarzamy operację 100 razy i uśredniamy wynik. Czasowa złożoność tego algorytmu jest linowa O(n), gdyż zależy od ilości danych wejściowych.

```
-----------------------------------------------------
Size: 25000, cache: 500, operation: searching for element, result(nanoseconds): 589821
Size: 50000, cache: 500, operation: searching for element, result(nanoseconds): 979265
Size: 75000, cache: 500, operation: searching for element, result(nanoseconds): 1369391
Size: 100000, cache: 500, operation: searching for element, result(nanoseconds): 2058661
Size: 125000, cache: 500, operation: searching for element, result(nanoseconds): 2488773
Size: 150000, cache: 500, operation: searching for element, result(nanoseconds): 2638493
Size: 175000, cache: 500, operation: searching for element, result(nanoseconds): 2748247
-----------------------------------------------------
```

| L.p. | Ilość elementów | Uśredniony czas 100 pomiarów [ns] |
|---|---|---|
| 1 | 25000 | 11.79642 |
| 2 | 50000 | 19.5853 |
| 3 | 75000 | 27.38782 |
| 4 | 100000 | 41.17322 |
| 5 | 125000 | 49.77546 |
| 6 | 150000 | 52.76986 |
| 7 | 175000 | 54.96494 |



## Wnioski

Wykresy dla dodawania i usuwania liczb do listy z końca i początku są zbliżone do stałych. Pojedyncze pomiary zaburzają statystykę, i dlatego linia trendu jest liniowa. Błąd ten spowodowany jest niedokładnością funkcji mierzącej czas, jak i różną chwilową hierarchizacją wątków w procesorze.

Czasami punkty na płaszczyźnie wyglądają jako porabola, ale to z powodu specyfiki dobierania punktów obserwowalnych. W przypadku 2000, 4000, 8000, 16000... elementów tworzą oni kwadratową zależność. Z tego powodu jeżeli punkty tworzą parabole to świadczy o złożoności obliczeniowej O(n).

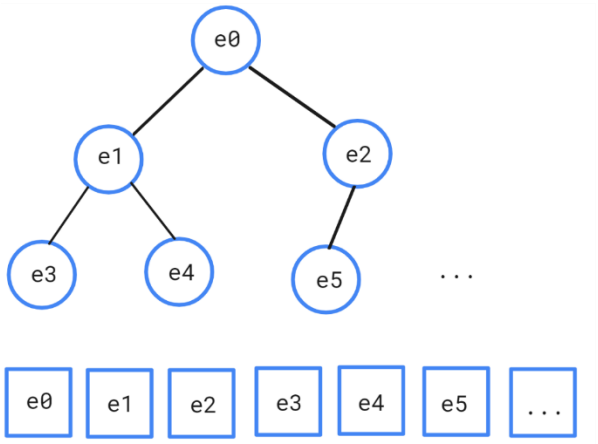
## Kopiec

**Kopiec** – struktura danych w kształcie drzewa, w której drzewo jest pełnym drzewem binarnym. Nie jest ADT. Jeśli kopiec ma być kopcem zupełnym to mają być spełnione jeszcze dwa dodatkowe warunki: liście w drzewie występują w ostatnim i ewentualnie przedostatnim poziomie w drzewie oraz liście na ostatnim poziomie są spójnie ułożone od strony lewej do prawej. Aby kopiec działał sprawnie, wykorzystamy logarytmiczną naturę drzewa binarnego do reprezentowania naszego kopcu. Aby zagwarantować wydajność logarytmiczną, musimy zachować równowagę naszego drzewa. Zrównoważone drzewo binarne ma mniej więcej taką samą liczbę węzłów w lewym i prawym poddrzewiu korzenia. Pełne drzewo binarne to drzewo, w którym każdy poziom ma wszystkie swoje węzły. Wyjątkiem jest dolny poziom drzewa, który wypełniamy od lewej do prawej. Metoda, której użyjemy do przechowywania elementów w kopce, opiera się na utrzymaniu właściwości zamówienia kopca. Właściwość kolejności kopca jest następująca: w stercie dla każdego węzła *x* z nadrzędnym *p* klucz w *p* jest mniejszy lub równy kluczowi w *x*.

Zastosowanie kpców:

- System operacyjny może używać kopców do określenia następnego procesu.
- Sortowanie przez kopcowanie używa kopców aby sortować tablice w czasie O(log(n))
- Kolejka priorytetowa: kolejki priorytetowe mogą być skutecznie zaimplementowane za pomocąkopca binarnego, ponieważ obsługuje operacje insert (), delete () i extractmax (), decreaseKey () w czasie O(log(n)). Stos dwumianowy i stos Fibonacciego są odmianami stosu binarnego. Zmiany te wykonują połączenie również skutecznie.

Przykład:




Tablica złożoności obliczeniowej poszczególnych operacjina kopcu:

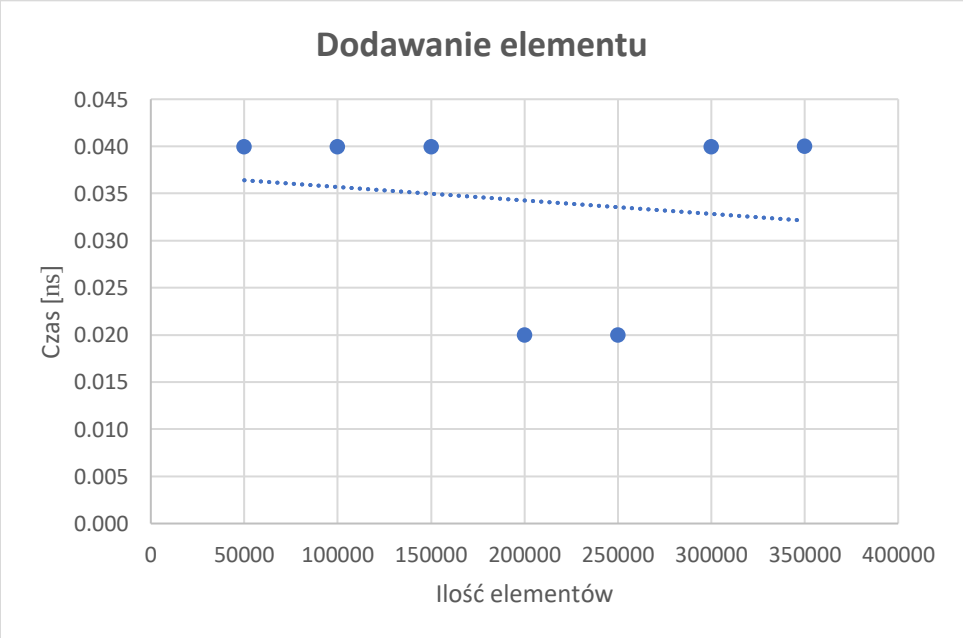
| Operacja | Średnia złożoność | Pesymistyczna złożoność |
|---|---|---|
| Dostęp (nie sprawdzamy) | O(1) | O(1) |
| Dodawanie elementu | O(1) | O(log n) |
| Usuwanie elementu | O(log n) | O(log n) |
| Poszuk elementu | O(n) | O(n) |

## Dodawanie elementu

Aby dodać element do kopca należy dodać element do na koniec kopca i następnie sprawdzić czy spełniony jest warunek kopca(w naszym eksperymencie: czy jest mniejszy lub równy rodzicowi). Jeśli warunek kopca nie jest spełniony należy zamienić ze sobą nasz element(syna) z jego rodzicem. Następnie należy sprawdzić warunek poziom wyżej i wykonywać zamiany dopóki jest to konieczne. Kopiec jest uporządkowany, gdy dotarliśmy do korzenia, lub gdy jedno z kolejnych porównań w algorytmie zwróci nam wynik, że zachowany jest warunek kopca. Zmianiamy punkty obserwacki na 50000, 100000, 150000, 200000, 250000, 300000, 350000 elementów. Dodajemy 5000 elementów. Powtarzamy operację 100 razy, wynik uśredniamy. Złożoność obliczeniowa takiej operacji to O(1) lub O(log n) w zależnośći od przypadku.

```
-----------------------------------------------------
Size: 50000, cache: 5000, operation: adding element on random position, result(nanoseconds): 19986
Size: 100000, cache: 5000, operation: adding element on random position, result(nanoseconds): 19987
Size: 150000, cache: 5000, operation: adding element on random position, result(nanoseconds): 19988
Size: 200000, cache: 5000, operation: adding element on random position, result(nanoseconds): 9994
Size: 250000, cache: 5000, operation: adding element on random position, result(nanoseconds): 9997
Size: 300000, cache: 5000, operation: adding element on random position, result(nanoseconds): 19985
Size: 350000, cache: 5000, operation: adding element on random position, result(nanoseconds): 19991
-----------------------------------------------------
```

| L.p. | Ilość elementów | Uśredniony czas 100 pomiarów [ns] |
|---|---|---|
| 1 | 50000 | 0.039972 |
| 2 | 100000 | 0.039974 |
| 3 | 150000 | 0.039976 |
| 4 | 200000 | 0.019988 |
| 5 | 250000 | 0.019994 |
| 6 | 300000 | 0.03997 |
| 7 | 350000 | 0.039982 |

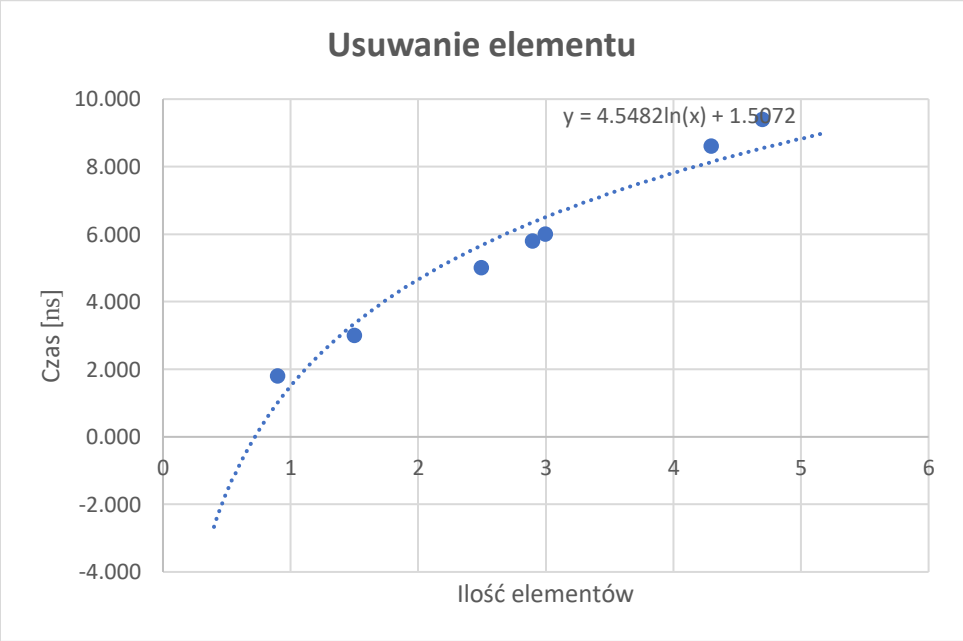


## Usuwanie elementu

Aby usunąć korzeń należy przenieść ostatni element kopca(o ile istnieje) w miejsce korzenia, a stary korzeń usunąć. Następnie trzeba sprawdzić warunek kopca(czy korzeń jest większy od obu synów. Jeśli warunek kopca nie jest spełniony należy zamienić korzeń z synem o większej wartości, a następnie powtarzać sprawdzanie i ewentualne zamiany, aż kopiec będzie uporządkowany. Kopiec jest uporządkowany, gdy dojdziemy do liścia, lub gdy jedno z kolejnych porównań zwróci nam wynik, że warunek kopca jest spełniony(rodzic niemniejszy od synów). Zmianiamy punkty obserwacji. Złożoność obliczeniowa takiej operacji to O(log n) nizależnie od przypadku.

```
-----------------------------------------------------
Size: 25000, cache: 1000, operation: removing random element, result(nanoseconds): 89945
Size: 50000, cache: 1000, operation: removing random element, result(nanoseconds): 149908
Size: 75000, cache: 1000, operation: removing random element, result(nanoseconds): 249721
Size: 100000, cache: 1000, operation: removing random element, result(nanoseconds): 289819
Size: 125000, cache: 1000, operation: removing random element, result(nanoseconds): 299835
Size: 150000, cache: 1000, operation: removing random element, result(nanoseconds): 469724
Size: 175000, cache: 1000, operation: removing random element, result(nanoseconds): 429746
-----------------------------------------------------
```

| L.p. | Ilość elementów | Uśredniony czas 100 pomiarów [ns] |
|---|---|---|
| 1 | 25000 | 0.89945 |
| 2 | 50000 | 1.49908 |
| 3 | 75000 | 2.49721 |
| 4 | 100000 | 2.89819 |
| 5 | 125000 | 2.99835 |
| 6 | 150000 | 4.69724 |
| 7 | 175000 | 4.29746 |

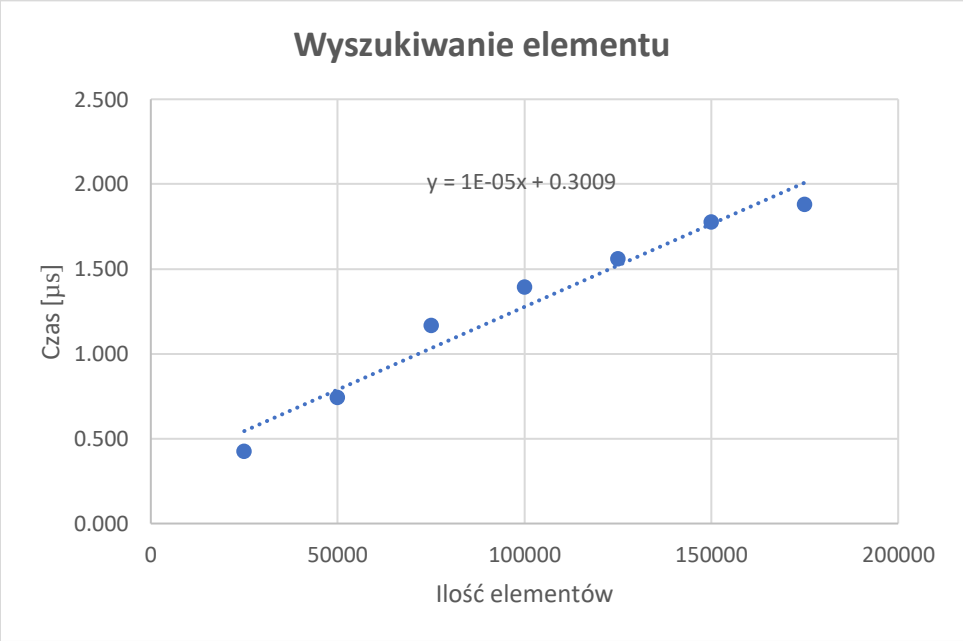


## Wyszukiwanie elementu

Wyszukiwanie elementu w lopce binarnym jest takie same jak w tablicy. Nie zmieniamy punktu obserwacji. Złożoność obliczeniowa tej operacji jest O(n) bo jest to generalnie iteracja w tablice.

```
-------------------------------------------------------
Size: 25000, cache: 1000, operation: searching for element, result(microseconds): 42275.57
Size: 50000, cache: 1000, operation: searching for element, result(microseconds): 74147.37
Size: 75000, cache: 1000, operation: searching for element, result(microseconds): 116582.54
Size: 100000, cache: 1000, operation: searching for element, result(microseconds): 139279.65
Size: 125000, cache: 1000, operation: searching for element, result(microseconds): 155989.97
Size: 150000, cache: 1000, operation: searching for element, result(microseconds): 177647.32
Size: 175000, cache: 1000, operation: searching for element, result(microseconds): 187851.56
-------------------------------------------------------
```

| L.p. | Ilość elementów | Uśredniony czas 100 pomiarów [ns] |
|---|---|---|
| 1 | 25000 | 0.422756 |
| 2 | 50000 | 0.741474 |
| 3 | 75000 | 1.165825 |
| 4 | 100000 | 1.392797 |
| 5 | 125000 | 1.5599 |
| 6 | 150000 | 1.776473 |
| 7 | 175000 | 1.878516 |



## Wnioski

Kopiec jest znacznie szybszą strukturą danych niż poprzednie, ale nie tak szybka jak drzewo binarne wyszukiwań. Z tego powodu zminiałem punkty obserwacji, aby nie było zerowych wyników.

Ostatnia operacja wygląda bardziej jak O(1) i trudno powiedzieć z jakiego to powodu. Być może dlatego że bezpośrednie przejście przez tablice ciężko jest zarejestrować, bo jest ono bardzo szybkie. W każdym razie implementacja tego algorytmu jest O(n) złożoności.

## Drzewo czerwono-czarne

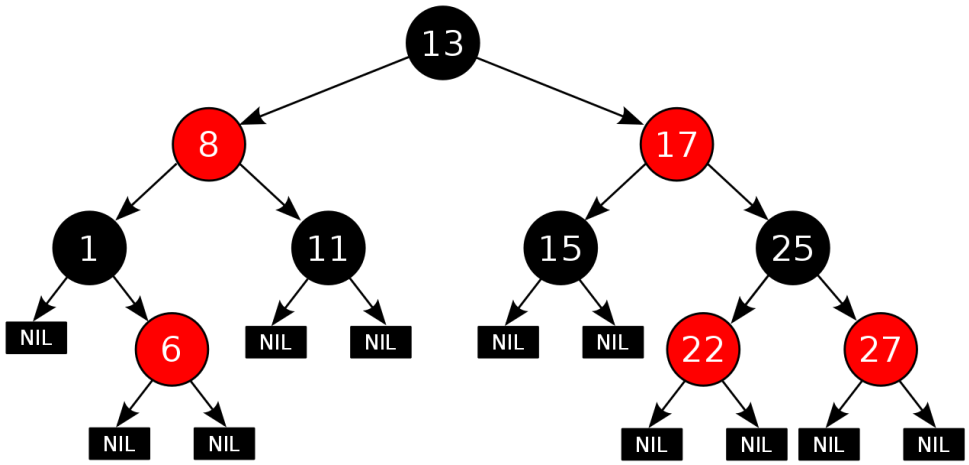
**Drzewo czerwono-czarne** jest samo-równoważącym się binarnym drzewem wyszukiwania, w którym każdy węzeł zawiera dodatkowy bit do oznaczenia koloru węzła, czerwony lub czarny. Każde drzewo czerwono-czarne spełnia poniższe kryteria :

- Każdy węzeł drzewa jest albo czerwony, albo czarny.
- Każdy liść drzewa (węzeł pusty null) jest zawsze czarny
- Korzeń drzewa jest zawsze czarny
- Jeśli węzeł jest czerwony, to obaj jego synowie są czarni – innymi słowy, w drzewie nie mogą występować dwa kolejne czerwone węzły, ojciec i syn
- Każda prosta ścieżka od danego węzła do dowolnego z jego liści potomnych zawiera tę samą liczbę węzłów czarnych

Zastosowanie dzrew czerwono-czarnych:

- Linux używa drzewa RB w swojej obecnej implementacji.
- Drzewa czerwono-czrne są używane w większości bibliotek językowych, takich jak map, multimap, multiset w C++

Przykład:




Złożoność obliczeniowa operacji wykonanych na drzewie czerwono-carnym:

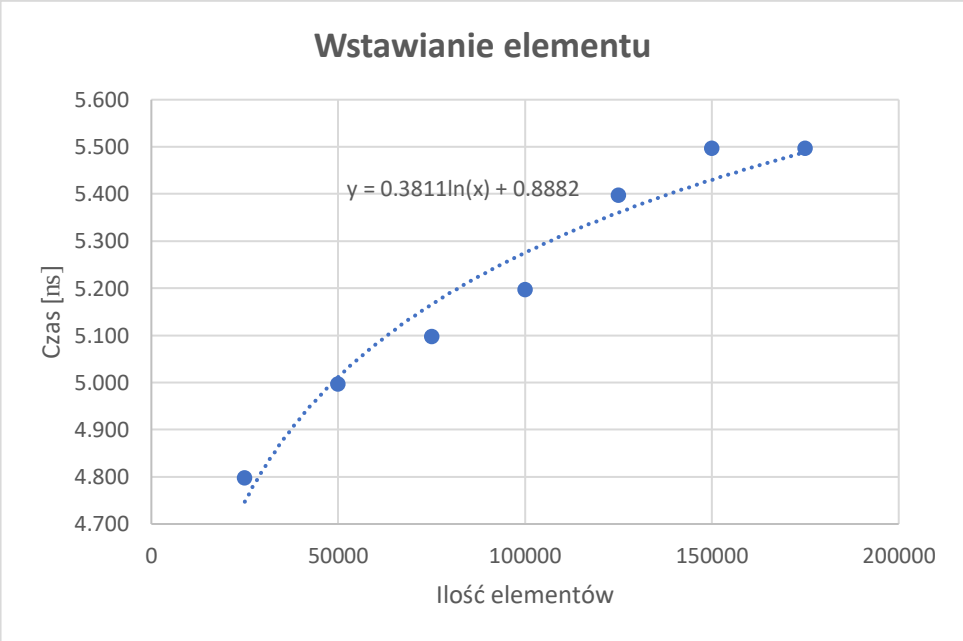
| Operacja | Średnia złożoność | Pesymistyczna złożoność |
|---|---|---|
| Dostęp (nie sprawdzamy) | O(log n) | O(log n) |
| Dodawanie elementu | O(log n) | O(log n) |
| Usuwanie elementu | O(log n) | O(log n) |
| Poszuk elementu | O(log n) | O(log n) |

## Dodawanie elementu

Wstawianie rozpoczyna się od dodwania węzła w bardzo podobny sposób jak w standardowym binarnym drzewie wyszukiwania i przez zabarwienie go na czerwono. Duża różnica polega na tym, że w binarnym drzewie wyszukiwania nowy węzeł jest dodawany jako liść, podczas gdy liście nie zawierają informacji w czerwono–czarnym drzewie, więc zamiast tego nowy węzeł zastępuje istniejący liść, a następnie ma dwa czarne liście własne dodane. Złożoność obliczeniowa danej operacji jest O(log n).

```
-----------------------------------------------------
Size: 25000, cache: 1000, operation: adding element on random position, result(nanoseconds): 479726
Size: 50000, cache: 1000, operation: adding element on random position, result(nanoseconds): 499713
Size: 75000, cache: 1000, operation: adding element on random position, result(nanoseconds): 509707
Size: 100000, cache: 1000, operation: adding element on random position, result(nanoseconds): 519698
Size: 125000, cache: 1000, operation: adding element on random position, result(nanoseconds): 539688
Size: 150000, cache: 1000, operation: adding element on random position, result(nanoseconds): 549691
Size: 175000, cache: 1000, operation: adding element on random position, result(nanoseconds): 549681
-----------------------------------------------------
```

| L.p. | Ilość elementów | Uśredniony czas 100 pomiarów [ns] |
|---|---|---|
| 1 | 25000 | 4.79726 |
| 2 | 50000 | 4.99713 |
| 3 | 75000 | 5.09707 |
| 4 | 100000 | 5.19698 |
| 5 | 125000 | 5.39688 |
| 6 | 150000 | 5.49691 |
| 7 | 175000 | 5.49681 |

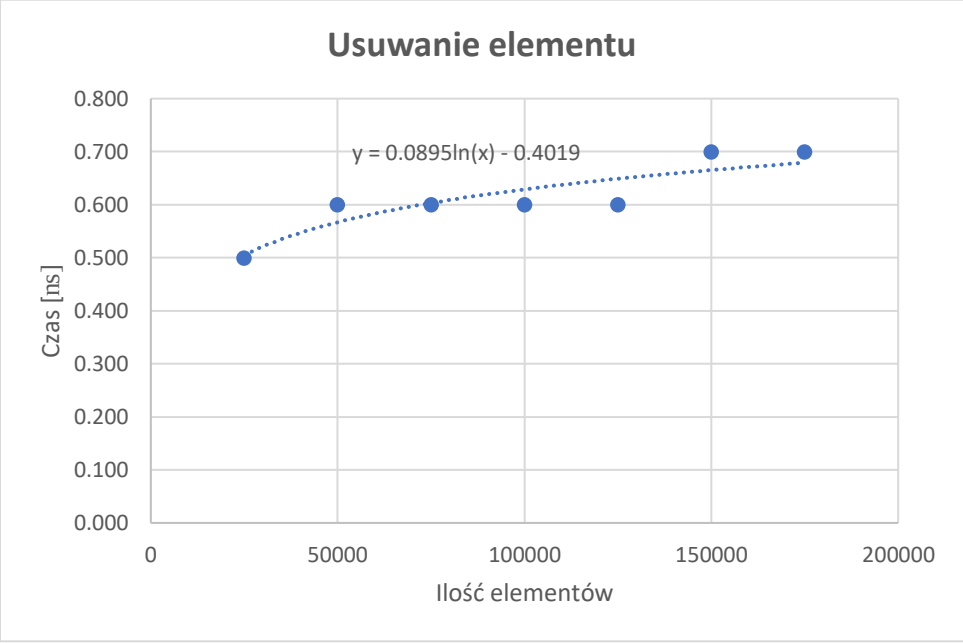


## Usuwanie elementu

Usuwanie elementu z drzewa binarnego jest dość skomplikowane. Wymaga rozpatrzenia sześciu różnych przypadków. Tutaj ten algorytm nie będzie omowiany. Punktów obserwacji nie zmieniamy. Złożoność obliczeniowa danej operacji jest O(log n).

```
-----------------------------------------------------
Size: 25000, cache: 1000, operation: removing random element, result(nanoseconds): 49964
Size: 50000, cache: 1000, operation: removing random element, result(nanoseconds): 59961
Size: 75000, cache: 1000, operation: removing random element, result(nanoseconds): 59960
Size: 100000, cache: 1000, operation: removing random element, result(nanoseconds): 59962
Size: 125000, cache: 1000, operation: removing random element, result(nanoseconds): 59960
Size: 150000, cache: 1000, operation: removing random element, result(nanoseconds): 69958
Size: 175000, cache: 1000, operation: removing random element, result(nanoseconds): 69966
-----------------------------------------------------
```

| L.p. | Ilość elementów | Uśredniony czas 100 pomiarów [ns] |
|---|---|---|
| 1 | 25000 | 0.49964 |
| 2 | 50000 | 0.59961 |
| 3 | 75000 | 0.5996 |
| 4 | 100000 | 0.59962 |
| 5 | 125000 | 0.5996 |
| 6 | 150000 | 0.69958 |
| 7 | 175000 | 0.69966 |

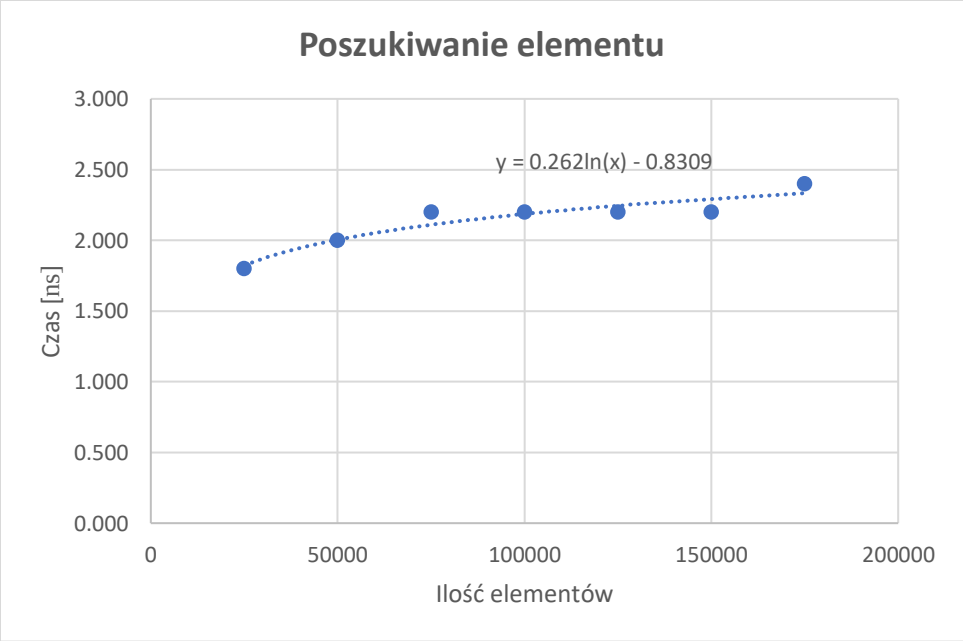


## Poszukiwanie elementu

Poszukiwanie lementu jest takie same jak w dzewie binarnym poszukiwań. Przechodzimy po drzewie w celu sprawdzenia czy występuje tam żądana wartość. Punktów obserwacji nie zmieniamy. Złożoność obliczeniowa danej operacji jest O(log n).

```
-----------------------------------------------------
Size: 25000, cache: 1000, operation: searching for element, result(nanoseconds): 89944
Size: 50000, cache: 1000, operation: searching for element, result(nanoseconds): 99942
Size: 75000, cache: 1000, operation: searching for element, result(nanoseconds): 109935
Size: 100000, cache: 1000, operation: searching for element, result(nanoseconds): 109947
Size: 125000, cache: 1000, operation: searching for element, result(nanoseconds): 109934
Size: 150000, cache: 1000, operation: searching for element, result(nanoseconds): 109939
Size: 175000, cache: 1000, operation: searching for element, result(nanoseconds): 119921
-----------------------------------------------------
```

| L.p. | Ilość elementów | Uśredniony czas 100 pomiarów [ns] |
|---|---|---|
| 1 | 25000 | 1.79888 |
| 2 | 50000 | 1.99884 |
| 3 | 75000 | 2.1987 |
| 4 | 100000 | 2.19894 |
| 5 | 125000 | 2.19868 |
| 6 | 150000 | 2.19878 |
| 7 | 175000 | 2.39842 |



## Wnioski

Alternatywnym sposobem równoważenia BST jest użycie drzew AVL. AVL jest prostsze w implementacji i daje bardziej zrównoważone drzewo, lecz z tego powodu operacje dodawania bądź usuwania są bardziej kosztowne; w najgorszym przypadku będzie konieczne przejście całej ścieżki od liścia do korzenia, podczas gdy przywrócenie własności czerwono-czarnych wymaga wykonania maksymalnie dwóch rotacji. Ale nie zostało ono zaimplementowane w tym projekcie.

# Literatura


http://jaroslaw.mierzwa.staff.iiar.pwr.wroc.pl/

https://pl.wikipedia.org/wiki/Drzewo_czerwono-czarne

https://www.bigocheatsheet.com/