AISDI zadanie 2

-

Drzewa BST i AVL

Mikołaj Rożek, Aleksandra Szymańska

1. Opisy struktur danych drzew

1. Drzewo BST

Drzewo Binary Search Tree (wyszukiwania binarnego) to struktura danych, która umożliwia między innymi sprawne wyszukiwanie i sortowanie wartości w nim zadanych. Składa się z obiektów klasy *Node* o określonych wartościach liczbowych, połączonych relacjami lewo- i prawo- synostwa. W przypadku drzewa BST parametr poddrzewa nie jest aktualizowany i zawsze wynosi 1. Każdy syn w lewym poddrzewie jest mniejszy od swojego rodzica, a prawym większy. Klasa *BinarySearchTree* posiada parametr *root* który węzłem korzeniowym drzewa i prowadzi do kolejnych węzłów drzewa (jeśli istnieją). Konstruktor klasy inicjalizuje obiekt z parametrem *root* (domyślnie None), który jest korzeniem drzewa i prowadzi do kolejnych węzłów (jeśli istnieją). Drzewo BST posiada również metody *insert_ite(), find_rec(), find_ite(), delete_node(), print_horizontally()* oraz funkcje pomocnicze *_get_node_with_minimal_value()* i *_delete_node()*. Większość tych funkcji zostanie opisana dokładniej przy porównaniu ich szybkości, dlatego tylko wspomnę, że w metodzie *print_horizontally()* drzewo jest wypisywane w terminalu przekręcone o 90 stopni w lewą stronę. To oznacza, że największy węzeł (najbardziej prawy) jest wypisywany w pierwszym wierszu, a najmniejszy (najbardziej na lewo) w ostatniej. Węzły na tym samej głębokości zaczynają się znakiem '*' w tej samej kolumnie.

2. Drzewo AVL

Drzewo **A**delson-**V**elsky and **L**andis (od nazwisk autorów) wariant drzewa BST, który się dodatkowo samobalansuje. Oznacza to, że wysokości lewego poddrzewa mogą się różnić maksymalnie o 1. Zapobiega to sytuacji, w której gdy węzły są dodawane rosnąco lub malejąco, korzeń będzie miał odpowiednio tylko prawych lub lewych synów, co ułatwia wyszukiwanie i przyspiesza obsługę przypadku pesymistycznego. Klasa *AVLTree* dziedziczy po klasie *BinarySearchTree* i używa z niej metod *find_ite()* oraz *print_horizontall()*. Posiada również własną metodę *insert_node()* oraz zmodyfikowaną metodę *delete_node()* z klasy *BinarySearchTree*. Używa funkcji pomocniczych *_insert_node(), _get_balance(), _get_height(), _rotate_left(), _rotate_right()*. Powyższe metody także zostaną bardziej szczegółowo opisane przy okazji przedstawiania szybkości ich działania.

2. Porównanie szybkości działania metod struktur danych

1. Tworzenie

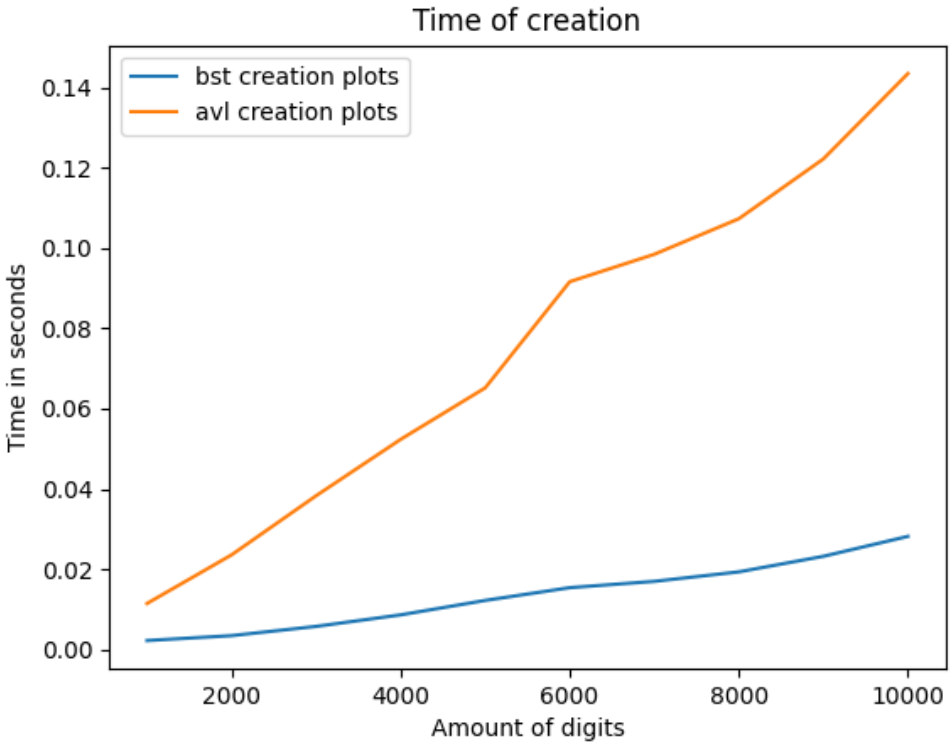
1. BST

Tworzenie drzewa BST polega na wstawianiu nowych węzłów, tak by każdy węzeł lewego poddrzewa był większy od rodzica, a węzły prawego poddrzewa powinny być od niego większe. Jest to realizowane funkcją *insert_ite()*, która tworzy węzeł o wskazanej wartości i szuka dla niego miejsca zaczynając od korzenia, sprawdzając czy wartość węzła jest większa czy mniejsza od porównywanej. Cały czas zapamiętywany jest rodzic obecnie sprawdzanego węzła, po to, żeby, gdy trafi się na wolne miejsce (rodzic nie będzie miał danego syna), można było wstawić węzeł jako odpowiednio lewy lub prawy syn tego rodzica. W przypadku, gdy drzewo nie ma korzenia, nowy węzeł się nim staje. Gdy węzeł o wartości znajdującej się już w drzewie, zostanie pominięty i nie będzie do niego wstawiony.

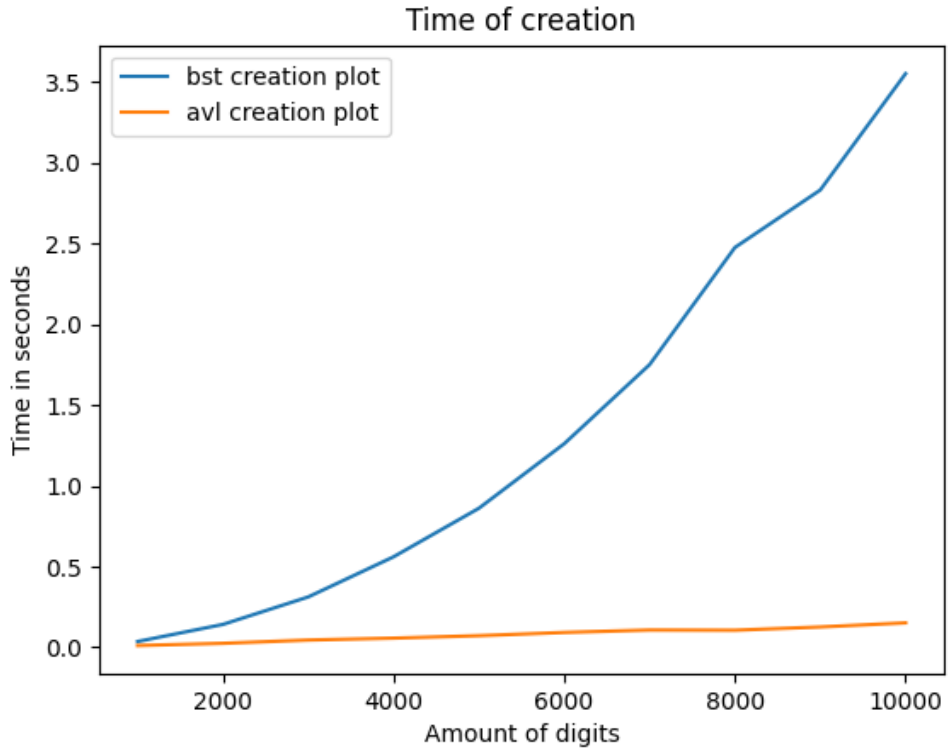
2. AVL

Tworzenie drzewa AVL polega na wstawieniu nowych węzłów tak, jak w przypadku drzewa BST, a następnie zbalansowaniu wielkości lewych i prawych poddrzew (jeśli to konieczne). Jest to realizowane rekursywną metodą *insert_node()* z pomocniczymi funkcjami *_insert_node(), _get_height(), _get_balance(), _rotate_left()* i *_rotate_right()*. Do wartości *roota* przypisywany jest wynik funkcji *_insert_node(),* która jeśli nie ma węzła podanego jej jako argument (czyli znaleziono wolne, prawidłowe miejsce dla danej wartości) tworzy go. jeśli wartość wartości dodawanej do drzewa jest mniejsza od wartości podanego węzła, to do lewego syna danego węzła przypisywany jest wynik kolejnego wywołania *_insert_node()* z tym lewym synem jako argumentem, a jeśli wartość podana jest większa niż wartość sprawdzanego węzła, to do prawego syna węzła przypisywany jest wynik kolejnego wywołania *_insert_node()* z prawym synem jako argumentem. Po wstawieniu węzła i powrocie z funkcji wywołanej rekurencyjnie, aktualizowana jest wysokość każdego węzła, przez który przeszła funkcja podczas działania (z użyciem funkcji *_get_height()*). Następnie wykonywane jest rebalansowanie. Jeżeli lewe poddrzewo węzła jest większe o więcej niż jeden poziom od prawego (*balance > 1*) i wartość wstawianego węzła jest mniejsza od wartości lewego syna węzła to przeprowadzana jest rotacja w prawo (*_rotate_right()*), powodująca, że lewy syn węzła staje się korzeniem poddrzewa, a węzeł staje się prawym synem tego korzenia. Jeśli *balance* jest większy od 1 a wstawiany węzeł ma wartość większa od wartości lewego syna węzła w argumencie, to najpierw do lewego syna przypisywany jest wynik rotacji w lewo (*_rotate_left()*) a potem do węzła wynik rotacji w prawo. Jeżeli wysokość prawego poddrzewa jest wieksza od lewego o ponad 1 (*balance < -1*) a wstawiana wartość jest większa od prawego syna węzła podanego w argumencie to wykonywana jest rotacja w lewo skutkująca tym ze korzeniem poddrzewa staje się prawy syn a węzeł z argumentu staje się lewym synem nowego korzenia. Jeśli *balance < -1* i wstawiany węzeł ma wartość mniejsza od prawego syn węzła podanego w argumencie to na prawym synie wykonywana jest rotacja w prawo a węźle z argumentu rotacja w lewo.

3. Wykres
    1. Porównanie szybkości tworzenia drzewa z losowych wartości z przedziału (1, 30000)



    2. Porównanie szybkości tworzenia drzewa z uporządkowanych wartości od 1 do 10000



2. Znajdowanie

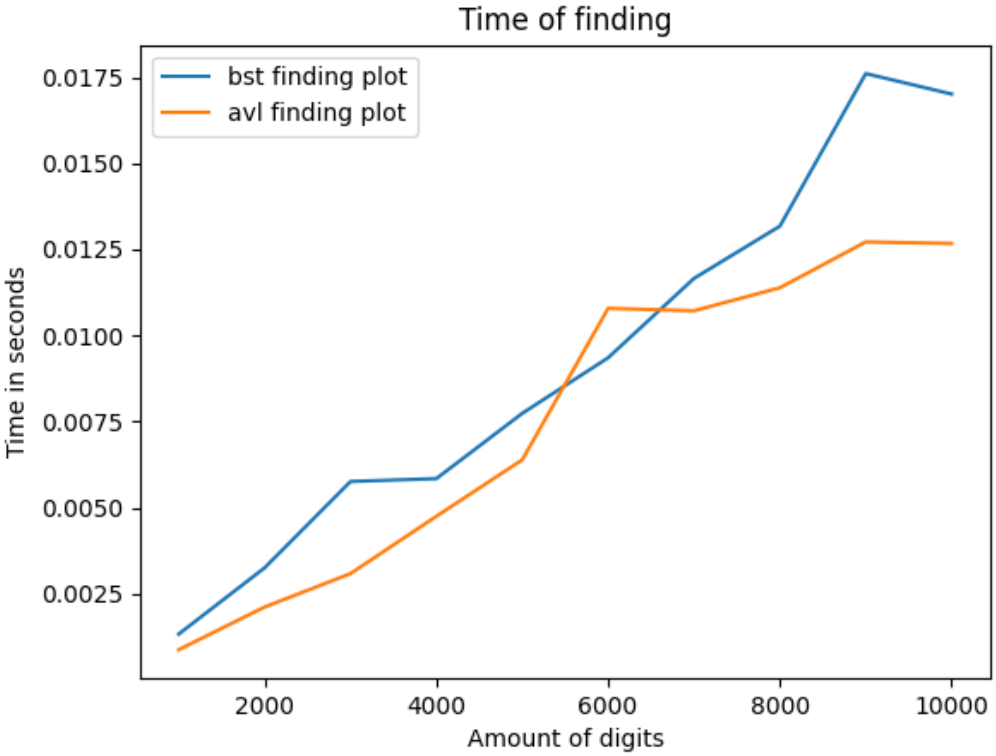
1. BST

Znajdowanie węzła o danej wartości w drzewie BST polega na przechodzeniu drzewa w pętli *while,* pamiętając poprzedni i aktualny sprawdzany węzeł, dopóki nie natrafi się na brak węzła. Jeżeli szukana wartość jest mniejsza od wartości obecnego węzła, to szukana jest dalej w jego lewym poddrzewie, jeśli większa to w prawym, a jeżeli równa to zwracany jest rodzic i szukany węzeł. Gdyby się tak zdarzyło, że węzeł o szukanej wartości nie znajduje się w drzewie to po wyjściu z pętli *while node*, zwracane są wartości None, None.

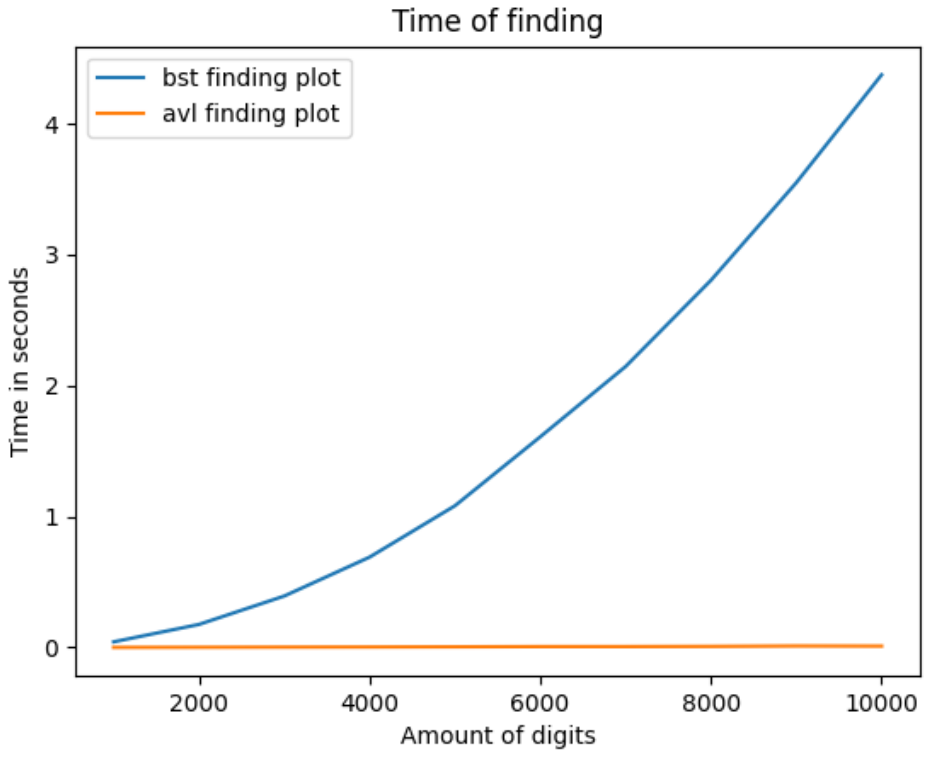
2. AVL

W drzewie AVL zastosowano ten sam algorytm znajdowania co w drzewie BST.

3. Wykres
    1. Porównanie szybkości znajdowania losowych wartości z przedziału (1, 30000)



    2. Porównanie szybkości znajdowania uporządkowanych wartości od 1 do 10000



3. Usuwanie

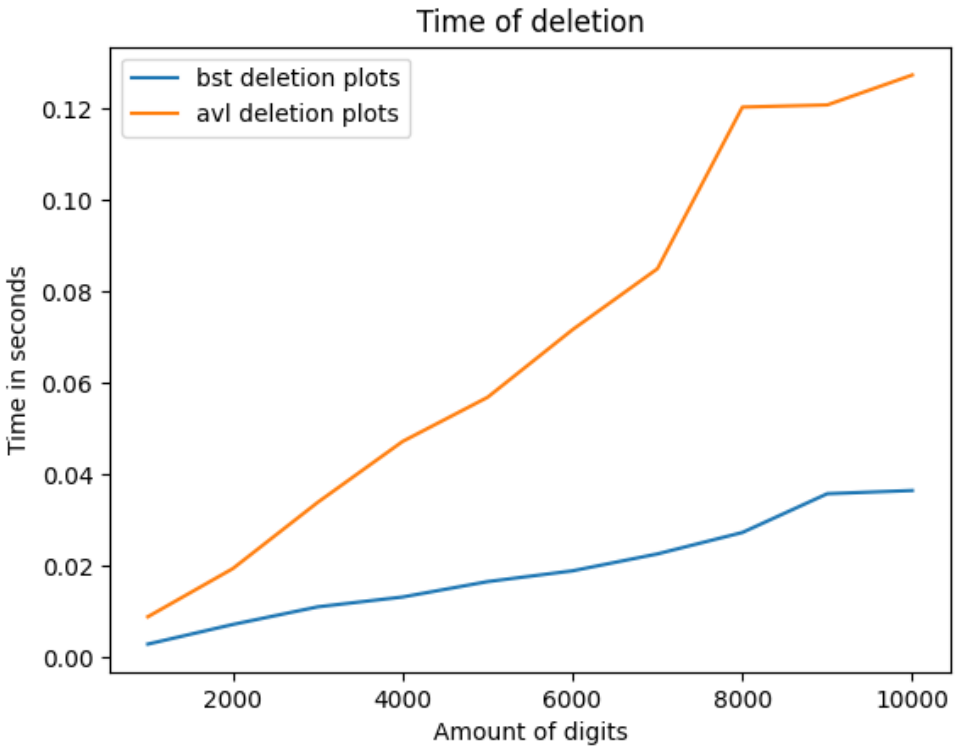
1. BST

Usuwanie węzła w drzewie BST (funkcją *delete_node()*) polega na przypisaniu do korzenia węzła, który jest wynikiem wykonania funkcji *_delete_node()*. W praktyce oznacza to rekurencyjne szukanie wartości węzła do usunięcia. Czyli jeśli nie ma takiego węzła to zwracany jest ten węzeł. Jeśli wartość szukanego węzła to do lewego syna danego węzła przypisywany jest wynik działania funkcji *_delete_node()* z lewym synem jako argumentem. Analogicznie jeżeli usuwany węzeł ma mieć większą wartość od obecnego to do prawego drzewa przypisywana wartość zwracana przy wywołaniu funkcji *_delete_node()* z prawym synem obecnego węzła. Po znalezieniu węzła do usunięcia, jeśli nie ma on lewego syna to do *temp_node* zapamiętywana jest wartość prawego syna, a gdy nie ma prawego syna to zapamiętywany jest lewy. Następnie do obecnego węzła przypisywana jest wartość None i zwracany jest *temp_node*. Czyli jeżeli węzeł nie miałby żadnego syna to wartość oryginalnego węzła zostałaby zmieniona na None, a tak to będzie to odpowiednio prawy lub lewy syn usuwanego węzła. Jeżeli węzeł do usunięcia ma dwóch synów to do jego wartość jest ustawiana na wartość jego najmniejszego następnika (znajdowanego funkcją *_get_node_with_minimal_value()*), a z jego prawego poddrzewa usuwany jest najmniejszy następnik.

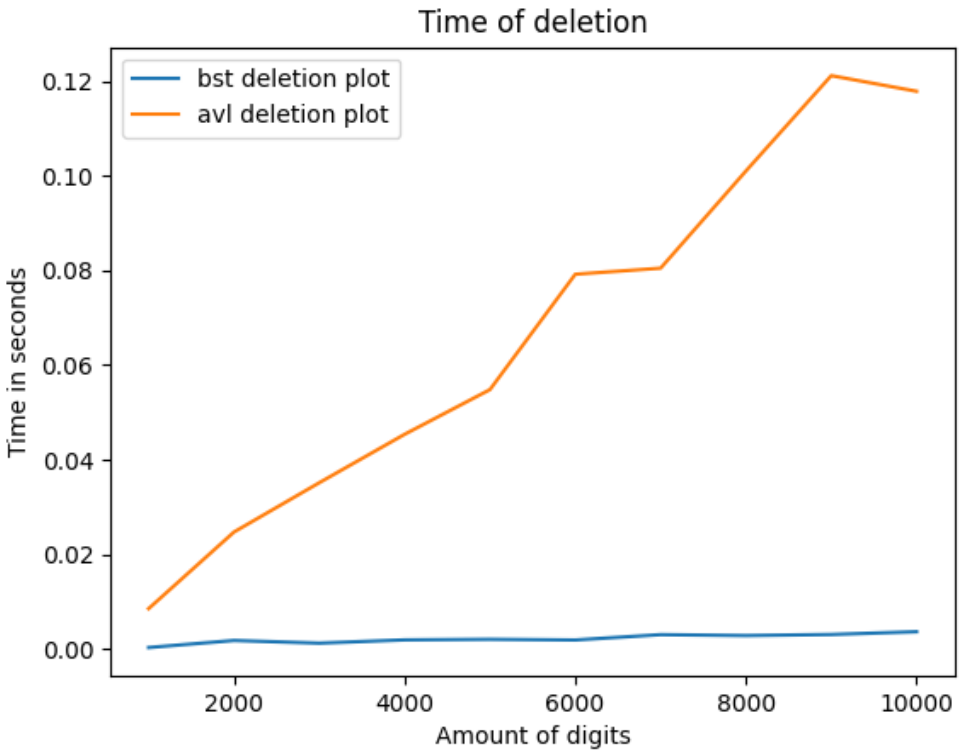
2. AVL

W drzewie AVL zastosowano ten sam mechanizm usuwania co w drzewie BST a następnie algorytm rebalansowania, który był już opisany przy okazji tworzenia drzewa.

3. Wykres
    1. Porównanie szybkości usuwania losowych wartości z przedziału (1, 30000)



    2. Porównanie szybkości usuwania uporządkowanych wartości od 1 do 10000



3. Wnioski:

Do drzewa BST zazwyczaj szybciej dodawane i usuwane są wartości, a wyszukiwanie jest minimalnie szybsze w przypadku drzewa AVL. Gdy dodawane wartości węzłów są uporządkowane, szybciej powstaje i wykonuje wyszukiwanie drzewo AVL. Jest to spowodowane tym, że zazwyczaj rebalansowanie drzewa AVL trwa, dłużej niż wyszukiwanie węzła do usunięcia lub do którego należy połączyć nowy węzeł, a w przypadku pesymistycznym odwrotnie. Można by się spodziewać, że usuwanie w tym przypadku trwało by też krócej w tym przypadku, ale wykresy wskazują inaczej. Może być to spowodowane tym, że usuwane węzły są usuwane również w kolejności uporządkowanej, więc węzeł do usunięcia jest od razu korzeniem.