Helena Masłowska gr. 5.1
indeks: 148182
2 sem. Informatyka

Opracowanie struktur AVL, BST, list jednokierunkowych

Link do repozytorium: https://github.com/HelenaMaslowska/AVL-BST-LISTS

Spis treści

Wstęp

Celem niniejszego sprawozdania jest zaimplementowanie struktur w języku programowania C++ i sprawdzenie efektywności algorytmów takich jak AVL, BST, list w zależności od rodzaju danych wejściowych i liczby danych. Dane wejściowe zawierają liczbę elementów z przedziału od 10000 do 100000 elementów. Testy zostały powtórzone kilka razy tak, by można było wyrazić efektywność działania struktur. Do struktur zostały dodane takie funkcje jak: dodawanie, usuwanie, czy istnieje.

AVL

Dynamiczna struktura danych

Rozwiązanie

Jest podobne do BST, jednak różni się wysokością poszczególnych gałęzi drzew, wszystkie gałęzie powinny znajdować się na jednej linii ze sobą. Należy dodać funkcję która balansuje drzewo po dodaniu lub odjęciu z niego elementów.

Zalety

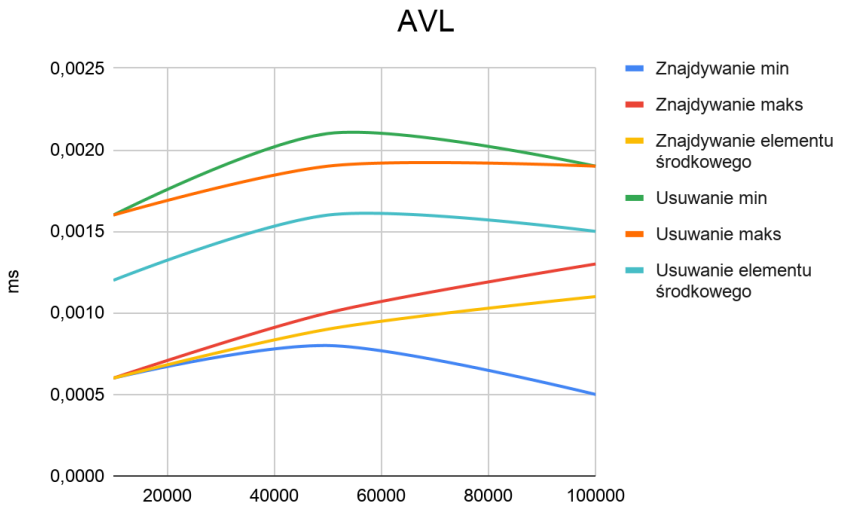
- działa szybciej niż BST
- zapewniona złożoność logarytmiczna O(log n)

Wady

- trudny do implementacji
- przy usuwaniu i dodawaniu elementu należy dokonać zrównoważenia drzewa AVL
- nie działa bez dodatkowej funkcji, której nie ma m.in. w BST: równoważenie drzewa AVL

Koszty operacji (wstawianie, wyszukiwanie, usuwanie - takie same dla każdego rodzaju):

- optymistyczny: O(1) - pierwszy element jest tym, którego szukamy/dodajemy/usuwamy
- pesymistyczny: O(log n) - najniżej położony element
- średni: O(log n) - średni czas



Ze względu na niedokładność pomiarową oraz potrzebę równoważenia drzewa te dane mogą nie przedstawiać faktycznej złożoności obliczeniowej. Ponadto dane wejściowe są różne na wejściu. Różnice czasowe są bliskie zeru, więc można przyjąć, że nie ma różnicy w złożoności.

BST

Dynamiczna struktura danych

Rozwiązanie

W struturze dodajemy pierwszy element, który jest korzeniem. Każdy kolejny element zostaje dodany przez następującą zasadę: jeśli dodawany element jest większy od korzenia to przejdź do kolejnego elemntu na prawo (jeśli istnieje) i ten staje się korzeniem, w przeciwnym razie w lewo (i lewy staje się korzeniem). Jeśli wskaźnik kolejnego (prawego lub lewego poddrzewa) jest równy NULL to dodaj wskaźnik na dodawany element w zamiast NULLa.
Przeszukiwanie działa analogicznie: jeśli szukamy elementu większego od korzenia to “idziemy na prawo” od korzenia, w przeciwnym razie w lewo, dopóki nie znajdziemy elementu.

Zalety

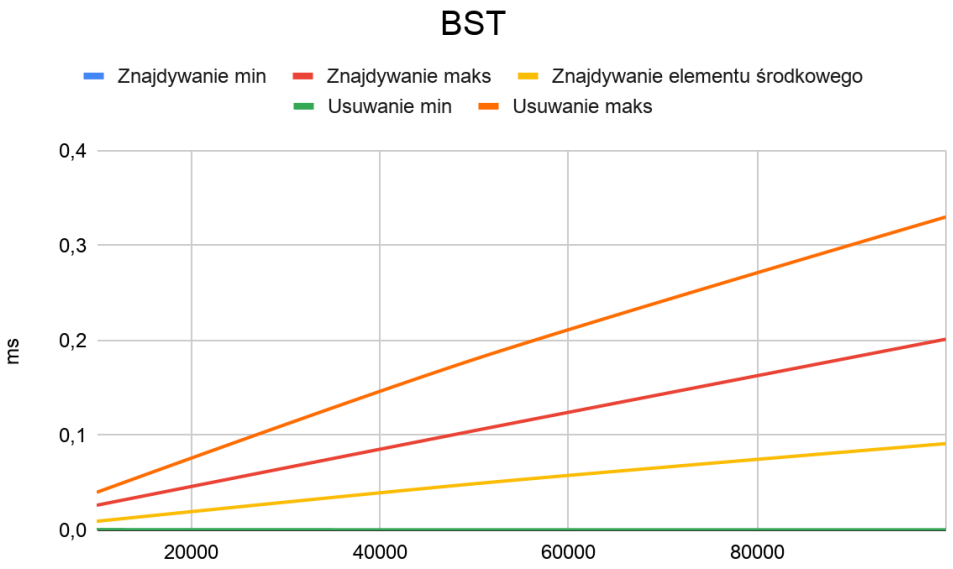
- niska złożonosć przeszukiwania
- działa szybciej niż lista jednokierunkowa

Wady

- działa dłużej niż AVL
- trudny do implementacji
- trudno się usuwa element z takiego drzewa (należy pamiętać, że po usunięciu elementu trzeba wstawić wskaźnik do “oderwanej” części drzewa

Koszty operacji (wstawianie, wyszukiwanie, usuwanie - takie same dla każdego rodzaju):

- proporcjonalny do wysokości drzewa h
- optymistyczny: O(1) - pierwszy element jest szukanym/usuwanym/dodawanym
- pesymistyczny: O(n) - gdy BST będzie listą jednokierunkową, a element będzie dodawany/usuwany / szukany element na końcu drzewa (szczególny przypadek testowy)
- średni: O(logn) - średnia złożoność



Usuwanie i znajdywanie min działa natychmiastowo, gdyż jest to pierwszy element. Najdłużej działa dla maksymalnych wartości, gdyż są one najdalej od korzenia.

Listy jednokierunkowe

Dynamiczna struktura danych

Rozwiązanie

Tworzymy posortowaną jednokierunkową listę jednokierunkową odwołując się do kolejnych elementów za pośrednictwem wskaźników “next”, które przechowują adresy kolejnych elementów. Aby dostać się do kolejnych elementów, należy przeiterować po każdym elemencie w liście aż do uzyskania “odpowiedniego” miejsca dla elementu.

Zalety

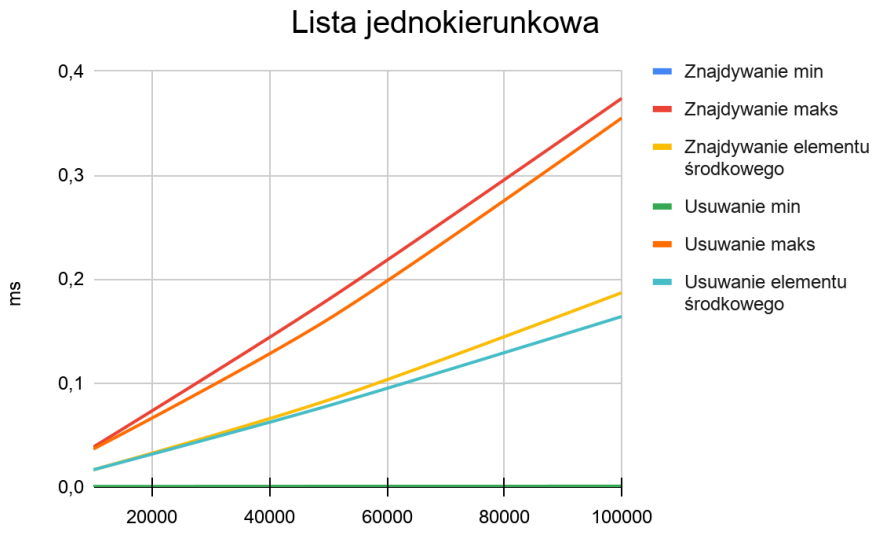
- warto używać gdy mamy mały zbiór danych wejściowych

Wady

- aby dostać się do ostatniego elementu listy (max) należy przeiterować po wszystkich elementach listy
- nieefektywny w porównaniu do porzedników (duża złożoność)

Koszty operacji (wstawianie, wyszukiwanie, usuwanie - takie same dla każdego rodzaju):

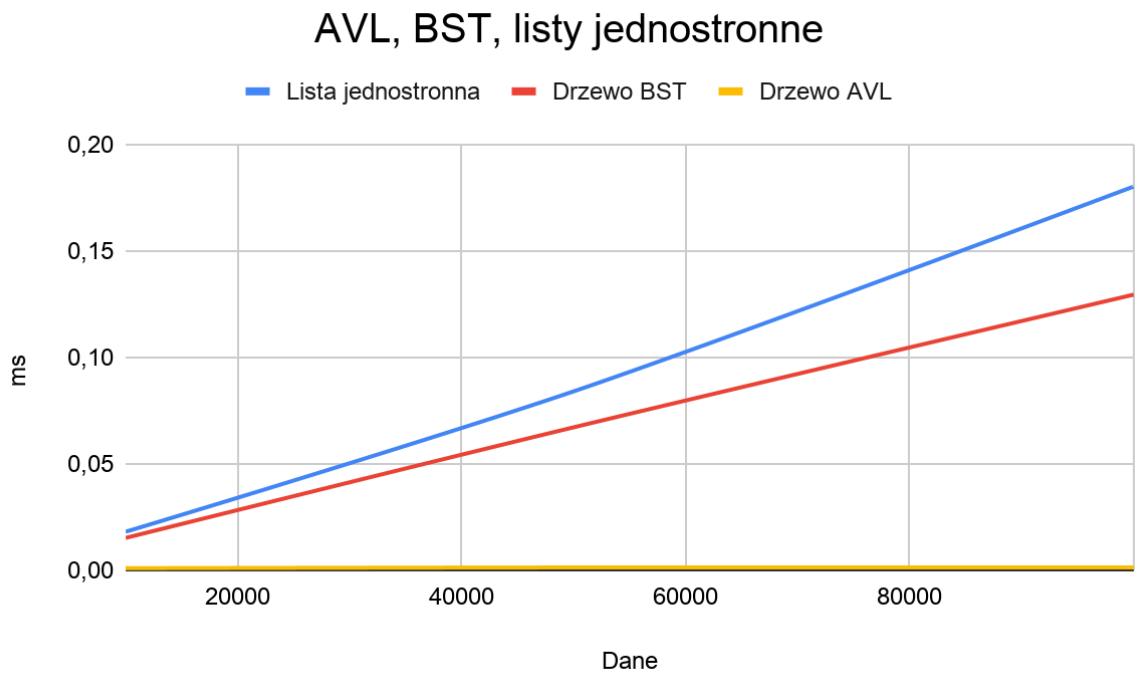
- optymistyczny: O(1) - element szukany/dodawany/usuwany stoi na pierwszym miejscu
- pesymistyczny: O(n) - element ostatni jest tym szukanym/usuwanym/dodawanym
- średni: O(n) - średnia złożoność



Najdłużej szuka maksymalną wartość w liście, gdyż występuje ona jako ostatnia, najkrócej usuwa i dodaje element minimalny w liście.

Zestawienie

Poniżej zostały przedstawione wszystkie 3 struktury porównujące uśrednione czasy wykonywania operacji na drzewach.



Podsumowanie

Zostały przedstawione 3 drzewa: AVL, BST, lista jednokierunkowa. Najbardziej efektywnym drzewem okazało się drzewo AVL i zdecydowanie jest bardziej opłacalny czasowo dla każdych danych, ze względu na niską złożoność obliczeniową. Jednak jeśli zależy nam na czasie i nie mamy za wiele czasu na napisanie kodu czy też mamy małą liczbę danych wejściowych to można się posłużyć listą, czy też BST.