Piotr Karoń, 241 626 Wrocław, dn. 03.04.2019

Zadanie projektowe 1.

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

PROWADZĄCY:

dr Jarosław Mierzwa

Spis treści

1 Założenia projektowe

1.1 Cel

Celem projektu jest zbadanie efektywności operacji dodawania, usuwania i wyszukiwania elementów w strukturach danych takich jak:

- tablica
- lista dwukierunkowa
- kopiec binarny (maksymalny)
- drzewo BST
- drzewo AVL
- drzewo czerwono-czarne

1.2 Technologie

Do implementacji wymienionych struktur użyto języka ***Kotlin*** w wersji ***Native***, która jest kompilowana do kodu maszynowego danej platformy.

Wszystkie struktury zostały zaimplementowane samodzielnie, bez użycia gotowych rozwiań z biblioteki standardowej. Należy jednak zwrócić uwagę, że **Kotlin** nie pozwala na bezpośrednie utworzenie tablicy. W zamian udostępnia klasę parametryzowaną *Array*. Została więc ona wykorzystana jako podstawa implementacji. Jedynymi użytymi metodami tej klasy są *set* oraz *get* odpowiadające operatorowi *[]*.

1.3 Przebieg eksperymentu

Każda ze struktur zostanie wypełniona elementami w ilościach 10, 100, 200, 500, 1000, 2000, 5000, 7000, 10 000.

Następnie, po wypełnieniu zostaną wykonane operacje dodawania, usuwania i wyszukiwania elementu. Dla drzewa BST osobno dodawanie i usuwanie z każdorazowym równoważeniem algorytmem DSW. Dla każdej ilości testy zostaną przeprowadzone 30 razy, a otrzymany wynik zostanie uśredniony. Czas nie będzie liczony dla generowania/wczytywania danych.

2 Krótki opis struktur

Przy tworzeniu poniższych opisów korzystano z `Wprowadzenia do algorytmów` autorstwa T. Cormen, C. Leiserson, R. Rivest [1] oraz materiałów udostępnionych na stronie dr Tomasza Kapłona [3].

2.1 Tablica

Tablicą nazywamy ciągły blok pamięci, gdzie każda komórka przechowuje informację jednego typu. Do każdej z nich mamy bezpośredni dostęp poprzez indeks.
Dostęp do określonego elementu jest natychmiastowy właśnie dzięki indeksacji. Złożoność czasowa wynosi $\mathbf{O(1)}$.
Wyszukanie elementu o zadanej wartości w najgorszym przypadku będzie wymagało przejścia przez całą strukturę – $\mathbf{O(n)}$.
Wstawienie elementu na koniec jest proste i tanie (nie licząc ewentualnej potrzeby realokacji tablicy) – sprowadza się do zwiększenia rozmiaru i wstawienia elementu – $\mathbf{O(1)}$
Wstawienie wewnątrz wymaga przesunięcia elementów za wskazanym indeksem – w najgorszym wypadku wstawienie na 1. pozycję skutkuje złożonością rzędu $\mathbf{O(n)}$. Usuwanie jest analogiczne do wstawiania – usuwanie z końca ma złożoność $\mathbf{O(1)}$, a usuwanie z wewnątrz także wymaga przesunięcia elementów – $\mathbf{O(n)}$.

2.2 Lista

Lista jest strukturą, która zawiera elementy składające się z: pola danych oraz wskaźnika na element następny (ewentualnie także poprzedni w liście dwukierunkowej). Dodatkowo lista musi mieć pole zawierające referencję do pierwszego elementu list - *głowy* (oraz *ogona* w wariancie dwukierunkowym). Zaletą tej struktury jest brak konieczności zapewnienia ciągłości w pamięci – może ona być w różnych miejscach.
Brak indeksacji powoduje, że dostęp i wyszukiwanie ma złożoność czasową rzędu $\mathbf{O(n)}$, gdyż trzeba przechodzić przez kolejne elementy.
Wstawienie sprowadza się do utworzenia nowego elementu i "podłączenia"go do odpowiednich wskaźników – koszt $\mathbf{O(1)}$.

Samo usunięcie elementu to $\mathbf{O(1)}$, ale trzeba ten element najpierw znaleźć – koszt rośnie więc do $\mathbf{O(n)}$.

2.3 Kopiec binarny (maksymalny)

Jest to drzewo binarne (prawie) pełne, w którym wartość rodzica jest zawsze nie mniejsza od obu potomków. Dzięki temu mamy zapewnienie, że w korzeniu znajduje się element maksymalny. Ta struktura może zostać zaimplementowana jako tablica lub lista.
Dostęp istnieje jedynie do elementu w korzeniu.
Po każdej operacji usunięcia i wstawienia elementu konieczne jest wywołanie procedury przywracającej własność kopca – złożoność obliczeniowa wynosi $\mathbf{O}(log_2 n)$, gdzie $log_2 n$ to wysokość drzewa.

2.4 Drzewo BST

Aby drzewo było BST musi dla każdego węzła x spełniać warunek, że wartość każdego elementu lezącego w lewym poddrzewie węzła x jest nie większa niż wartość węzła x, natomiast wartość każdego elementu leżącego w prawym poddrzewie węzła x jest nie mniejsza niż wartość tego węzła. Drzewo BST nie musi być pełne, więc jego wysokość k może być większa niż $log_2 n$ – w najgorszym wypadku drzewo może się zdegenerować do listy liniowej – ale można wykazać, że średnia wartość k dla losowo zbudowanego drzewa wynosi $\mathbf{O}(log_2 n)$. W celu zrównoważenia drzewa stosuje się algorytm DSW lub implementuje drzewa czerwono-czarne albo AVL.
Wyszukanie elementu można wykonać w czasie $\mathbf{O(k)}$. Na tę operacje składa się ze schodzenia po drzewie i sprawdzaniu czy szukany klucz jest mniejszy czy większy od aktualnie sprawdzanego węzła.
Wstawienie i usunięcie węzła także ma złożoność $\mathbf{O(k)}$

2.5 Drzewo czerwono-czarne

Dzięki własności czerwono-czarnym [2] to drzewo jest w przybliżeniu zrównoważone.
Wstawienie nowego węzła do drzewa czerwono-czarnego o n węzłach można wykonać w czasie $\mathbf{O}(log_2 n)$. Najpierw wstawiamy węzeł x do drzewa, a następnie należy naprawić powstałe drzewo.

Usuwanie węzła również odbywa się w czasie $\mathbf{O}(log_2 n)$.

Wyszukiwanie analogicznie do BST.

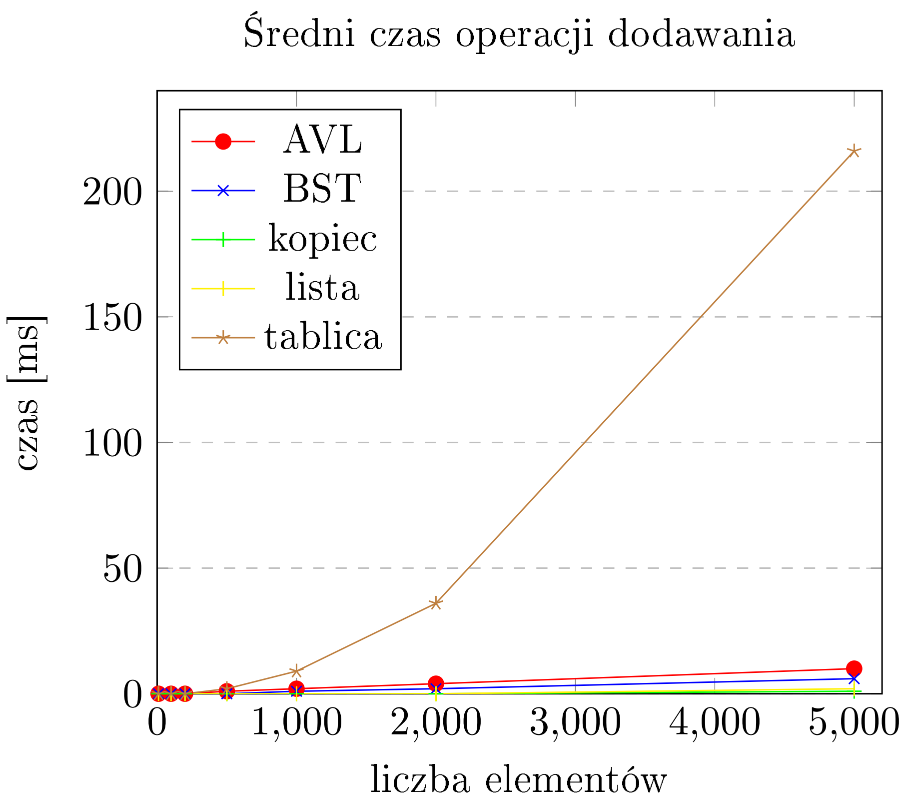
2.6 Drzewo AVL

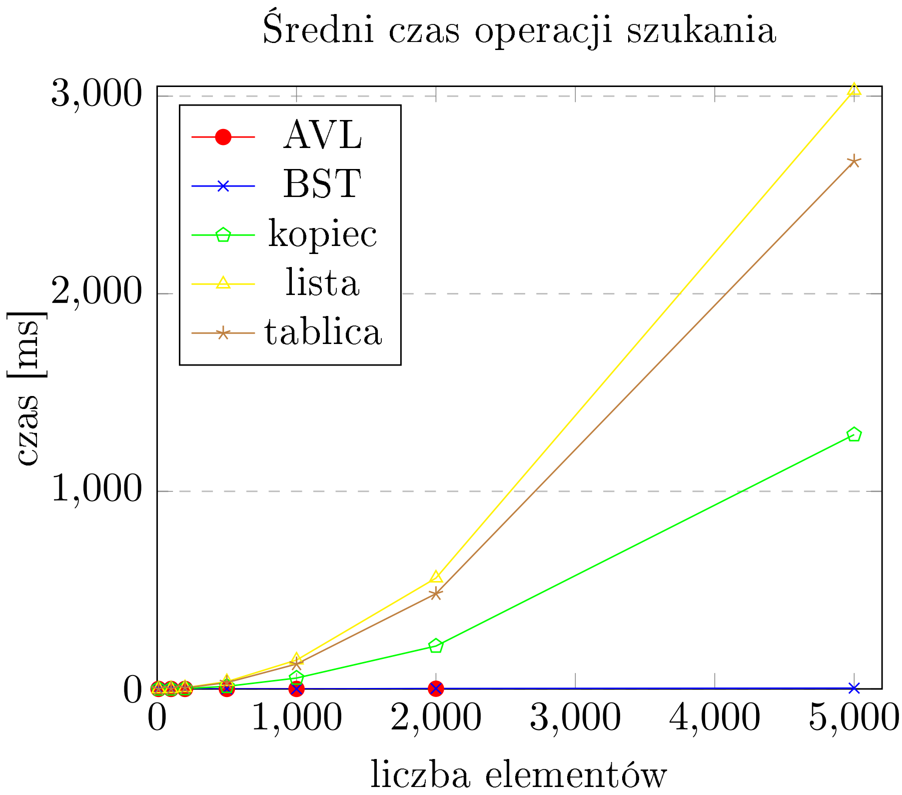
Drzewo binarne jest drzewem AVL, jeśli dla każdego węzła różnica wysokości dwóch jego poddrzew wynosi co najwyżej 1. Własności drzew AVL gwarantują, że nawet w najgorszym przypadku wysokość drzewa wyniesie $1.44 \times log(n+2)$.

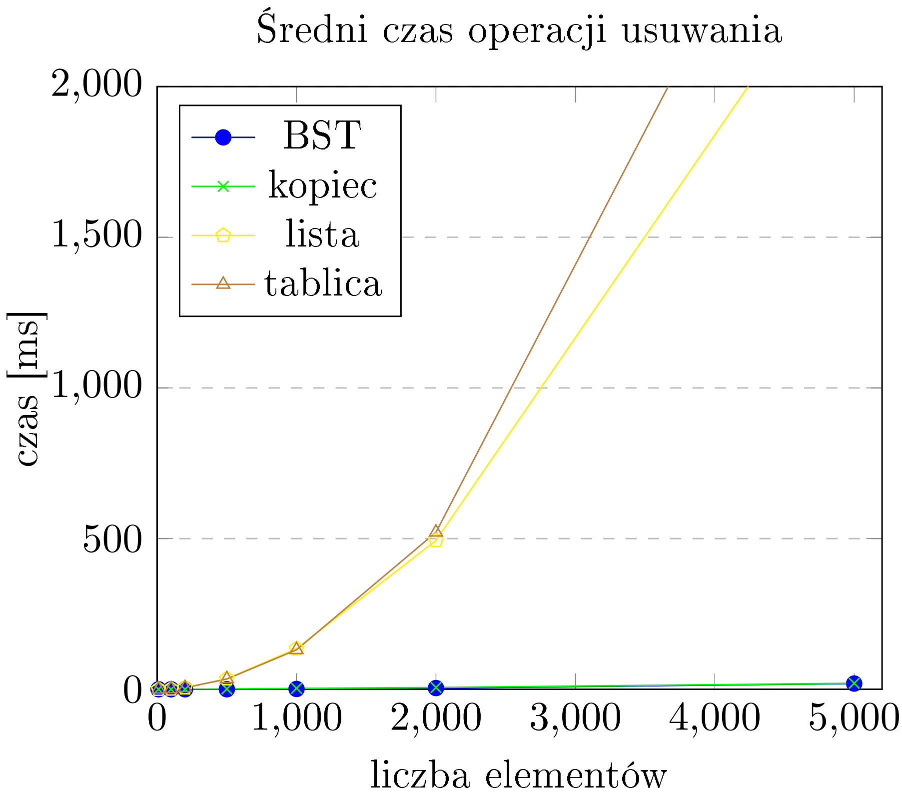
Koszt operacji wyszukiwania, dodawania, usuwania jest analogiczny jak w drzewie BST.

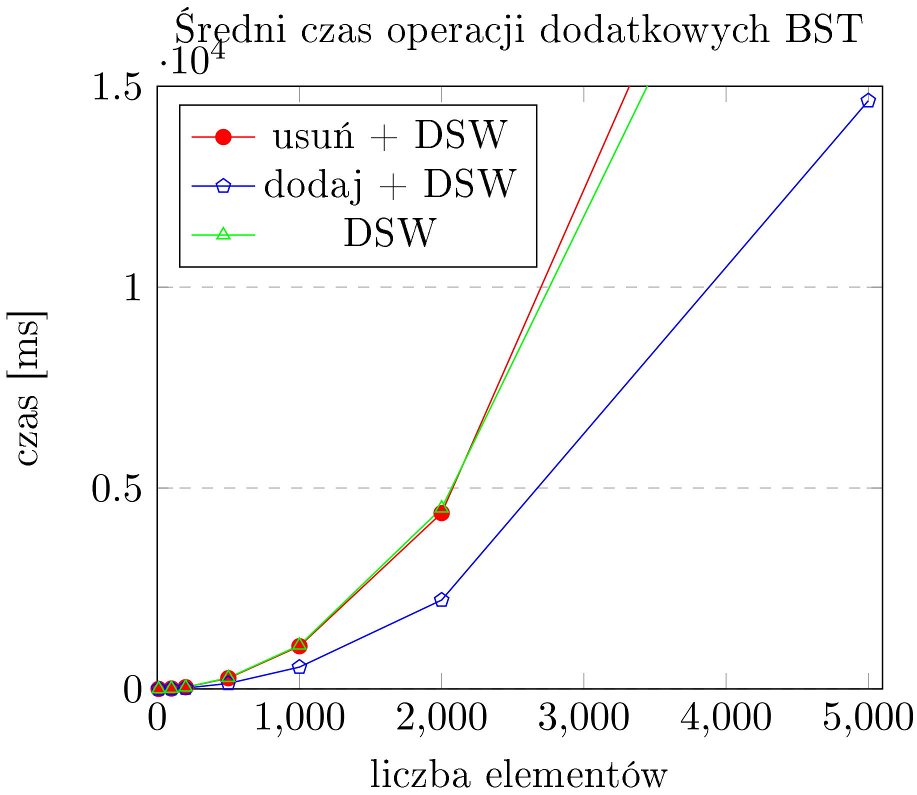
3 Wyniki

3.1 Wykresy




Średni czas operacji szukania
3,000
2,000
1,000
0
czas [ms]
0
1,000
2,000
3,000
4,000
5,000
liczba elementów
AVL
BST
kopiec
lista
tablica


Średni czas operacji usuwania
2,000
1,500
1,000
500
0
czas [ms]
0
1,000
2,000
3,000
4,000
5,000
liczba elementów
BST
kopiec
lista
tablica



3.2 Tabele

Tablica 1: Wyniki operacji dodawania w milisekundach

*	Tablica	Lista	Kopiec	Bst	Avl
10	0	0	0	0	0
100	0	0	0	0	0
200	0	0	0	0	0
500	2	0	0	0	1
1000	9	0	0	1	2
2000	36	0	0	2	4
5000	216	2	1	6	10

Tablica 2: Wyniki operacji szukania w milisekundach

*	Tablica	Lista	Kopiec	Bst	Avl
10	0	0	0	0	0
100	1	1	0	0	0
200	5	6	0	0	0
500	32	37	0	0	0
1000	126	146	0	1	0
2000	482	561	0	2	1
5000	2671	3029	1	6	2

Tablica 3: Wyniki operacji usuwania w milisekundach

*	Tablica	Lista	Kopiec	Bst	Avl
10	0	0	0	0	*
100	1	1	0	0	*
200	5	5	0	0	*
500	34	35	1	0	*
1000	131	135	3	1	*
2000	521	494	6	4	*
5000	3185	2512	20	19	*

Tablica 4: Wyniki operacji specjalnych BST

	Równoważenie	Usuwanie + DSW	Dodawanie + DSW
10	0	0	0
100	10	10	5
200	42	42	21
500	279	267	136
1000	1086	1063	543
2000	4497	4377	2212
5000	28523	26293	15005

4 Podsumowanie

Zaimplementowane algorytmy nie są optymalne. W większości jednak założona złożoność obliczeniowa sprawdziła się. Najbardziej obciążające były operacje na drzewie BST z wykorzystaniem algorytmu DSW. Równoważenie drzewa po każdym wstawieniu węzła nie jest dobrym pomysłem. Można to robić co kilka, kilkanaście wstawień – takie niezrównoważenie nie wpłynie bardziej na złożoność w przypadku innych operacji.

Bibliografia

[1] T. Cormen, C. Leiserson, R. Rivest *Wprowadzenie do algorytmów*, Wydawnictwa Naukowo-Techniczne Warszawa, Wyd. IV, 2004

[2] [1] str. 304.

[3] `tomasz.kaplon.staff.iiar.pwr.wroc.pl/`, strona dr Tomasza Kapłona

[4] `kotlinlang.org/docs/reference/native-overview.html`, dokumentacja języka Kotlin/Native

[5] `eduinf.waw.pl`, materiały na stronie I LO w Tarnowie