Sprawozdanie Struktury danych i złożoność obliczeniowa

Projekt 1

Imię i Nazwisko	Michał Kaźmierczak
Numer indeksu	263924
Termin zajęć	Wtorek, 15^{15}-16^{55} TN
Prowadzący	Mgr. inż. Antoni Sterna

Wprowadzenie

W ramach projektu 1 zadaniem było zaimplementować następujące struktury danych: listę, tablicę, kopiec oraz drzewo BST. Celem projektu było nie tylko zrozumienie działania tych struktur, ale również zbadanie ich złożoności obliczeniowych dla poszczególnych operacji.

W ramach implementacji przeanalizowaliśmy trzy podstawowe operacje dla każdej z tych struktur: push, pop oraz find dla listy; insertFront, insertBack, get, del oraz indexOf/find dla tablicy; oraz push, pop oraz search dla kopca.

Operacja push polega na dodaniu nowego elementu do struktury, operacja pop na usunięciu elementu z końca struktury, a operacja find/indexOf na znalezieniu elementu w strukturze. W przypadku listy, operacje push i pop są wykonywane w czasie stałym O(1), co oznacza, że czas wykonania nie zależy od ilości elementów w liście. Natomiast operacja find ma złożoność O(n), co oznacza, że czas wykonania rośnie liniowo wraz ze wzrostem ilości elementów.

W przypadku tablicy, operacje insertFront i insertBack mają złożoność O(n), co oznacza, że czas wykonania rośnie liniowo wraz ze wzrostem ilości elementów. Natomiast operacje get i indexOf/find są wykonywane w czasie stałym O(1). Operacja del ma złożoność O(n), co oznacza, że czas wykonania rośnie liniowo, ponieważ wymaga ona przesunięcia pozostałych elementów w tablicy.

W przypadku kopca, operacje push i pop mają złożoność O(log n), co oznacza, że czas wykonania rośnie logarytmicznie wraz ze wzrostem ilości elementów. Natomiast operacja search ma złożoność O(n).

W raporcie przedstawie szczegółowe opisy każdej z tych operacji oraz ich złożoności obliczeniowe dla poszczególnych struktur danych. W ten sposób, będzie można dokładnie poznać jak każda z tych struktur działa oraz jakie są jej mocne i słabe strony.

1. Lista dwukierunkowa

1.1 Wstęp

Implementacja listy dwukierunkowej. Lista ta posiada trzy pola prywatne - wskaźnik na pierwszy element, wskaźnik na ostatni element i pole przechowujące ilość elementów w liście. Klasa ta posiada kilka metod publicznych, które umożliwiają dodawanie elementów na początek, na koniec i na dowolną pozycję, usuwanie elementów z początku, końca oraz elementów o podanej wartości, a także dodawanie elementu przed i za wskazanym elementem, wypisywanie elementów listy od początku i od końca, znajdowanie elementu o podanej wartości i usuwanie wszystkich elementów z listy. Każdy element listy jest reprezentowany przez obiekt klasy Elem, który posiada trzy pola - wartość, wskaźnik na następny element i wskaźnik na poprzedni element.

Metoda printFront() wypisuje wartości elementów listy od początku, a metoda printBack() wypisuje wartości elementów listy od końca. Metody pushFront() i pushBack() służą do dodawania elementów na początek i na koniec listy. Metoda insertOnIndex() służy do dodawania elementu na wskazaną pozycję w liście. Metoda popFront() usuwa i zwraca pierwszy element listy, a metoda popBack() usuwa i zwraca ostatni element listy. Metoda findElementByValue() znajduje element o podanej wartości i zwraca wskaźnik na niego. Metody insertBefore() i insertAfter() służą do dodawania elementów przed i za wskazanym elementem. Metoda deleteItem() usuwa wszystkie elementy o podanej wartości. Metoda get() zwraca wskaźnik na element o podanym indeksie. Metoda clear() usuwa wszystkie elementy z listy.

Fragment implementacji w programie

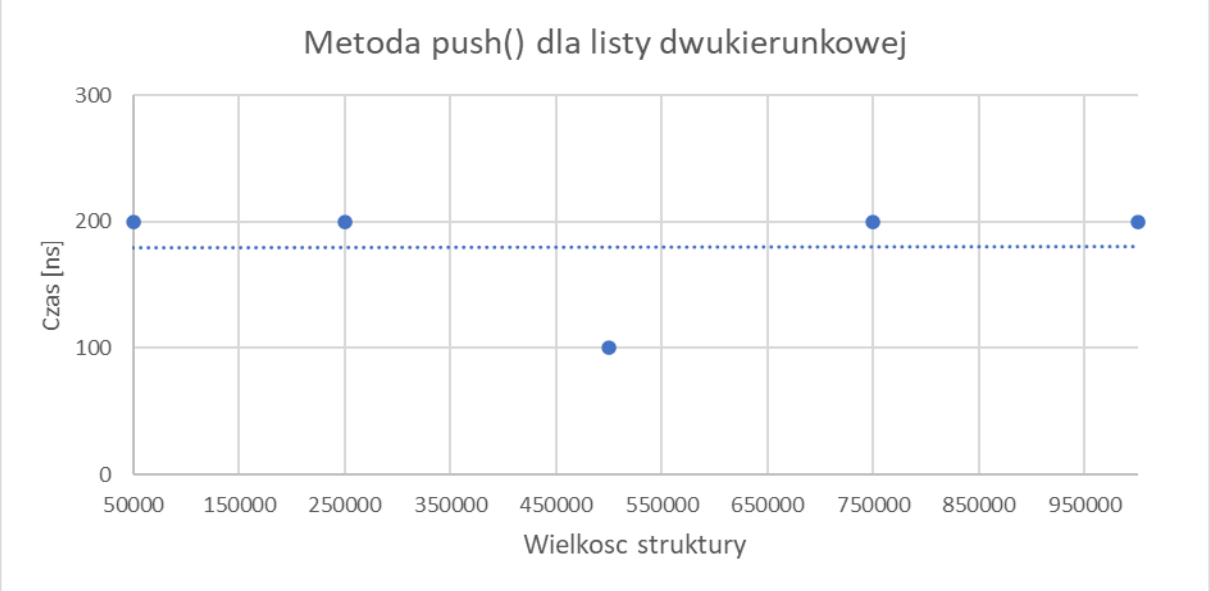
```
class Elem {
public:
    int data;   // - przechowuje wartość elementu
    Elem* next; // - wskazuje na następny element listy
    Elem* prev; // - wskazuje na poprzedni element listy

    explicit Elem(int value) {
        data = value;
        next = nullptr;
        prev = nullptr;
    }
};

class List {

private:
    Elem *head; //  - wskazuje na pierwszy element listy
    Elem *tail; // - wskazuje na ostatni element listy
    int size; // - przechowuje ilość elementów w liście

public:
    List() {
        head = nullptr;
        tail = nullptr;
        size = 0;
    }
```

1.2 Złożoności obliczeniowe metod listy

1.2.1 PushFront() oraz PushBack()

Operacja dodania elementu na początek listy oraz na jej koniec wymaga tylko jednego przypisania referencji do nowego węzła, więc jej złożoność obliczeniowa wynosi **O(1)**. Zgadza się to z wykresem dla testów struktury wraz z zwiększającą się ilością elementów.

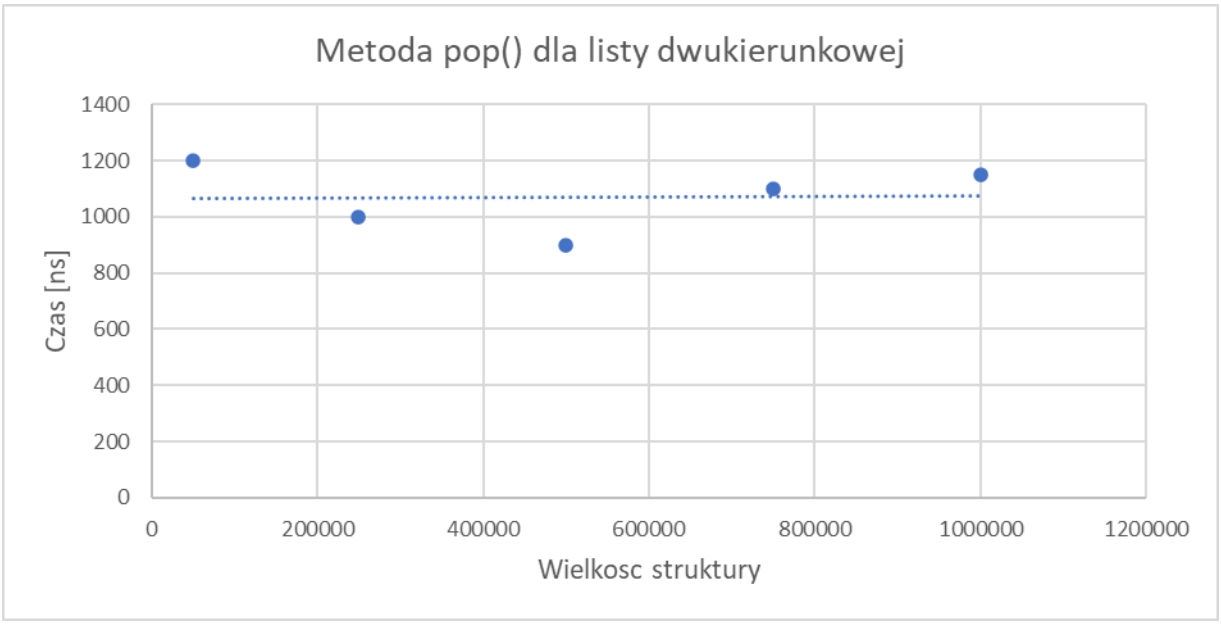
Wielkość struktury	Czas [ns]
50000	200
250000	200
500000	100
750000	200
1000000	200



1.2.2 PopFront() oraz PopBack()

Lista przechowuje wskaźnik na pierwszy oraz ostatni element, więc usunięcie ostatniego elementu ma złożoność obliczeniową **O(1)** - stałą, ponieważ wystarczy przepiąć wskaźnik na odpowiednio drugi lub przedostatni element jako odpowiednio pierwszy lub ostatni ostatni i zwolnić pamięć zajmowaną przez usuwany element. Potwierdzają to rezultaty testów, które jasno pokazują złożoność **O(1)**, z niewielkimi rozbieżnościami.

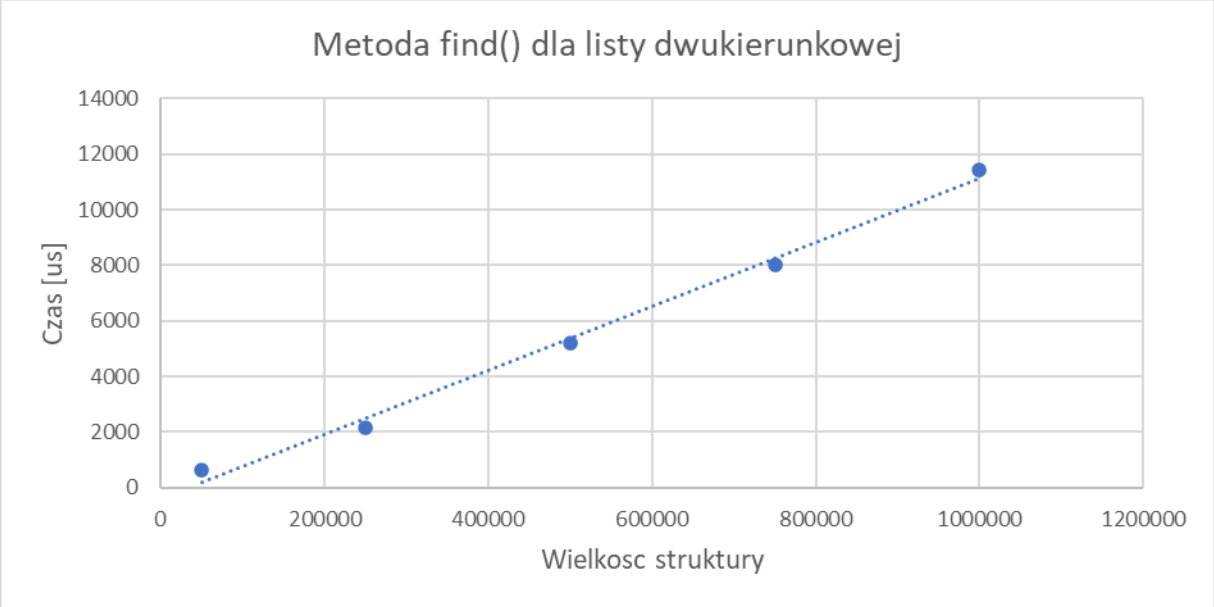
Wielkość struktury	Czas [ns]
50000	1200
250000	1000
500000	900
750000	1100
1000000	1150



1.2.3 Find()

W przypadku wyszukiwania elementu po wartości, złożoność obliczeniowa metody find wynosi **O(n)**, gdzie n to liczba elementów w liście. Dlatego też, jeśli lista jest bardzo duża, to czas wyszukiwania może być długi. Jest to przeszukiwanie liniowe. Dla zaimplementowanej struktury złożoność zgadza się z tą założoną i wynosi ona również **O(n).**

Wielkość struktury	Czas [us]
50000	620,3
250000	2172,1
500000	5203,1
750000	8035,7
1000000	11439



2. Tablica

2.1. Wstęp

Klasa Table, reprezentuje tablicę dynamiczną. Klasa ta zawiera prywatne pola: size (rozmiar tablicy) oraz wskaźnik *table na początek tablicy. Konstruktor bezargumentowy tworzy pustą tablicę o zerowym rozmiarze. Metoda insertBack(), dodająca nowy element na koniec tablicy. Metoda insertFront() dodaje wartość na początek tablicy. Metoda insertOnIndex dodaje wartość na wybranej pozycji w tablicy, tworząc Metoda del() usuwa element o podanym indeksie z tablicy. Metody delFront() i delBack() usuwają odpowiednio pierwszy i ostatni element z tablicy. Metoda get(int index) zwraca wartość elementu o podanym indeksie. Metoda print() wypisuje na ekranie wszystkie elementy tablicy. Metoda getSize() zwraca aktualny rozmiar tablicy. Metoda indexOf(int value) zwraca indeks pierwszego wystąpienia podanej wartości w tablicy.

Fragment implementacji w programie

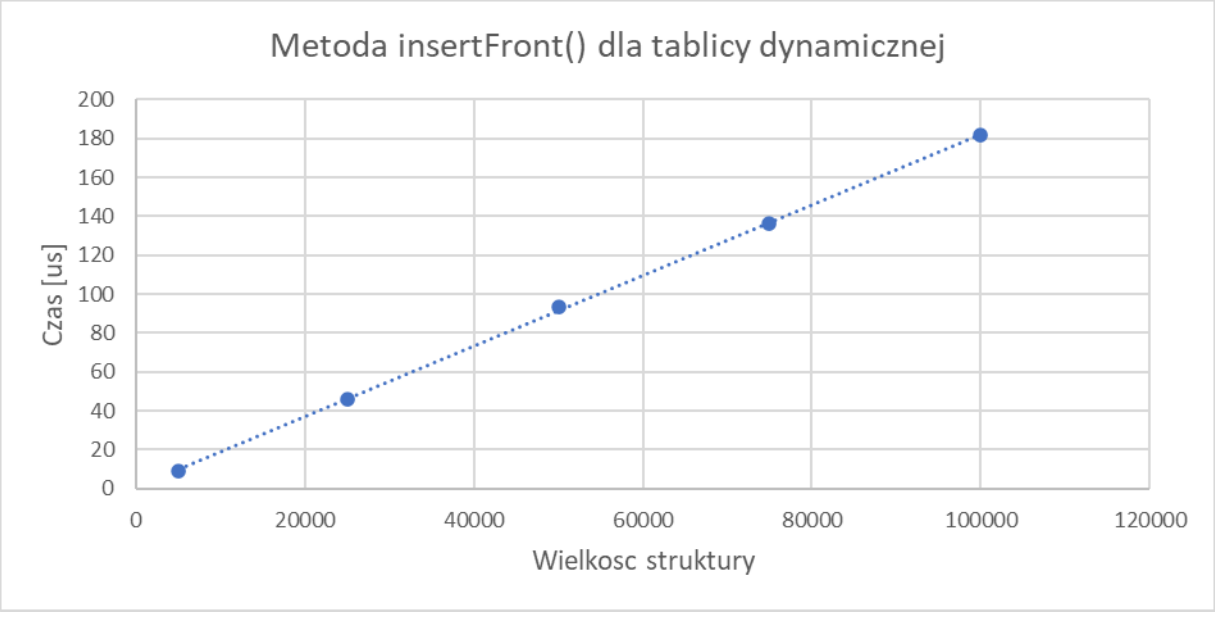
```
class Table {

    int size;
    int *table;
public:
    Table() {
        this->size=0;           // dane inicjalizacyjne
        this->table= nullptr;
    }
```

2.2. Złożoności obliczeniowe metod listy

2.2.1. Insertfront()

Złożoność obliczeniowa metody insertFront dla tablicy dynamicznej, która przy każdym dodaniu alokuje miejsce w pamięci na nową tablicę o 1 większą wynosi **O(n)**, ponieważ każde dodanie elementu wymaga przepisania wszystkich elementów już znajdujących się w tablicy do nowo zaalokowanej pamięci, co jest bardzo kosztowne pod względem czasu. Lepszym rozwiązaniem byłoby zwiększenie rozmiaru tablicy o stałą wartość lub o pewien procent jej aktualnego rozmiaru, aby uniknąć częstych realokacji pamięci. Wtedy złożoność obliczeniowa metody insertFront wyniosłaby **O(1)** lub **O(log n)**, w zależności od sposobu zwiększania rozmiaru tablicy. Moja implementacja jest wersją, która realokuje pamięć przy każdym dodaniu elementu, więc jej złożoność wynosi **O(n)**.

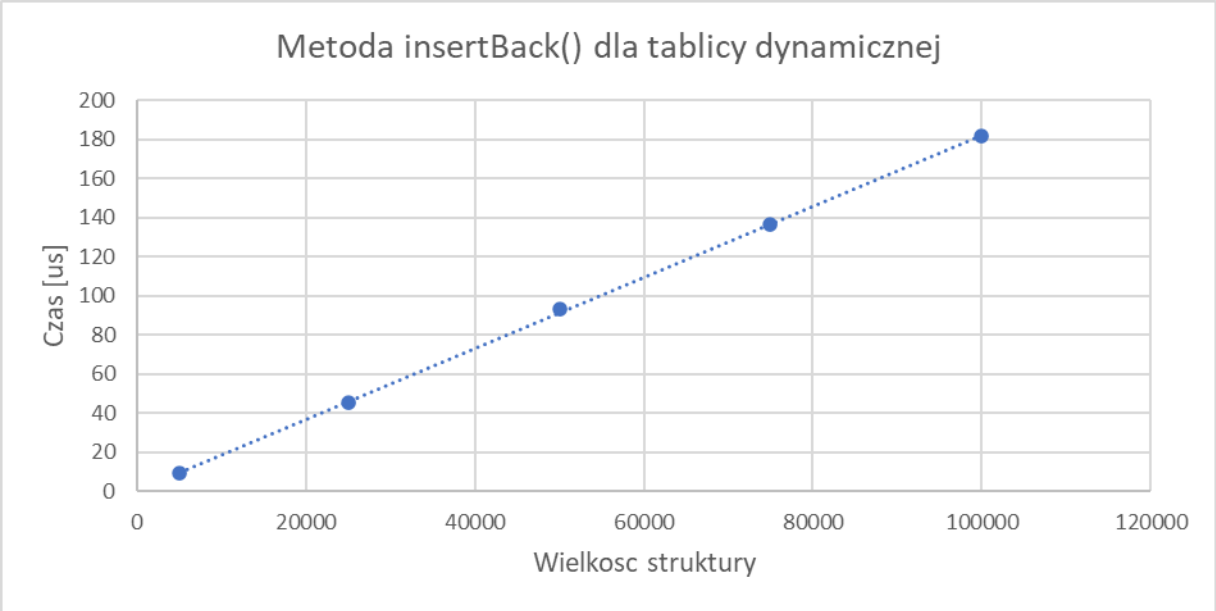
Wielkość struktury	Czas [us]
5000	9,3
25000	45,6
50000	93,4
75000	136,5
100000	181,7



2.2.2. InsertBack()

Złożoność obliczeniowa metody insertBack dla tablicy dynamicznej, jest identyczna jak w metodzie insertFront, z różnicą, że element jest dodawany na koniec. Złożoność obliczeniowa tej metody wynosi **O(n)**.

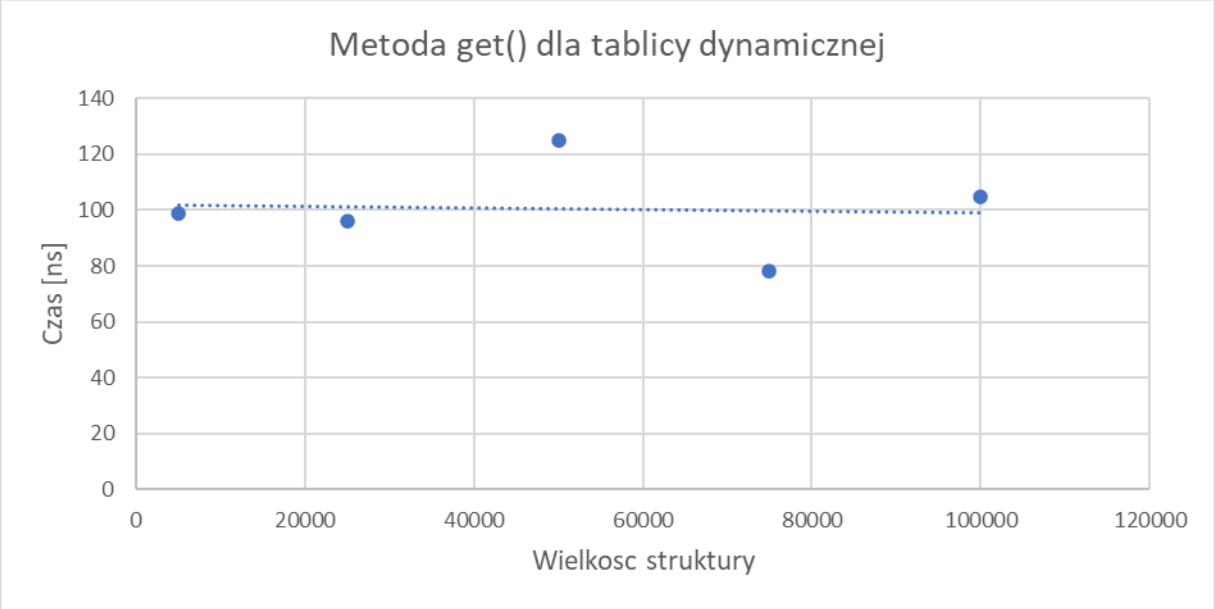
Wielkość struktury	Czas [us]
5000	9,3
25000	45,2
50000	113,7
75000	135,1
100000	179,4



2.2.3. Get()

Złożoność obliczeniowa metody get() dla tablicy dynamicznej wynosi **O(1)**, ponieważ elementy tablicy są przechowywane w ciągłej sekwencji pamięci i dostęp do każdego elementu jest stały i odbywa się przez indeks. Zgadza się to z testami czasowymi wykonanej implementacji.

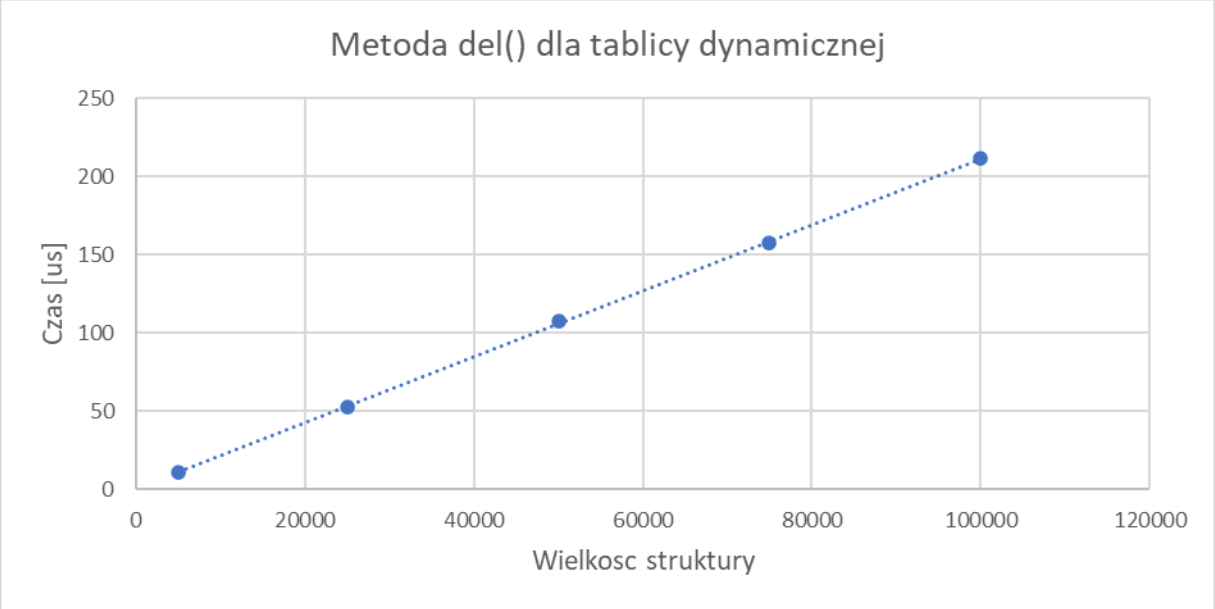
Wielkość struktury	Czas [ns]
5000	99
25000	96
50000	125
75000	78
100000	105



2.2.4. Del()

Złożoność obliczeniowa metody del dla tablicy dynamicznej, która przy każdym usunięciu alokuje miejsce w pamięci na nową tablicę o 1 mniejszą wynosi tyle samo co dla dodania, **O(n)**, ponieważ każde usunięcie elementu wymaga przepisania wszystkich elementów już znajdujących się w tablicy do nowo zaalokowanej pamięci. Zgadza się to z przetestowanymi wynikami.

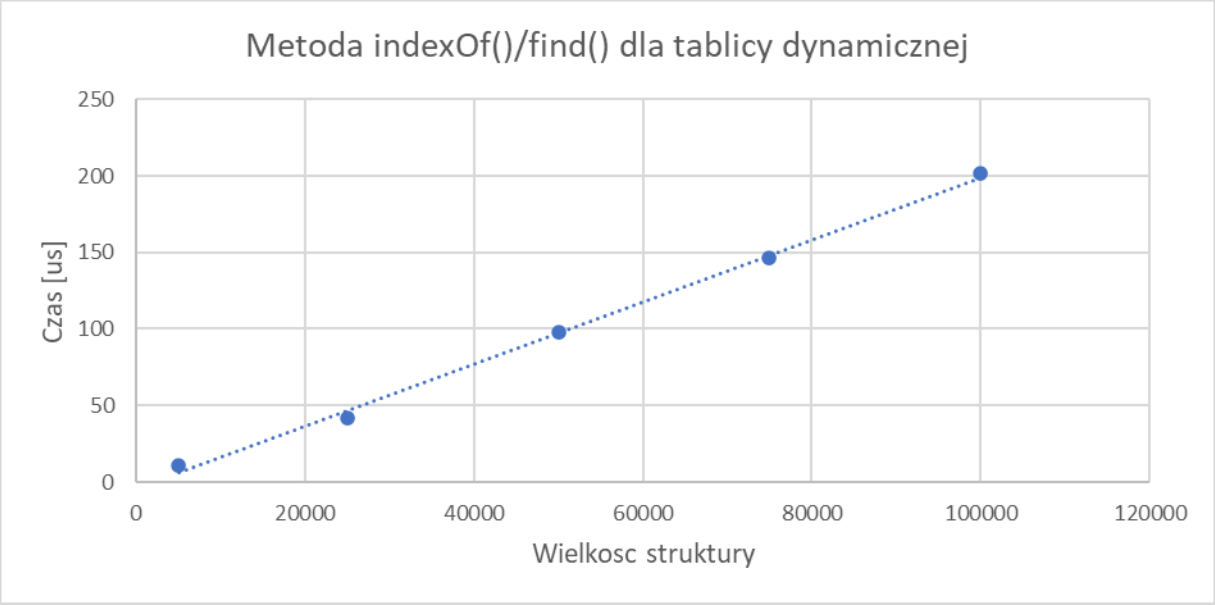
Wielkość struktury	Czas [us]
5000	10,9
25000	52,9
50000	107,5
75000	157,6
100000	211,4



2.2.5. Find()/IndexOf()

Jeśli tablica jest nieposortowana, to, aby znaleźć element o podanej wartości, konieczne jest przeszukanie jej liniowo, zajmuje to czas proporcjonalny do rozmiaru tablicy **O(n)**, gdzie n to liczba elementów w tablicy. Jeśli jednak tablica jest posortowana, to można skorzystać z wyszukiwania binarnego, co zmniejsza złożoność czasową do **O(log n).** W naszym przypadku tablica nie jest posortowana, a w testach jej złożoność wynosi O(n).

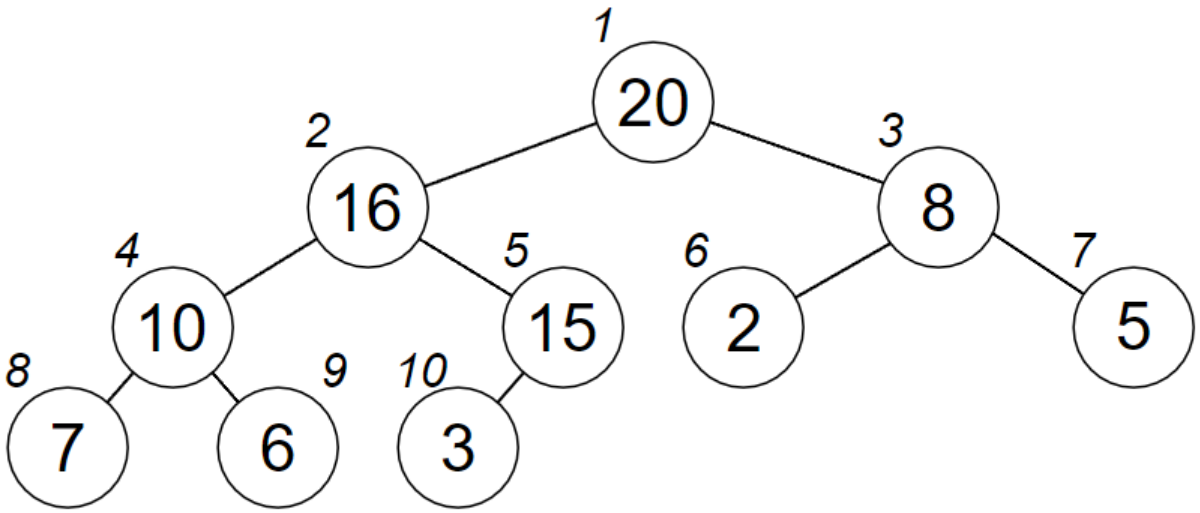
Wielkość struktury	Czas [us]
5000	10,4
25000	41,5
50000	98,3
75000	146,6
100000	201,4



3. Kopiec binarny

3.1. Wstęp

Kopiec binarny to struktura danych, która służy do przechowywania zbioru elementów oraz wykonywania operacji dodawania i usuwania elementów. Jest to drzewo, w którym każdy węzeł ma wartość większą lub równą wartości w swoich dzieciach, a korzeń ma największą wartość.



Źródło: https://pl.wikipedia.org/wiki/Kopiec_binarny

Metoda print() to funkcja pomocnicza do wizualizacji kopca na ekranie. Wykorzystuje ona łańcuchy znaków cr, cl i cp, aby rysować kopiec. Metoda print jest wywoływana rekurencyjnie dla każdego węzła drzewa, zaczynając od korzenia. Funkcja ta została wzięta ze źródła stackoverflow.

Metoda pop() usuwa korzeń kopca, a następnie odtwarza właściwości kopca binarnego. Metoda **push**() dodaje nowy element do kopca, a następnie odtwarza właściwości kopca binarnego. Metoda **getSize**() zwraca rozmiar kopca. Metoda **peek**() zwraca wartość korzenia kopca bez usuwania go. Metoda **search**() wyszukuje element o określonej wartości w kopcu i zwraca indeks tego elementu lub -1, jeśli taki element nie został znaleziony.

Fragment implementacji w programie

```
using namespace std;

class Heap {

private:
    string cr, cl, cp;      // łańcuchy znaków do wyświetlania kopca
    int size = 0;
    int *tab;


public:

    Heap(){
        cr = cl = cp = "  ";
        cr [0] = 218; cr [1] = 196;
        cl [0] = 192; cl [1] = 196;
        cp [0] = 179;
        this -> tab = nullptr;
    }
```

Klasa Kopca (Heap) zawiera trzy prywatne pola:

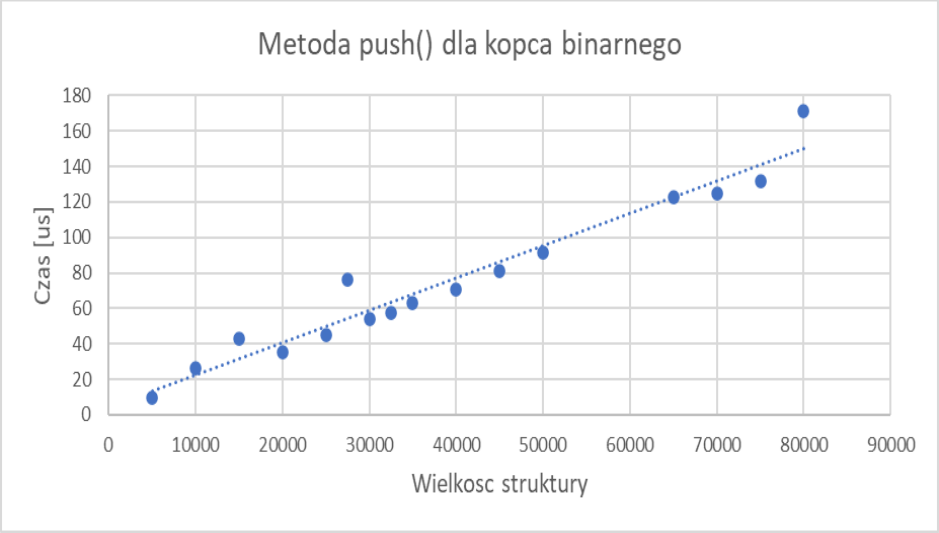
- **cr**, **cl** i **cp** - to są łańcuchy znaków używane do wyświetlania kopca,
- **size** - rozmiar kopca,
- ***tab** - to jest wskaźnik na tablicę przechowującą elementy kopca.

3.2. Złożoności obliczeniowe metod kopca

3.2.1. Push()

W przypadku standardowego kopca binarnego (binary heap), dodanie elementu za pomocą metody push() wymaga przeprowadzenia operacji wstawienia elementu na ostatni poziom kopca, a następnie wykonywania operacji przesiewania w górę, aby przywrócić porządek kopca. W najgorszym przypadku, operacja przesiewania w górę musi zostać wykonana dla każdego poziomu kopca, co skutkuje złożonością obliczeniową O(log n), gdzie n to liczba elementów w kopcu. Jednakże dla naszej implementacji przy pomocy tablicy trzeba zaalokować pamięć na nową tablicę, co wynosi O(n), więc **O(log n + n)** = **O(n)**, dla naszego kopca. Można przerobić tą implementacją tworząc tablicę większą, niż jest potrzebna, wtedy, dopóki elementy mieściły by się w tablicy złożoność metody push() wynosiłaby **O(log n)**.

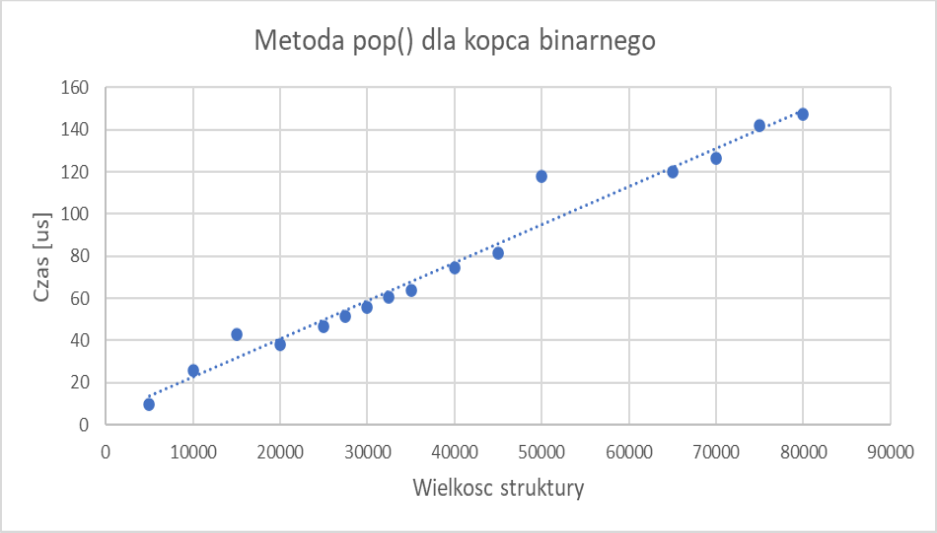
Wielkość struktury	Czas [us]
5000	9,6
10000	26
15000	42,9
20000	35,4
25000	45,3
27500	76
30000	54,2
32500	57,3
35000	63,2
40000	70,5
45000	81
50000	91,3
65000	122,4
70000	124,4
75000	131,7
80000	171,3



3.2.2. Pop()

Podobnie jak dla metody push() w przypadku standardowego kopca binarnego usunięcie elementu za pomocą metody pop() zaalokowania pamięć na nową tablicę, co wynosi O(n), więc **O(log n + n)** = **O(n)**, dla naszego kopca. Można przerobić tą implementacją tworząc tablicę większą, niż jest potrzebna, wtedy, dopóki elementy mieściły by się w tablicy złożoność metody push() wynosiłaby **O(log n)**.

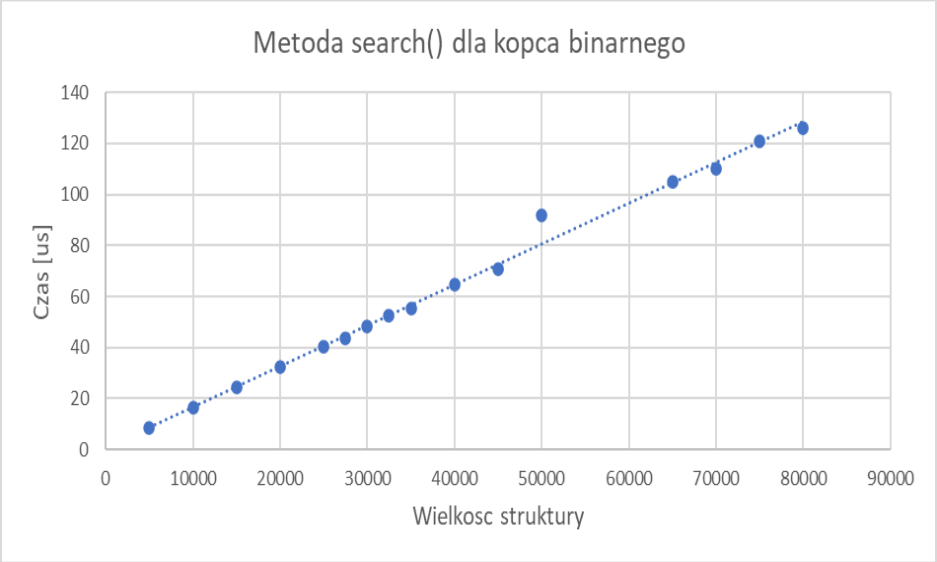
Wielkość struktury	Czas [us]
5000	9,7
10000	25,9
15000	42,8
20000	37,8
25000	46,5
27500	51,3
30000	55,7
32500	60,3
35000	63,6
40000	74,6
45000	81,2
50000	117,7
65000	120,1
70000	126,2
75000	141,8
80000	147,4



3.2.3. Search()

Złożoność metody search() dla kopca binarnego zaimplementowanego jako tablica wynosi O(n), gdzie n to liczba elementów w kopcu. W metodzie tej iterujemy po każdym elemencie w tablicy, aż znajdziemy element równy szukanemu, lub dojdziemy do końca tablicy. W najgorszym przypadku, gdy szukany element nie znajduje się w kopcu lub znajduje się na końcu tablicy, będziemy musieli przejrzeć całą tablicę, co daje nam złożoność czasową **O(n).** Warto jednak zaznaczyć, że kopiec binarny nie jest strukturą danych przeznaczoną do efektywnego wyszukiwania pojedynczych elementów, a raczej do szybkiego dodawania i usuwania elementów o najwyższej/ najniższej wartości.

Wielkość struktury	Czas [us]
5000	8,2
10000	16,6
15000	24,3
20000	32,4
25000	40,2
27500	43,4
30000	48,4
32500	52,5
35000	55,2
40000	64,5
45000	70,9
50000	92
65000	104,8
70000	110,3
75000	120,8
80000	126,3



3.2.4. Seek()

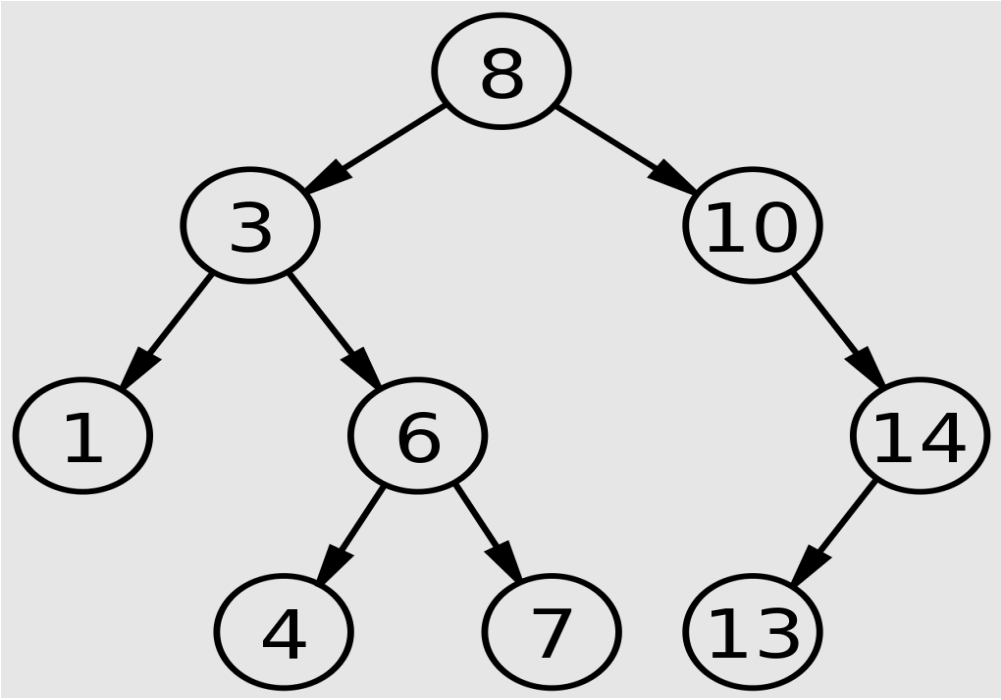
Złożoność metody peek() dla kopca binarnego zaimplementowanego jako tablica wynosi O(1), ponieważ wartość korzenia (czyli pierwszego elementu w tablicy) jest przechowywana w stałej pozycji i nie wymaga żadnych dodatkowych operacji obliczeniowych, aby ją zwrócić.

Wielkość struktury	Czas [us]
5000	0
10000	0
15000	0,1
20000	0,1
25000	0
27500	0
30000	0
32500	0
35000	0,1
40000	0
45000	0
50000	0
65000	0,1
70000	0
75000	0
80000	0

4. Drzewo przeszukiwań binarnych BST

4.1. Wstęp

Drzewo BST jest to struktura danych, w której każdy węzeł drzewa przechowuje klucz (liczbę), przy czym klucz ten jest większy niż klucz każdego węzła w jego lewym poddrzewie i mniejszy niż klucz każdego węzła w jego prawym poddrzewie. Dzięki temu **BST** umożliwia bardzo szybkie wyszukiwanie, dodawanie oraz usuwanie elementów. Implementacja zawiera dwie klasy: **Node** oraz **BST**. Klasa **Node** reprezentuje węzeł drzewa, zawiera pole przechowujące wartość oraz wskaźniki na lewe i prawe poddrzewo. Klasa **BST** to klasa implementująca **BST**, zawiera metody umożliwiające operacje na drzewie, takie jak dodawanie, usuwanie oraz wyszukiwanie elementów. Metoda **insert()** pozwala na dodanie nowego elementu do drzewa. Metoda **search()** umożliwia wyszukanie elementu o podanym kluczu. Metoda **remove()** umożliwia usuwanie elementu o podanym kluczu. Metoda **getRoot()** zwraca korzeń drzewa. Implementacja nie zawiera algorytmu równoważenia drzewa, przez co metody, dla najgorszych scenariuszy nie są najszybsze jak by mogły.



Źródło: https://pl.wikipedia.org/wiki/Binarne_drzewo_poszukiwa%C5%84

Fragment implementacji w programie

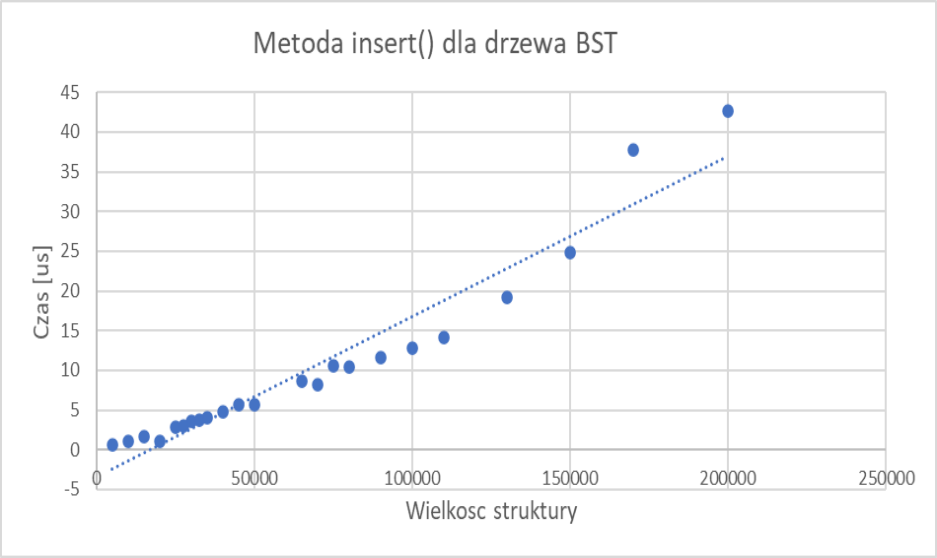
```
class Node {
public:
    int data;    // data = key   klucz odpowiadajacy za wartosc elementu
    Node *left;
    Node *right;

    Node(int k) {
        data = k;
        left = right = nullptr;
    }
};

class BST {
private:
    string cr, cl, cp;
    Node *root;
```

4.2. Złożoności obliczeniowe metod BST

4.2.1. Insert()

Złożoność obliczeniowa metody insert w drzewie BST wynosi **O(h)**, gdzie h to wysokość drzewa. W najlepszym przypadku, gdy drzewo jest zrównoważone, h wynosi **O(log n)**, gdzie n to liczba węzłów w drzewie. W najgorszym przypadku, gdy drzewo jest wyprostowane, czyli przypomina listę, złożoność wynosi O(n). W takim przypadku, każdy nowy węzeł musi zostać dodany jako liść, co wymaga przejścia przez całą długość drzewa, co daje nam **O(n)**. W tej implementacji bez równoważenia drzewa, złożoność w najgorszym przypadku wynosi **O(n)**. Można poprawić szybkość tego drzewa dodając metody rotacji, oraz równoważenia drzewa np. algorytmem DSW.

Wielkość struktury	Czas [us]
5000	0,6
10000	1,1
15000	1,6
20000	1,1
25000	2,9
27500	3
30000	3,6
32500	3,7
35000	4,1
40000	4,8
45000	5,7
50000	5,7
65000	8,7
70000	8,2
75000	10,6
80000	10,4
90000	11,6
100000	12,8
110000	14,1
130000	19,2
150000	24,9
170000	37,7
200000	42,7



4.2.2. Remove()

Złożoność obliczeniowa funkcji **remove()** drzewa BST zależy od wysokości drzewa. W najgorszym przypadku drzewo może mieć wysokość równą liczbie elementów, co oznacza, że algorytm będzie musiał przejść przez wszystkie elementy, co da złożoność czasową **O(n).** Jednakże, jeśli drzewo jest złożoność czasowa usuwania elementu wynosi **O(log n).**

Usuwanie elementu możemy podzielić na trzy przypadki:

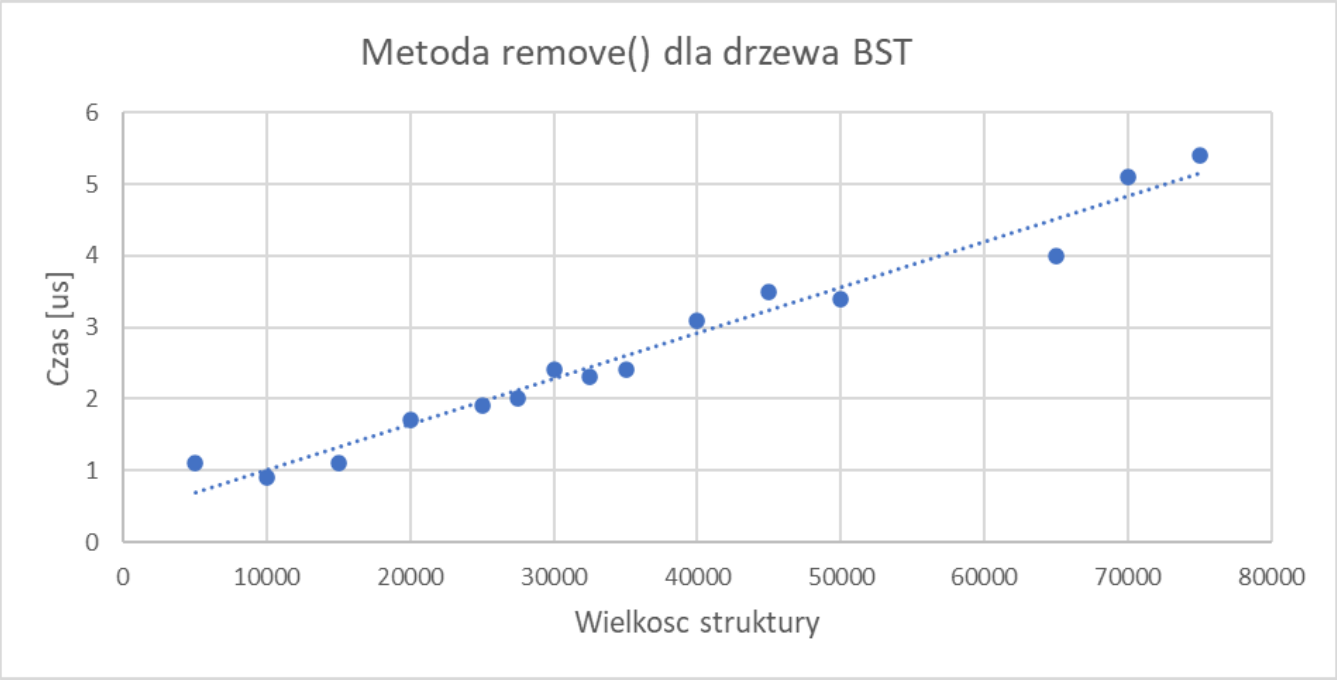
Przypadek 1: element jest liściem: Gdy usuwany element nie ma żadnych potomków, funkcja usuwa go z drzewa i zwalnia pamięć zajmowaną przez niego. W zależności od pozycji elementu w drzewie aktualizowane są wskaźniki left lub right na jego rodzica lub korzeń drzewa.

Przypadek 2: element ma 1 potomka: Gdy usuwany element ma jednego potomka, funkcja zastępuje go tym potomkiem i usuwa element, aktualizując wskaźniki na rodzica i korzeń drzewa w ten sam sposób jak w przypadku 1.

Przypadek 3: element ma dwoch potomkow - metoda następnika: Gdy usuwany element ma dwóch potomków, funkcja znajduje następnik (najmniejszy element w poddrzewie prawego potomka) i zastępuje nim usuwany element. Następnie funkcja usuwa znaleziony następnik, aktualizując wskaźniki na rodzica i korzeń drzewa w ten sam sposób jak w przypadku 1 i 2.

Wielkość struktury	Czas [us]
5000	1,1
10000	0,9
15000	1,1
20000	1,7
25000	1,9
27500	2
30000	2,4

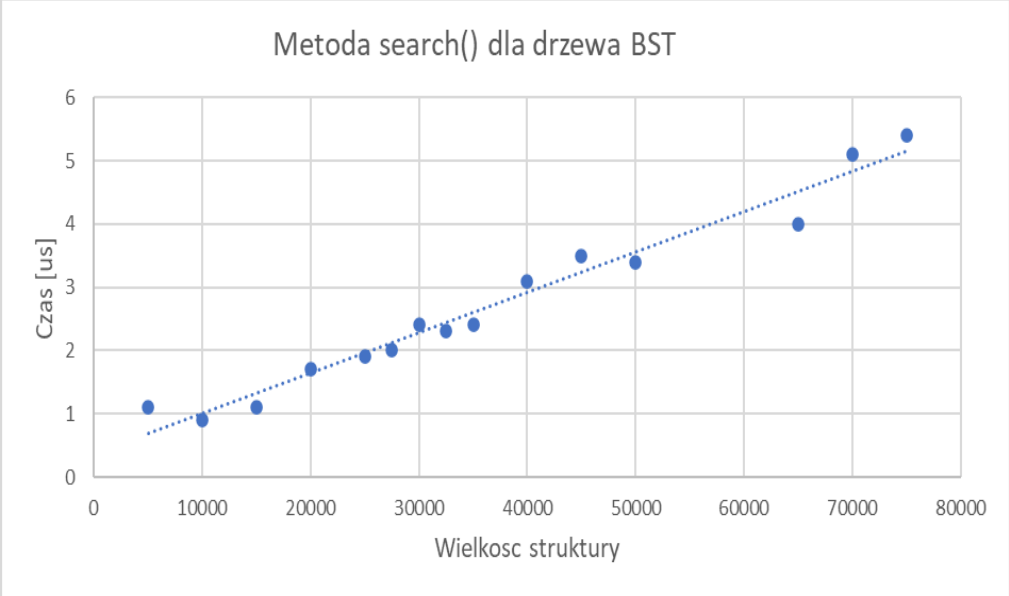
Wielkość struktury	Czas [us]
32500	2,3
35000	2,4
40000	3,1
45000	3,5
50000	3,4
65000	4
70000	5,1
75000	5,4
80000	6,6
90000	7,4
100000	7,9
110000	9
130000	12,2
150000	14,3
170000	17,5
200000	22,6
250000	30,6



4.2.3. Search()

Złożoność obliczeniowa metody **search()** to **O(h)**, gdzie **h** to wysokość drzewa binarnego. W przypadku najlepszego scenariusza, kiedy drzewo jest zrównoważone, wysokość h jest rzędu logarytmu o podstawie 2 z liczby węzłów w drzewie, co oznacza, że złożoność obliczeniowa wynosi **O(log n)**. Jednakże, w naszej implementacji, kiedy drzewo nie jest równoważone, jest możliwy najgorszy przypadek (na przykład w postaci ciągu węzłów umieszczonych w jednej linii), wysokość drzewa wynosi n, a złożoność obliczeniowa wynosi **O(n)**. Dlatego, złożoność obliczeniowa metody wyszukiwania w drzewie binarnym zależy od jego struktury i ilości węzłów.

Wielkość struktury	Czas [us]
5000	0,3
10000	0,5
15000	0,7
20000	1
25000	1,4
27500	1,5
30000	1,7
32500	1,8
35000	2
40000	2,4
45000	2,8
50000	2,9
65000	4,4
70000	4,5
75000	5,1
80000	5,7



5. Podsumowanie

W ramach projektu z zaimplementowano tablice dynamiczną, listę dwukierunkową, kopiec oraz drzewo BST przetestowano działanie struktur na przykładzie dodawania, usuwania i wyszukiwania elementów. Wszystkie testy zostały pomyślnie zakończone, a otrzymane wartości zgadzają się z wartościami oczekiwanymi. Jednakże dla kilku struktur, lekkie zmiany w implementacji, znacznie zmniejszyły by czas wykonywania implementacji. Np.

- Alokowanie tablicy większej, niż aktualnie jest potrzebna.

- Równoważenie drzewa BST, po każdym dodaniu lub usunięciu elementu.

Struktura	dodaj()	usuń()	znajdź()
Tablica dynamiczna	O(n)	O(n)	O(1)
Lista dwukierunkowa	O(1)	O(1)	O(n)
Kopiec	O(n)	O(n)	O(n)
Drzewo BST	O(n)/O(log n)	O(n)/O(log n)	O(n)/O(log n)