Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Ewelina Matusiewicz

Nr albumu: 1065577

Implementacja drzew binarnych w języku Python

Praca magisterska na kierunku Informatyka

Praca wykonana pod kierunkiem
dra hab. Andrzeja Kapanowskiego
Instytut Fizyki

Kraków 2017

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Kraków, dnia Podpis autora pracy

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Kraków, dnia Podpis kierującego pracą

Pragnę złożyć serdeczne podziękowania Panu doktorowi habilitowanemu Andrzejowi Kapanowskiemu za poświęcony czas oraz nieocenioną pomoc, bez której praca ta nie powstałaby.

Streszczenie

W pracy przedstawiono implementację w języku Python wybranych rodzajów drzew binarnych. Opisano podstawowe operacje na binarnych drzewach poszukiwań, takie jak dodawanie nowego węzła, usuwanie węzła, wyszukiwanie elementu o danym kluczu, wyszukiwanie elementu o największym lub najmniejszym kluczu, wyszukiwanie następnika lub poprzednika danego elementu. Pokazano najważniejsze metody przechodzenia przez drzewo: inorder, preorder, postorder, poziomami.

Binarne drzewa poszukiwań są podstawową strukturą używaną do zaimplementowania zbiorów dynamicznych, których elementy mają być wyszukiwane przez swój klucz. Operacje wyszukiwania są wydajne dla drzew zrównoważonych, ale nie zawsze tak musi być. Z tego powodu w pracy przedstawiono zmodyfikowane drzewa, z usprawnieniami poprawiającymi wydajność podstawowych operacji. Drzewa czerwono-czarne i drzewa AVL utrzymują się w postaci w przybliżeniu zrównoważonej dzięki dodatkowemu atrybutowi dołączonemu do węzła. Drzewa splay natomiast poprawiają dostęp do najczęściej używanych elementów przez przemieszczenie ich bliżej korzenia.

W pracy został również zaprezentowany algorytm DSW, który każde binarne drzewo poszukiwań wydajnie sprowadza do postaci zrównoważonej. Dla czterech wymienionych rodzajów drzew przygotowano kod zarówno w podejściu obiektowym (hierarchie klas drzewowych), jak i w podejściu z funkcjami (zestawy funkcji przypominających pseudokod). Cały kod został pokryty testami jednostkowymi, dodano także testy porównujące zachowanie różnych rodzajów drzew.

Słowa kluczowe: binarne drzewo poszukiwań, drzewo czerwono-czarne, drzewo AVL, drzewo splay, algorytm DSW, rotacje drzewa, przechodzenie drzewa

English title: Python implementation of binary trees

Abstract

Python implementation of selected binary trees is presented. Basic operations supported by binary search trees are described: insertion of a node, deletion of a node, searching for a specific key, finding minimum or maximum node, finding the successor or the predecessor of a current node. Main tree traversal methods are shown: inorder, preorder, postorder, level-order.

Binary search trees are basic structures used to implement dynamic sets of items that allow finding an item by its key. Searching is efficient only for balanced trees but for real data a tree may become unbalanced. That is why three modified kinds of trees are also shown. Red-black trees and AVL trees use an additional node attribute to ensure that trees remain approximately balanced. Splay trees improve access to frequently used nodes, they are placed near the root of the tree.

The DSW algorithm is also presented, because this is a method for efficiently balancing binary search trees. Two implementations for every kind of binary search tree were prepared. In an object-oriented approach a tree class hierarchy is created. In a procedural approach a set of functions similar to pseudocode was used. All source code has been covered by unit test scripts. At the end a comparison between different types of binary search trees is presented which may be useful for the reader to decide what type of trees are best in specific situations.

Keywords: binary search tree, red-black tree, AVL tree, splay tree, DSW algorithm, tree rotations, tree traversal

Spis treści

Spis tabel

Spis rysunków

Listings

1. Wstęp

Drzewa binarne są jedną z podstawowych struktur danych spotykanych w informatyce, obok tablic i list powiązanych. Służą do reprezentowania i przetwarzania *zbiorów dynamicznych*, czyli zbiorów, które mogą zmieniać się w czasie. Element zbioru dynamicznego jest reprezentowany przez węzeł drzewa. Węzeł drzewa posiada wyróżniony atrybut, nazywany *kluczem*, który służy do wyszukiwania elementów. Oprócz klucza węzeł może zawierać dodatkowe dane związane z kluczem, oraz łącza do innych węzłów albo dodatkowe pomocnicze atrybuty.

Celem pracy jest implementacja w języku Python [1] wybranych rodzajów drzew binarnych. Pierwszym powodem jest zastosowanie powstałego kodu w edukacji. Język Python pozwala tworzyć czytelny kod obiektowy, który jest nazywany wykonywalnym pseudokodem. Można objaśniać konstrukcję drzew używając kodu w Pythonie, a ponadto kod można wykonać i badać efekty jego działania. Drugim powodem jest wygoda i szybkość użycia Pythona w prototypowaniu, w tworzeniu eksperymentalnego kodu do nowych zastosowań. Drzewa binarne często są podstawą do tworzenia nowych struktur danych, przez dodanie nowych operacji i dodatkowych atrybutów. Jest to nazywane *wzbogacaniem struktur danych* do nowych zastosowań. Nie zawsze jest to łatwe, ponieważ nie można pogorszyć wydajności standardowych operacji na drzewie binarnym [2], [3].

Drzewa binarne mają bardzo szerokie zastosowanie w informatyce, jak i poza tą dziedziną. Mogą być używane między innymi do indeksowania pól baz danych [4], [5], oraz przy każdym innym zastosowaniu, w którym dane są często wyszukiwane i przydaje się szybki dostęp do elementów, w algorytmach sortowania [6], a także parserach [7], [8]. Procesy decyzyjne, przebiegi dowolnych działań mogą zostać przedstawione za pomocą drzewa binarnego, gdyż każdy węzeł na ścieżce od korzenia do liści może służyć do podjęcia pewnego wyboru [9]. W formatach bezstratnej kompresji obrazu i dźwięku, np. JPEG czy MP3 powszechnie stosuje się kody Huffmana [10], których wyznaczanie bazuje na drzewie binarnym. Drzewa binarne są wydajne, w bibliotece standardowej C++ takie struktury jak mapy czy zbiory opierają się na implementacji drzew czerwono-czarnych [11]. Ciekawym i użytecznym zastosowaniem drzew binarnych w grafice komputerowej jest drzewo BSP, które służy do podziału płaszczyzny i opiera się na podobnych zasadach jak drzewo BST. Struktura ta pozwala w stosownie szybkim czasie narysować wszystkie ściany dla zamkniętych obszarów w scenerii gry komputerowej tak, aby były widoczne dla obserwatora [12], [13].

Treść pracy jest zorganizowana w następujący sposób. Rozdział 1 zawiera krótkie wprowadzenie do tematu drzew binarnych, cele pracy i przykłady zastosowań drzew binarnych. Rozdział 2 wprowadza podstawowe pojęcia i

operacje stosujące się do dowolnych drzew binarnych. W rozdziale 3 opisane zostały binarne drzewa poszukiwań. Procedury z tego rozdziału są wykorzystane w większości funkcji zaprezentowanych w dalszych rozdziałach, opisujących zmodyfikowane binarne drzewa poszukiwań. Rozdział 4 przedstawia podstawowe funkcje w drzewach czerwono-czarnych. W rozdziale 5 zawarte zostały implementacje stosowane w drzewach AVL. W rozdziale 6 znajduje się opis drzew splay. Rozdział 7 przedstawia podsumowanie całej pracy. W dodatku A zostały zebrane klasy dla węzłów drzew, a mianowicie podstawowy węzeł użyty w implementacji drzew BST oraz splay, a także klasy węzłów drzewa RBT oraz AVL. Dodatek B zawiera porównanie operacji stosowanych w poszczególnych drzewach pod kątem czasowym w zależności od liczby węzłów, a także zestawienie wysokości zbudowanych drzew.

2. Drzewa binarne

W tym rozdziale przedstawimy założenia implementacyjne drzew binarnych, oraz zestaw funkcji działających dla dowolnych drzew binarnych.

2.1. Implementacja drzew

Podstawowy węzeł drzewa binarnego reprezentuje klasa Node. Oprócz łączy do rodzica (node.parent), lewego dziecka (node.left) i prawego dziecka (node.right), zawiera atrybut node.data, który jest kluczem wyszukiwania. Węzły są przygotowywane poza drzewami i mogą zawierać jeszcze inne dane związane z kluczem.

Specyficzne drzewa binarne mogą korzystać z dodatkowych atrybutów. Przez dziedziczenie z klasy Node otrzymujemy klasy węzłów dla innych drzew. Wszystkie klasy dla węzłów przechowywane są w module treenodes. Kod źródłowy klas reprezentujących węzły drzew binarnych przedstawiono w dodatku A.

2.2. Przechodzenie przez drzewo

Drzewo binarne nie jest strukturą liniową jak listy, czy macierze jednowymiarowe, dlatego po węzłach drzewa można się przemieszczać na wiele sposobów. Najważniejsze sposoby przechodzenia przez drzewo binarne (ang. *tree traversal*) są następujące [14]:

— Metoda *preorder* (węzeł, lewe, prawe).
— Metoda *inorder* (lewe, węzeł, prawe).
— Metoda *postorder* (lewe, prawe, węzeł).
— Metoda przechodzenia poziomami (ang. *level-order*).

Listing 2.1 przedstawia rekurencyjne funkcje realizujące przechodzenie przez drzewo trzema pierwszymi metodami. Funkcje zwracają odpowiednie listy kluczy odwiedzanych elementów.

Listing 2.1. Przechodzenie drzewa binarnego.

```
def btree_inorder(top):
    if top is None:
        return []
    order = []
    order.extend(btree_inorder(top.left))
    order.append(top.data)
    order.extend(btree_inorder(top.right))
    return order
```

```
def btree_preorder(top):
    if top is None:
        return []
    order = []
    order.append(top.data)
    order.extend(btree_preorder(top.left))
    order.extend(btree_preorder(top.right))
    return order

def btree_postorder(top):
    if top is None:
        return []
    order = []
    order.extend(btree_postorder(top.left))
    order.extend(btree_postorder(top.right))
    order.append(top.data)
    return order
```

Przechodzenie przez drzewo metodą *preorder* można zrealizować nierekurencyjnie za pomocą stosu. Z kolei zamiana stosu na kolejkę pozwala uzyskać przechodzenie poziomami.

2.3. Wyznaczanie parametrów drzewa binarnego

Drzewa binarne mają strukturę rekurencyjną, dlatego algorytmy rekurencyjne mają z reguły bardzo prostą postać. Z drugiej strony, dla dużych drzew przy algorytmach rekurencyjnych istnieje zagrożenie przekroczenia limitu głębokości rekurencji.

Listing 2.2. Liczba węzłów drzewa binarnego.

```
def btree_count(top):
    if top is None:
        return 0
    return btree_count(top.left) + 1 + btree_count(top.right)
```

Listing 2.3. Wysokość drzewa binarnego.

```
def btree_height(top):
    if top is None:
        return 0
    left = btree_height(top.left)
    right = btree_height(top.right)
    return 1 + max(left, right)
```

Głębokość danego wierzchołka definiujemy jako liczbę kroków na ścieżce do korzenia drzewa. Wykorzystywane jest łącze do rodzica parent.

Listing 2.4. Głębokość wierzchołka w drzewie binarnym.

```
def btree_depth(node):
    depth = 0
    while node.parent:
        node = node.parent
        depth += 1
    return depth
```

2.4. Rotacje drzewa

Na dowolnym drzewie binarnym można wykonać operację rotacji drzewa, która zmienia lokalną strukturę drzewa. Zwykle o rotacjach mówi się w kontekście drzew BST, dlatego omówienie rotacji znajduje się w rozdziale 3.

2.5. Korzystanie z drzew binarnych

Przykładowa sesja interaktywna.

```
>>> from bst import BinarySearchTree
>>> from rbt import RedBlackTree
>>> from avl import AVLTree
>>> from splay import SplayTree
>>> from treenodes import *
>>> T = BinarySearchTree()
>>> for x in [5, 25, 10, 50, 40, 30, 15, 20, 70, 90]:
        T.insert(Node(x))
>>> print T          # wypisanie drzewa
            90
        70
    50
        40
            30
25
            20
        15
    10
5

>>> T.iterative_search(70)
Node(70)
>>> root = T.iterative_search(5)
>>> print root
5
>>> T.find_min(root)
Node(5)
>>> T.find_max(root)
Node(90)
>>> T.preorder(root)
[5, 25, 10, 15, 20, 50, 40, 30, 70, 90]
>>> T.inorder(root)
[5, 10, 15, 20, 25, 30, 40, 50, 70, 90]
>>> T.postorder(root)
[20, 15, 10, 30, 40, 90, 70, 50, 25, 5]
>>> T.tree_height(root)
5
>>> node = T.search(root, 25)
>>> print node
25
>>> T.delete(node)
Node(30)
>>> print T
            90
        70
    50
```

```
            40
    30
                20
            15
        10
5
>>> n = T.number_of_nodes()
>>> print n
9
>>> T.create_perfect_tree(n)
>>> print T
        90
    70
        50
40
        30
    20
            15
        10
            5

>>> T = AVLTree()
>>> T.insert(AVLNode(3))
AVLNode(3, balance=0)
>>> T.insert(AVLNode(9))
AVLNode(3, balance=1)
>>> T.insert(AVLNode(7))
AVLNode(7, balance=0)
>>> T.insert(AVLNode(1))
AVLNode(7, balance=-1)
>>> T.insert(AVLNode(2))
AVLNode(7, balance=-1)
>>> T.insert(AVLNode(8))
AVLNode(7, balance=0)
>>> T.insert(AVLNode(6))
AVLNode(7, balance=-1)
>>> T.insert(AVLNode(4))
AVLNode(7, balance=-1)
>>> T.insert(AVLNode(5))
AVLNode(7, balance=-1)
>>> print T
    9(-1)
        8(0)
7(-1)
        6(-1)
            5(0)
    4(0)
            3(0)
        2(0)
            1(0)

>>> T.iterative_search(3)
AVLNode(3, balance=0)
>>> node = T.iterative_search(3)
>>> T.delete(node)
AVLNode(4, balance=0)
>>> print T
```

```
    9(-1)
        8(0)
7(-1)
        6(-1)
            5(0)
    4(0)
        2(-1)
            1(0)
```

```
>>> T = RedBlackTree()
>>> T.insert(RBTNode(1))
RBTNode(1, color=None)
>>> T.insert(RBTNode(2))
RBTNode(1, color=None)
>>> T.insert(RBTNode(3))
RBTNode(2, color=None)
>>> T.insert(RBTNode(4))
RBTNode(2, color=None)
>>> T.insert(RBTNode(5))
RBTNode(2, color=None)
>>> T.insert(RBTNode(6))
RBTNode(2, color=None)
>>> T.insert(RBTNode(8))
RBTNode(2, color=None)
>>> T.insert(RBTNode(9))
RBTNode(4, color=None)
>>> T.insert(RBTNode(7))
RBTNode(4, color=None)

>>> T.iterative_search(4)
RBTNode(4, color=None)
>>> node = T.iterative_search(3)
>>> T.delete(node)

>>> T = SplayTree()
>>> for x in [3, 4, 6, 7]:
        T.insert(Node(x))
>>> node = T.iterative_search(6)
>>> T.delete(node)
Node(7)
>>> print T
7
    4
        3
```

---

3. Binarne drzewa poszukiwań

Binarne drzewo poszukiwań (ang. *binary search tree, BST*) jest to drzewo binarne, w którym lewe poddrzewo każdego węzła zawiera wyłącznie elementy o kluczach mniejszych niż klucz węzła, zaś prawe poddrzewo zawiera wyłącznie elementy o kluczach nie mniejszych niż klucz węzła [15].

Następujące operacje dla drzewa BST mają złożoność czasową $O(h)$, gdzie h to wysokość drzewa: wyszukiwanie dowolnego klucza, wyszukiwanie najmniejszego klucza, wyszukiwanie największego klucza, znajdowanie następnika, znajdowanie poprzednika [2]. *Drzewo zrównoważone* (ang. *balanced tree*) ma najmniejszą możliwą wysokość $h \sim \log_2 n$, a wtedy operacje wyszukiwania lub wstawiania danych zajmują czas $O(\log n)$ (tabela 3.1).

3.1. Wyszukiwanie dowolnego klucza w drzewie

Listing 3.1 przedstawia rekurencyjną i iteracyjną wersję algorytmu wyszukiwania dowolnego klucza w drzewie BST [15]. Funkcje zwracają węzeł zawierający poszukiwany klucz lub wartość None.

Listing 3.1. Wyszukiwanie dowolnego klucza w drzewie BST.

```
def bst_search(root, data):
    if root is None or root.data == data:
        return root, root
    elif data < root.data:
        _, node = bst_search(root.left, data)
        return root, node
    else:   # data > top.data
        _, node = bst_search(root.right, data)
        return root, node

def bst_iterative_search(root, data):
    top = root
    while top  and top.data != data:
        if data < top.data:
            top = top.left
        else:       # data > top.data
            top = top.right
```

Tabela 3.1. Złożoność obliczeniowa drzew BST.

BST	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$

```
    return root, top
```

3.2. Wyszukiwanie najmniejszego i największego klucza w drzewie

Funkcje zwracają węzeł zawierający poszukiwany klucz lub rzucają wyjątek ValueError dla pustego drzewa [15].

Listing 3.2. Wyszukiwanie najmniejszego klucza w drzewie BST.

```
def bst_find_min(top):
    if top is None:
        raise ValueError("empty tree")
    while top.left:
        top = top.left
    return top
```

Listing 3.3. Wyszukiwanie największego klucza w drzewie BST.

```
def bst_find_max(top):
    if top is None:
        raise ValueError("empty tree")
    while top.right:
        top = top.right
    return top
```

3.3. Wyszukiwanie następnika i poprzednika w drzewie

Następnik (ang. *successor*) danego węzła jest węzłem, który jest odwiedzany jako następny w przypadku przechodzenia drzewa metodą *inorder* [15]. *Poprzednik* (ang. *predecessor*) danego węzła jest węzłem, który był odwiedzany jako poprzedni w przypadku przechodzenia drzewa metodą *inorder* [15]. W celu wyznaczenia następnika lub poprzednika danego węzła w drzewie BST nie trzeba przeprowadzać żadnych porównań kluczy.

Przy wyznaczaniu następnika danego węzła mogą wystąpić dwa przypadki:

1. Istnieje prawe poddrzewo danego węzła. Wtedy następnikiem jest najmniejszy węzeł w prawym poddrzewie.
2. Nie istnieje prawe poddrzewo danego węzła. Wtedy następnikiem jest węzeł będący najniższym (w sensie wysokości w drzewie) przodkiem węzła odniesienia, dla którego dany węzeł leży w lewym poddrzewie.

Analogiczne dwa przypadki wystepują przy wyznaczaniu poprzednika danego wezła, ale wtedy sprawdzane jest istnienie lewego poddrzewa danego węzła. Funkcje wyznaczające następnika i poprzednika korzystają z łącza do rodzica.

Listing 3.4. Wyszukiwanie następnika w drzewie BST.

```
def bst_find_successor(root, node):
    if root is None or node is None:
        return None
```

```
    if node.right:
        return bst_find_min(node.right)
    # Nie istnieje prawe poddrzewo dla node.
    successor = node.parent
    while successor and node == successor.right:
        node = successor
        successor = successor.parent
    return successor
```

Listing 3.5. Wyszukiwanie poprzednika w drzewie BST.

```
def bst_find_predecessor(root, node):
    if root is None or node is None:
        return None
    if node.left:
        return bst_find_max(node.left)
    # Nie istnieje lewe poddrzewo dla node.
    predecessor = node.parent
    while predecessor and node == predecessor.left:
        node = predecessor
        predecessor = predecessor.parent
    return predecessor
```

3.4. Wstawianie węzła

Nowe węzły wstawiane do drzewa BST będą liśćmi tego drzewa [15]. Zwracany jest nowy korzeń poddrzewa. W naszej implementacji nowy węzeł jest tworzony poza drzewem BST instrukcją new_node = Node(data), gdzie data będzie kluczem wyszukiwania.

Listing 3.6. Wstawianie elementu do drzewa BST.

```
def bst_insert_recursively(root, top, new_node):
    if root is None: # puste drzewo, koniec rekurencji
        # Czyszczenie lacz dla new_node.
        new_node.parent = None
        new_node.left = None
        new_node.right = None
        return new_node, new_node
    if new_node.data < top.data:
        if top.left:
            root, _ = bst_insert_recursively(root, top.left, new_node)
        else: # koniec rekurencji
            top.left = new_node
            new_node.parent = top
            new_node.left = None
            new_node.right = None
    else:
        if top.right:
            root, _ = bst_insert_recursively(root, top.right, new_node)
        else: # koniec rekurencji
            top.right = new_node
            new_node.parent = top
            new_node.left = None
            new_node.right = None
```

```
    return root, new_node

def bst_insert_iteratively(root, new_node):
    parent = None
    node = root
    while node:
        parent = node
        if new_node.data < node.data:
            node = node.left
        else:        # wieksze lub rowne na prawo
            node = node.right
    # new_node wstawiamy ponizej parent.
    if parent:       # w srodku drzewa
        if new_node.data < parent.data:
            parent.left = new_node
        else:
            parent.right = new_node
    else:            # puste drzewo
        root = new_node
    # Czyszczenie lacz dla new_node.
    new_node.parent = parent
    new_node.left = None
    new_node.right = None
    return root, new_node
```

3.5. Usuwanie węzła

Przy usuwaniu danego węzła z drzewa BST należy wyróżnić trzy przypadki [2].

1. Dany węzeł nie ma synów. Wtedy węzeł można bez problemu usunąć, w jego ojcu łącze ustawiamy na None.
2. Dany węzeł ma jednego syna. Wtedy podnosimy syna na pozycję danego węzła, aktualizując łącze w ojcu usuwanego węzła.
3. Dany węzeł node ma dwóch synów. Jeżeli węzeł node.right nie ma lewego syna, to podnosimy node.right na miejsce node. Jeżeli węzeł node.right ma lewego syna, to w tej gałęzi znajduje się następnik węzła node. Musimy zastąpić następnik przez jego prawego syna, a potem zastąpić node przez następnik.

W implementacji wykorzystana jest funkcja bst_transplant(), która wstawia jedno poddrzewo w miejsce drugiego w jego ojcu [2].

Listing 3.7. Usuwanie elementu z drzewa BST.

```
def bst_transplant(root, first, second): # zwraca (root, second)
    # Zastepowanie poddrzewa first przez poddrzewo second.
    # Nie ma tu aktualizacji second.left i second.right.
    if first.parent is None:
        root = second
        if root:
            root.parent = None
        return root, second
    elif first == first.parent.left:
```

```
        first.parent.left = second
    else:
        first.parent.right = second
    if second:              # second moze byc None
        second.parent = first.parent
    return root, second

def bst_delete(root, node):
    # Zwraca root i wezel, ktory przesunie sie na miejsce node.
    if root is None or node is None:
        return root, None
    if node.left is None:
        return bst_transplant(root, node, node.right)
    elif node.right is None:
        return bst_transplant(root, node, node.left)
    else:               # node.left i node.right niepuste
        y = bst_find_min(node.right)
        if y.parent != node:
            root, _ = bst_transplant(root, y, y.right)
            y.right = node.right
            if y.right:
                y.right.parent = y
        root, _ = bst_transplant(root, node, y)
        y.left = node.left
        y.left.parent = y
        return root, y
```

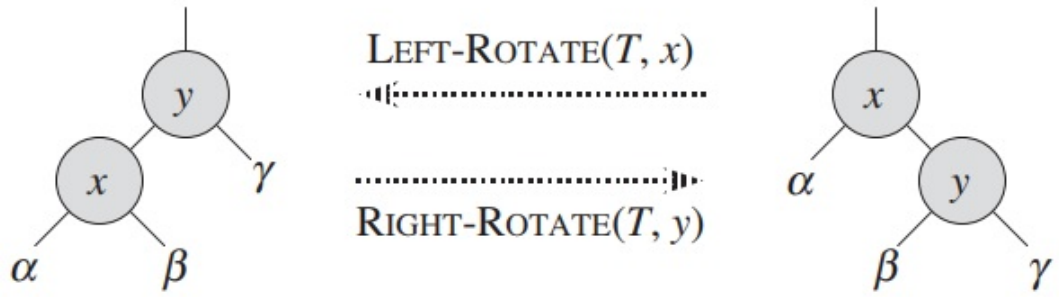
3.6. Rotacje drzewa

Rotacja drzewa (ang. *tree rotation*) jest operacją na drzewie BST, która zmienia jego lokalną strukturę bez zmiany kolejności elementów, przy przechodzeniu przez drzewo BST metodą *inorder* [16]. Wyróżnia się dwie symetryczne operacje: rotacja w prawo (listing 3.8) i rotacja w lewo (listing 3.9). Obie funkcje zwracają korzeń drzewa, nowy korzeń poddrzewa i działają w stałym czasie. Rotacje drzewa są wykorzystywane w drzewach czerwono-czarnych, drzewach AVL, drzewach splay i drzewcach (ang. *treaps*).

Jeżeli w danej implementacji węzeł nie posiada łącza do rodzica, to z podanych funkcji należy usunąć instrukcje korzystające z atrybutu parent. Ponadto należy upewnić się, że ojciec wierzchołka top, jeśli istnieje, wskazuje po rotacji na nowy korzeń poddrzewa.

Listing 3.8. Rotacja w prawo w drzewie binarnym.

```
def rotate_right(root, top):
    if top.left is None:    # nie ma rotacji
        return root, top
    node = top.left
    top.left = node.right
    if node.right:
        node.right.parent = top
    node.parent = top.parent
    if top.parent is None:
        root = node    # top byl korzeniem
    elif top == top.parent.right:
```



Rysunek 3.1. Przedstawione z lewej strony poddrzewo pod wpływem wykonania operacji prawej rotacji na węźle y przekształca się w poddrzewo przedstawione z prawej strony. Z kolei poddrzewo z prawej strony, po poddaniu węzła x operacji lewej rotacji, przekształca się w poddrzewo z lewej strony. α, β, γ symbolizują dowolne poddrzewa [2].

```
        top.parent.right = node
    else:
        top.parent.left = node
    node.right = top
    top.parent = node
    return root, node
```

Listing 3.9. Rotacja w lewo w drzewie binarnym.

```
def rotate_left(top):
    if top.right is None:    # nie ma rotacji
        return root, top
    node = top.right
    top.right = node.left
    if node.left:
        node.left.parent = top
    node.parent = top.parent
    if top.parent is None:
        root = node    # top byl korzeniem
    elif top == top.parent.left:
        top.parent.left = node
    else:
        top.parent.right = node
    node.left = top
    top.parent = node
    return root, node
```

3.7. Algorytm DSW

Drzewo BST można doprowadzić do postaci zrównoważonej za pomocą algorytmu DSW (Day, Stout, Warren) [17]. Wtedy wysokość drzewa zmniejsza się do wartości $O(\log n)$. Algorytm wykorzystuje rotacje drzewa, które nie zmieniają kolejności odwiedzania węzłów przy przechodzeniu drzewa metodą *inorder*.

W literaturze wymienia się dwa powody, dla których warto pamiętać o algorytmie DSW [17]. Po pierwsze, algorytm jest przydatny w sytuacji, gdy na początku przetwarzania buduje się całe drzewo BST, a następnie jest tylko faza wyszukiwania kluczy. Po drugie, podczas kursu struktur danych, algorytm DSW jest naturalnym ogniwem przy przejściu od zwykłych drzew BST do drzew samoorganizujących się, gdzie pojawiają się rotacje drzewa.

Dane wejściowe: Drzewo BST.

Problem: Zrównoważenie drzewa BST.

Opis algorytmu: Algorytm w pierwszej fazie zamienia drzewo w listę (kręgosłup) przez wielokrotne rotacje prawe. W drugiej fazie algorytm przywraca kształt drzewa przez wielokrotne rotacje lewe.

Złożoność: Złożoność czasowa algorytmu wynosi $O(n)$. Dodatkowe zasoby pamięci potrzebne do wykonania algorytmu są stałe. Wnioskujemy, że algorytm DSW jest optymalny ze względu na czas i na zużytą pamięć.

Listing 3.10. Moduł dsw.

```
#!/usr/bin/python
from math import pow, floor, log
def bst_create_backbone(root, top):
    parent = top
    left_child = None
    while parent:
        left_child = parent.left
        if left_child:
            root, _ = bst_rotate_right(root, parent)
            parent = left_child
        else:
            parent = parent.right
    return root

def bst_create_perfect_tree(root, n):
    """Time complexity of DSW algorithm: O(n) """
    root = bst_create_backbone(root, root)
    m = int(pow(2, floor(log(n+1, 2))) - 1)
    root = bst_make_rotations(root, n-m)
    while m > 1:
        m = m // 2 # jesli chcemy iterowac w make_rotations po m,
                   # to m musi byc calkowite
        root = bst_make_rotations(root, m)
    return root

def bst_make_rotations(root, x):
    p = root
    for i in range(x):
        if p:
            root, _ = bst_rotate_left(root, p)
            if p.parent:
                p = p.parent.right
    return root
```

4. Drzewa czerwono-czarne

Drzewo czerwono-czarne (ang. *red-black tree, RBT*) [18] to binarne drzewo poszukiwań, którego węzeł to podstawowy węzeł drzewa BST wzbogacony o informację o kolorze. Kolor węzła może być czerwony lub czarny, więc na informację o kolorze wystarczy przeznaczyć jeden bit. Informacja o kolorze węzła pomaga utrzymywać drzewo w postaci w przybliżeniu zrównoważonej, co z kolei prowadzi do szybkich operacji na drzewie, nawet w przypadku pesymistycznym. Oprócz własności drzew BST, drzewo czerwono-czarne musi spełniać dodatkowe *warunki czerwono-czarne* [2]:

1. Każdy węzeł jest czerwony lub czarny.
2. Korzeń jest czarny.
3. Każdy liść jest czarny [tu liść jest rozumiany jako puste łącze].
4. Jeśli węzeł jest czerwony, to jego synowie muszą być koloru czarnego.
5. Każda ścieżka z ustalonego węzła do liścia liczy tyle samo czarnych węzłów.

Operacja wyszukiwania klucza jest taka sama jak dla drzew BST. W algorytmie dodawania i usuwania węzła użyte są operacje rotacji wykorzystywane przy implementacji algorytmu DSW dla drzew BST.

W roku 2008 Sedgewick wprowadził prostszą wersję drzew RBT o nazwie LLRBT (ang. *left-leaning red-black trees*), gdzie wykorzystano pewną swobodę występującą w implementacji.

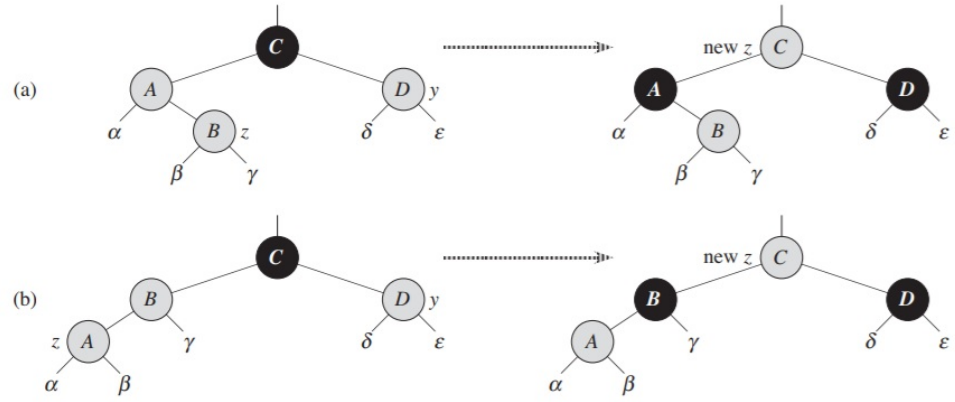
4.1. Przywrócenie własności drzewa RBT po wstawieniu węzła

W pętli rozważanych jest sześć przypadków, z których trzy są symetryczne w stosunku do pozostałych trzech [2].

Przypadek 1: Wstawiony węzeł, jego rodzic oraz wujek są koloru czerwonego. W tym przypadku należy pokolorować wujka i rodzica z na czarno, a dziadka na czerwono. W kolejnym kroku pętli przechodzimy poziom wyżej, żeby sprawdzić, czy pradziadek węzła z nie jest koloru czerwonego [2].

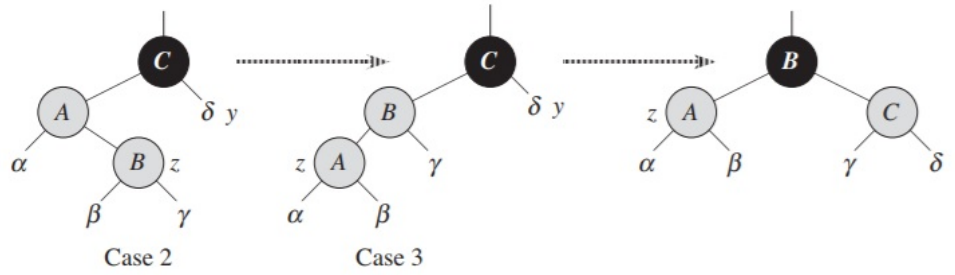
Tabela 4.1. Złożoność obliczeniowa drzew RBT.

RBT	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$



Rysunek 4.1. Przypadek 1 procedury przywracania własności drzew RBT po wstawieniu węzła: (a) z jest prawym synem; (b) z jest lewym synem [2].

Przypadki 2 i 3: Wujek z ma kolor czarny. Przypadek 2 zachodzi, gdy nowy węzeł to prawe dziecko swojego rodzica, zaś przypadek 3, gdy nowy węzeł to lewe dziecko swojego rodzica. W przypadku 2 stosujemy lewą rotację i otrzymujemy sytuację z przypadku 3. W przypadku 3 kolorujemy rodzica nowego węzła na czarno, a dziadka na czerwono i wykonujemy prawą rotację. W tym miejscu działanie pętli kończy się, gdyż kolor rodzica nowego węzła jest czarny [2].



Rysunek 4.2. Przypadki 2 i 3 procedury przywracania własności drzew RBT po wstawieniu węzła [2].

Listing 4.1. Przywrócenie własności drzewa RBT po wstawieniu węzła.

```
def rbt_insert_fixup(root, new_node):
    while new_node != root and new_node.parent.color == RED:
        # jezeli rodzic wstawionego wezla jest czarny,
        # to wlasnosc drzewa zostala zachowana - nie wchodzimy do petli

        # jezeli rodzic wstawionego wezla jest czerwony,
        # to wlasnosc nr 5 zostala naruszona - trzeba ja przywrocic
        if new_node.parent == new_node.parent.parent.left:
            uncle = new_node.parent.parent.right
            if uncle and uncle.color == RED: # Przypadek 1 - czerwony wujek
                new_node.parent.color = BLACK
                uncle.color = BLACK
                new_node.parent.parent.color = RED
```

```
                new_node = new_node.parent.parent
            else: # Przypadek 2 lub 3
                if new_node == new_node.parent.right: # Przypadek 2
                    # wujek czarny, new_node to prawe dziecko rodzica
                    new_node = new_node.parent
                    root, _ = bst_rotate_left(root, new_node)
                # Przypadek 3 - wujek czarny, new_node to
                # lewe dziecko rodzica
                new_node.parent.color = BLACK
                new_node.parent.parent.color = RED
                root, _ = bst_rotate_right(root, new_node.parent.parent)
        else: # Przypadki lustrzane
            uncle = new_node.parent.parent.left
            if uncle and uncle.color == RED: # Przypadek 1 lustrzany
                new_node.parent.color = BLACK
                uncle.color = BLACK
                new_node.parent.parent.color = RED
                new_node = new_node.parent.parent
            else: # Przypadek 2 lub 3 - lustrzane
                if new_node == new_node.parent.left:
                    # Przypadek 2 lustrzany
                    new_node = new_node.parent
                    root, _ = bst_rotate_right(root, new_node)
                # Przypadek 3 lustrzany
                new_node.parent.color = BLACK
                new_node.parent.parent.color = RED
                root, _ = bst_rotate_left(root, new_node.parent.parent)
    root.color = BLACK
    return root
```

4.2. Wstawianie węzła

Przy wstawianiu węzła do drzewa RBT wykorzystywana jest funkcja wstawiania do drzewa BST. Kolor dodanego węzła jest ustawiony na czerwony. Jeżeli rodzic wstawionego węzła jest czarny, to własności drzewa RBT zostały zachowane. W przeciwnym przypadku własność nr 4 została naruszona i należy ją przywrócić, co następuje w bst_insert_fixup.

Listing 4.2. Wstawianie elementu do drzewa RBT.

```
def rbt_insert_(root, new_node):
    # poczatkowo wstawiamy wezel jak do drzewa BST
    root, _ = bst_insert_iteratively(root, new_node)
    # kolor nowo dodanego wezla ustawiamy na czerwony
    new_node.color = RED
    # przywrocenie wlasciwosci drzewa rbt
    root = rbt_insert_fixup(root, new_node)
    return root, new_node
```

4.3. Przywrócenie własności drzewa RBT po usunięciu węzła

Procedura ta złożona z przekolorowania węzłów i rotacji ma na celu przywrócić własności drzewa czerwono-czarnego po usunięciu elementu. W pętli **while** mamy do czynienia z 8 przypadkami (4 z nich są symetryczne do 4 pozostałych w zależności od tego, czy rozważany w funkcji wezeł jest lewym czy prawym dzieckiem swojego rodzica) [2].

Przypadek 1: Jeżeli brat usuniętego węzła jest czerwony, to dokonywana jest zamiana kolorów pomiędzy węzłem brata i rodzica usuwanego węzła oraz lewa rotacja. Przypadek ten przekształca się w przypadek 2, 3 lub 4.

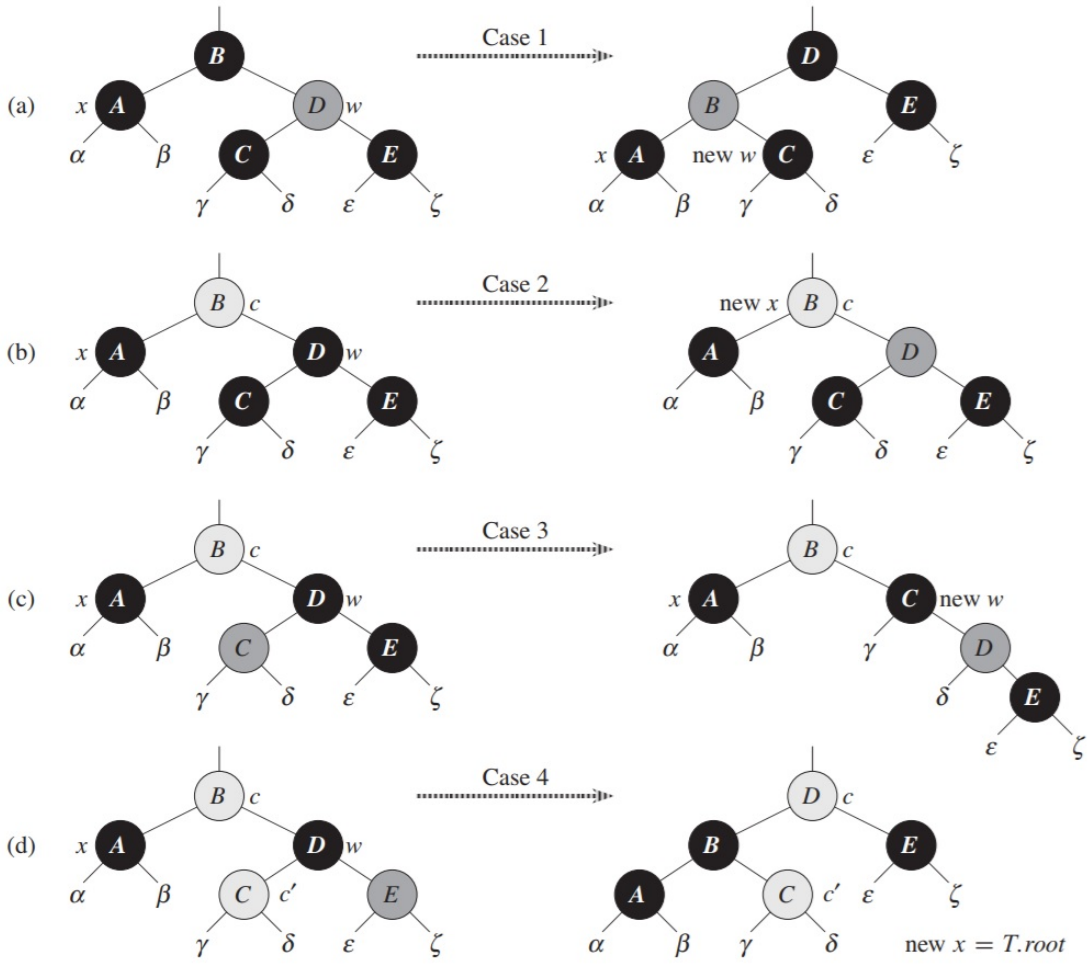
Przypadek 2: Jeżeli brat usuniętego węzła i jego synowie są koloru czarnego, to w celu naprawy własności drzewa brat jest kolorwany na czerwono. Jeśli przypadek 2 nastąpił po przejściu z przypadku 1, to pętla się kończy, gdyż rozważany węzeł kolorowany jest na czerwono i własności drzewa czerwono-czarnego są przywrócone.

Przypadek 3: Brat usuniętego węzła i jego prawy syn są koloru czarnego, a lewy syn ma kolor czerwony poprzez zamianę koloru lewego syna i usuniętego węzła oraz prawą rotację zostaje przekształcony w przypadek 4.

Przypadek 4: Brat usuniętego węzła jest czarny, a prawy syn czerwony. Poprzez przekolorwanie odpowiednich węzłów i lewą rotację własności drzewa zostają przywrócone i pętla się kończy.

Listing 4.3. Przywrócenie własności drzewa RBT po usunięciu węzła.

```
def rbt_delete_fixup(root, node, node_parent, y_is_left):
    while node != root and (node is None or node.color == BLACK):
        w = None
        if y_is_left:
            w = node_parent.right
            if w and w.color == RED:
                # Przypadek 1 - brat usunietego wezla jest czerwony
                w.color = BLACK
                node_parent.color = RED
                root, _ = bst_rotate_left(root, node_parent)
                w = node_parent.right
            if w and ((w.left is None and w.right is None) or \
               (w.left and w.left.color == BLACK and \
                w.right and w.right.color == BLACK)):
                # Przypadek 2 - brat usunietego wezla
                # i jego synowie sa czarni
                w.color = RED
                node = node_parent
            elif w:
                if w.right is None or w.right.color == BLACK:
                    # Przypadek 3 - brat usunietego wezla
                    # i jego prawy syn sa czarni, lewy syn czerwony
                    w.left.color = BLACK
                    w.color = RED
```



Rysunek 4.3. Operacja przywrócenia własności drzewa RBT. Czarne węzły są koloru czarnego, ciemnoszare - czerwonego, jasnoszare - mogą być czerwone lub czarne (kolor ten ustala się w późniejszych przypadkach/przebiegach pętli). (a) Przypadek 1; (b) Przypadek 2; (c) Przypadek 3; (d) Przypadek 4. Symetryczna sytuacja gdy x jest prawym dzieckiem i y lewym dzieckiem [2].

```
                root, _ = bst_rotate_right(root, w)
                w = node_parent.right
            # Przypadek 4 - brat usunietego wezla jest
            # czarny, prawy syn czerwony
            w.color = node_parent.color
            node_parent.color = BLACK
            if w.right:
                w.right.color = BLACK
            root, _ = bst_rotate_left(root, node_parent)
            node = root # aby zakonczyc petle
            node_parent = None
        else: # w nie istnieje
            node = root # aby wyjsc z petli
    else: # Przypadki lustrzane
        w = node_parent.left
        if w and w.color == RED: # Przypadek 1
            w.color = BLACK
            node_parent.color = RED
```

```
                root, _ = bst_rotate_right(root, node_parent)
                w = node_parent.left
            if w and ((w.left is None and w.right is None) or \
               (w.left and w.left.color == BLACK and \
                w.right and w.right.color == BLACK)):
                # Przypadek 2
                w.color = RED
                node = node_parent
                if node == node_parent.left:
                    y_is_left = True
                else:
                    y_is_left = False
            elif w:
                if w.left.color is BLACK: # Przypadek 3
                    w.right.color = BLACK
                    w.color = RED
                    root, _ = bst_rotate_left(root, w)
                    w = node_parent.left
                # Przypadek 4
                w.color = node_parent.color
                node_parent.color = BLACK
                if w.left:
                    w.left.color = BLACK
                root, _ = bst_rotate_right(root, node_parent)
                node = root # aby wyjsc z petli
                node_parent = None
            else: # w nie istnieje
                node = root # aby wyjsc z petli
    if node: # wazne przy usuwaniu root
        node.color = BLACK
    return root
```

4.4. Usuwanie węzła

Procedura ta to niewielka modyfikacja algorytmu usuwania węzła z drzewa BST, po której następuje przywrócenie własności drzewa czerwono-czarnego po usunięciu węzła [2]. Funkcja zwraca korzeń i węzeł zastępujący usuwany węzeł.

Listing 4.4. Usuwanie elementu z drzewa RBT.

```
def rbt_delete(root, node):
    if node is None:
        return None
    y = node
    y_original_color = y.color
    y_is_left = False
    if y.parent:
        if y == y.parent.left:
            y_is_left = True
    if node.left is None:
        x = node.right
        temp = x
        if x:
            x.parent = y.parent
```

```
        x_parent = y.parent
        root, _ = bst_transplant(root, node, node.right)
    elif node.right is None:
        x = node.left
        temp = x
        if x:
            x.parent = y.parent
        x_parent = y.parent
        root, _ = bst_transplant(root, node, node.right)
    else:
        y = bst_find_min(node.right)
        temp = y
        y_original_color = y.color
        x = y.right
        if y.parent == node:
            if x:
                x.parent = y.parent
            x_parent = y.parent
        else:
            root, _ = bst_transplant(root, y, y.right)
            y.right = node.right
            if y.right:
                y.right.parent = y
        root, _ = bst_transplant(root, node, y)
        y.left = node.left
        y.color = node.color
        if y.left:
            y.left.parent = y
    if y is None or y_originial_color == BLACK:
        root = rbt_delete_fixup(root, x, x_parent, y_is_left)
    return root, temp
```

5. Drzewa AVL

Drzewo AVL (ang. *AVL tree*) to drzewo BST, w którym dla każdego węzła wysokość jego poddrzew dla lewego i prawego potomka różni się co najwyżej o 1. Węzeł drzewa AVL to węzeł drzewa BST z dodatkowym atrybutem balance, którym jest *współczynnik wyważenia* (ang. *balance factor*), czyli różnica wysokości lewego i prawego poddrzewa. W drzewie AVL wartość tego współczynnika może wynosić -1, 0 lub $+1$. Kiedy współczynnik przybiera wartość mniejszą niż -1 lub większą niż $+1$ konieczne jest równoważenie drzewa. Operacje wstawiania i usuwania węzła bazują na wersji tych procedur z drzewa BST. Wyszukiwanie w drzewie AVL nie różni się od wyszukiwania w drzewie BST [19].

5.1. Operacja rebalance

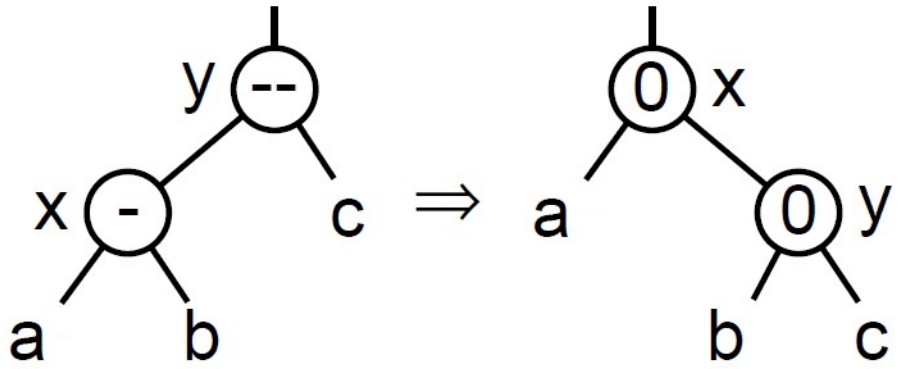
Operacja *rebalance* to procedura polegająca na wykonaniu sekwencji kroków, których celem jest przywrócenie własności drzewa AVL. Własności te mogą zostać zaburzone po wstawieniu węzła lub usunięciu go z prawego lub lewego poddrzewa. W każdym z tych przypadków, po osiągnięciu przez jakiś węzeł (y) współczynnika wyważenia równego -2 lub $+2$, możliwe są kolejne dwa przypadki: współczynnik wyważenia jednego z potomków (prawego lub lewego) węzła y może wynosić -1 lub $+1$ i wymagają one pojedynczej lub podwójnej rotacji.

W przypadku, gdy y ma współczynnik wyważenia równy -2, zaś jego lewy potomek ma współczynnik równy -1, konieczna jest prawa rotacja y.

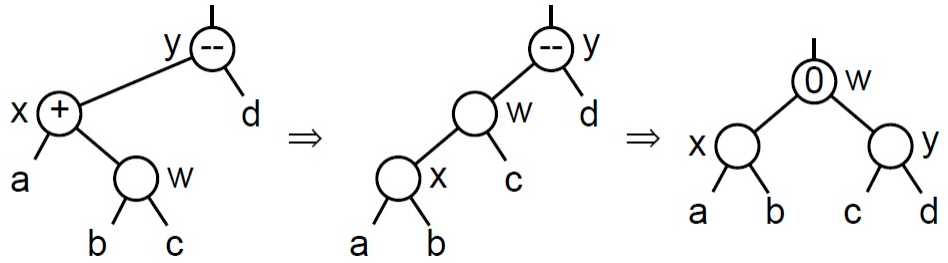
W przypadku, gdy y ma współczynnik wyważenia równy -2, zaś jego lewy potomek (x) ma współczynnik równy $+1$, konieczna jest podwójna rotacja: lewa rotacja węzła x, a następnie prawa rotacja węzła y.

Tabela 5.1. Złożoność obliczeniowa drzew AVL.

AVL	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(\log n)$



Rysunek 5.1. Przypadek 1 operacji *rebalance* w drzewie AVL: y.balance = −2, y. left .balance = −1 [20].



Rysunek 5.2. Przypadek 2 operacji *rebalance* w drzewie AVL: y.balance = −2, y. left .balance = +1 [20].

Listing 5.1 przedstawia realizację operacji *rebalance*, czli równoważenie drzewa AVL [20].

Listing 5.1. Operacja *rebalance* w drzewie AVL.

```
def avl_rebalance(node): # rownowazenie drzewa
    # metoda uzywana w metodach insert i delete
    w = None
    if y.balance == -2:
        # przy insert nowy wezel wstawiony jest do lewego poddrzewa,
        # przy delete wezel usuniety z prawego poddrzewa
        x = y.left
        w = x.right
        if x.balance == -1: # przypadek 1: balance -1
            w = x
            x.balance = 0
            y.balance = 0
            root, _ = bst_rotate_right(root, y)
        else: # przypadek 2: balance +1
            w = x.right
            root, _ = bst_rotate_left(root, x)
            root, _ = bst_rotate_right(root, y)
            if w.balance == -1:
                x.balance = 0
                y.balance = +1
            elif w.balance == 0:
```

```
                x.balance = 0
                y.balance = 0
            else: # w.balance == +1
                x.balance = -1
                y.balance = 0
            w.balance = 0
    elif y.balance == +2:
        # przy insert nowy wezel wstawiony jest do prawego poddrzewa,
        # przy delete wezel usuniety z lewego poddrzewa
        x = y.right
        w = y.left
        if x.balance == +1: # przypadek 1 lustrzany: balance +1
            w = x
            x.balance = 0
            y.balance = 0
            root, _ = bst_rotate_left(root, y)
        else:    # przypadek 2 lustrzany: balance -1
            w = x.left
            root, _ = bst_rotate_right(root, x)
            root, _ = bst_rotate_left(root, y)
            if w.balance == +1:
                x.balance = 0
                y.balance = -1
            elif w.balance == 0:
                x.balance = 0
                y.balance = 0
            else:    # w.balance == +1
                x.balance = +1
                y.balance = 0
            w.balance = 0
    return root
```

5.2. Operacja update

Operacja *update* służy do zaktualizowania współczynnika wyważenia po dodaniu węzła do drzewa AVL. Węzeł y to ostatnio odwiedzony węzeł o współczynniku wyważenia różnym od zera na drodze od korzenia do nowego węzła. Niektóre źródła podają, że aktualizacji przy wstawieniu należy dokonać od wstawionego węzła aż do korzenia, ale nie jest to konieczne. Listing 5.2 przedstawia operację aktualizowania współczynników wyważenia po wstawieniu węzła do drzewa AVL [20].

Listing 5.2. Operacja *update* w drzewie AVL.

```
def avl_update(node, y): # aktualizacja wspolczynnika balance
    while node != y:
        parent = node.parent
        if parent.left == node:
            parent.balance -= 1
        else:
            parent.balance += 1
        node = parent
```

5.3. Wstawianie węzła

Na listingu 5.3 przedstawiona jest iteracyjna wersja algorytmu wstawiania węzła do drzewa AVL. Najpierw ma miejsce iteracyjne wstawienie do drzewa BST, a po nim operacje *update* oraz *rebalance*. Funkcja zwraca korzeń oraz nowy węzeł [20].

Listing 5.3. Wstawianie węzła do drzewa AVL.

```
def avl_insert(top, new_node):
    # poczatkowo wstawiamy wezel jak do drzewa BST
    root, _ = bst_insert_iteratively(root, new_node)
    y = new_node
    while y != root: # poszukiwanie ostatnio odwiedzonego wezla,
                     # ktorego balance jest rozny 0
                     # potrzebny do metody update
                     # jest to poziom, do ktorego
                     # (przechodzac od nowo wstawionego wezla do gory)
                     # nalezy zaaktualizowac wspolczynnik balance
        y = y.parent
        if y.balance: # rozny od 0
            break
    # wywolanie metody update oraz rebalance
    avl_update(new_node, y)
    root = avl_rebalance(root, y)
    return root, new_node
```

5.4. Usuwanie węzła

Listing 5.4 przedstawia algorytm usuwania węzła z drzewa AVL. Na początku należy znaleźć następnik usuwanego węzła, następnie wezeł zostaje usunięty algorytmem z drzewa BST. Po tej operacji należy zbalansować drzewo, jeśli warunki zostały naruszone - w przeciwieństwie do operacji avl_insert, należy to zrobić na całej drodze od węzła, który zastąpił usuwany aż do korzenia, pamiętając o wcześniejszej aktualizacji współczynnika wyważenia każdego odwiedzonego węzła. Operacja *rebalance* po usunięciu węzła jest tożsama z operacją *rebalance* po wstawieniu nowego węzła. Funkcja zwraca korzeń oraz węzeł zastępujący usunięty węzeł [20].

Listing 5.4. Usuwanie węzła z drzewa AVL.

```
def avl_delete(root, node):
    if node is None:
        return node
    u = bst_find_successor(root, node)
    # poczatkowo usuwamy jak z drzewa BST
    root, _ = bst_delete(root, node)
    # nastepnie nalezy zbalansowac drzewo
    # jesli warunek drzewa AVL zostal naruszony
    succ = bst_find_successor(root, node) # potrzeba zaaktualizowac
    # balance nastepnika, gdyz przemiesci sie on na sciezke
    # od usunietego wezla do korzenia
    if succ:
        succ.balance = btree_height(succ.right) - \
```

```
            btree_height(succ.left)
    q = node.parent
    if u:
        u.balance = btree_height(u.right) - btree_height(u.left)
        if q is None:
            q = u
    while q:
        q.balance = btree_height(q.right) - btree_height(q.left)
        root = avl_rebalance(root, q)
        q = q.parent
    if u:
        return root, u
    return root, None
```

6. Drzewa splay

Drzewo splay (ang. *splay tree*), drzewo rozchylane, drzewo Sleatora-Tarjana to samoorganizujące się binarne drzewo poszukiwań [21]. Węzeł drzewa splay nie potrzebuje żadnych dodatkowych danych, jest to podstawowy węzeł z drzewa BST. Wykonywanie podstawowych operacji (wstawianie, wyszukiwanie, usuwanie) na tym drzewie wiąże się z wywołaniem procedury *splay*, która przemieszcza wybrany węzeł w górę do korzenia. Operacja *splay* powoduje, że częściej używane węzły są umieszczone bliżej korzenia [22].

6.1. Operacja splay

Operacja *splay* to procedura polegająca na wykonaniu sekwencji kroków, z których każdy przybliża element, na którym została ona wykonana, do korzenia. Wyróżniamy trzy przypadki kroków tej procedury, każdy z nich polega na wykonaniu pojedynczej lub podwójnej rotacji - kroki te nazywane są *zig*, *zig-zig*, oraz *zig-zag*.

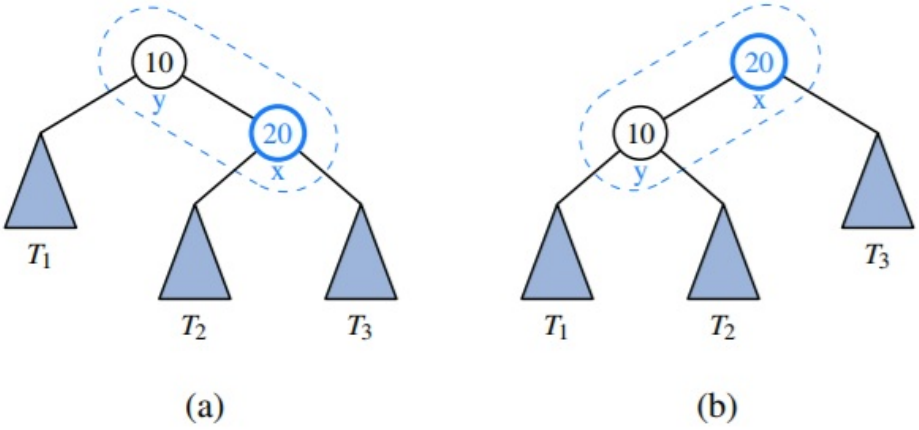
Operacja zig: Węzeł x ma rodzica, ale nie ma dziadka - w tym przypadku wykonujemy rotację rodzica - prawą jeśli x jest lewym dzieckiem y, lewą jeśli x jest prawym dzieckiem y.

Operacja zig-zig: Węzeł x i jego rodzic y są oboje lewymi lub oboje prawymi dziećmi swoich rodziców - w tym przypadku wykonujemy podwójną rotację: prawej dziadka i prawej rodzica jeśli wspomniane węzły są lewymi dziećmi, lewej dziadka i lewej rodzica w przeciwnym przypadku.

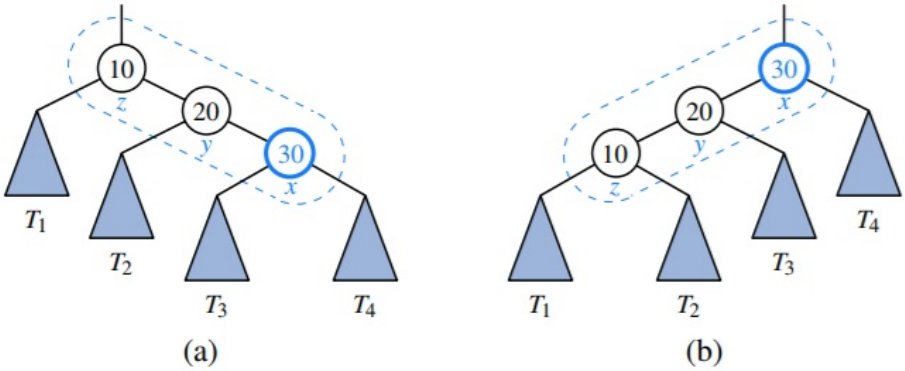
Operacja zig-zag: Jeden z węzłów x i y jest lewym dzieckiem, a drugi prawym - w tym przypadku wykonujemy podwójnej rotacji: lewej i prawej rodzica x, jeśli x jest prawym dzieckiem y, prawej i lewej rodzica x w przeciwnym przypadku.

Tabela 6.1. Złożoność obliczeniowa drzew splay.

Splay	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$



Rysunek 6.1. Operacja *zig* w drzewie splay: (a) przed; (b) po. Symetryczna sytuacja, gdy x jest początkowo lewym dzieckiem y [22].

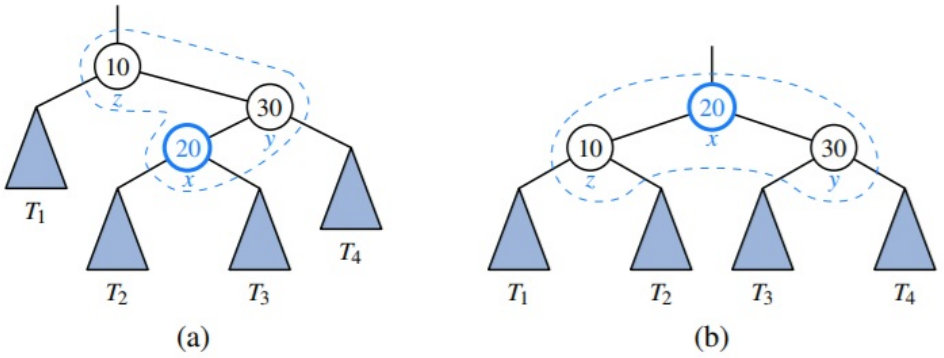


Rysunek 6.2. Operacja *zig-zig* w drzewie splay: (a) przed; (b) po. Symetryczna sytuacja, gdy x i y są lewymi dziećmi [22].

Listing 6.1 przedstawia operację *splay* w drzewie splay [22].

Listing 6.1. Operacja *splay* w drzewie splay.

```
def splay_splay(root, node):
    while True: #przesuwamy wezel do korzenia
        if node.parent is None: # node jest juz rootem
            break
        # ZIG - node nie ma dziadka
        if node.parent.parent is None: # ojcem node jest korzen - ZIG
            if node.parent.left == node:
                root, _ = bst_rotate_right(root, node.parent)
            else:
                root, _ = bst_rotate_left(root, node.parent)
            break # po powyzszych rotacjach node juz jest korzeniem
        # ZIG-ZIG node i jego rodzic sa oboje lewymi dziecmi lub oboje
        # prawymi dziecmi swoich rodzicow
        if node.parent.parent.left == node.parent and \
            node.parent.left == node: # prawy ZIG-ZIG
                root, _ = bst_rotate_right(root, node.parent.parent)
```



Rysunek 6.3. Operacja *zig-zag* w drzewie splay: (a) przed; (b) po. Symetryczna sytuacja, gdy x jest prawym dzieckiem i y lewym dzieckiem [22].

```
                root, _ = bst_rotate_right(root, node.parent)
                continue
        if node.parent.parent.right == node.parent and \
            node.parent.right == node: # lewy ZIG-ZIG
                root, _ = bst_rotate_left(root, node.parent.parent)
                root, _ = bst_rotate_left(root, node.parent)
                continue
        # ZIG-ZAG - jeden sposrod node i jego rodzica jest lewym dzieckiem
        # a drugi prawym
        if node.parent.right == node: # lewy ZIG, prawy ZAG
            root, _ = bst_rotate_left(root, node.parent)
            root, _ = bst_rotate_right(root, node.parent)
        else: # prawy ZIG, lewy ZAG
            root, _ = bst_rotate_right(root, node.parent)
            root, _ = bst_rotate_left(root, node.parent)
    return root
```

6.2. Wyszukiwanie dowolnego klucza w drzewie

Listing 6.2 przedstawia iteracyjną wersję algorytmu wyszukiwania dowolnego klucza w drzewie splay. Składa się on z wyszukiwania iteracyjnego w drzewie BST, wzbogaconego o łącze do rodzica rozważanego w danym momencie węzła. Jeśli węzeł zostanie znaleziony w drzewie, to następnie jest na nim wykonywana operacja *splay*, w przeciwnym przypadku poddajemy operacji *splay* liść, na którym zakończyło się nieudane wyszukiwanie. Funkcja zwraca korzeń drzewa [22].

Listing 6.2. Wyszukiwanie dowolnego klucza w drzewie splay.

```
def splay_search(root, data):
    if root:
        parent = None
        node = root
        while node and node.data != data:
            parent = node
            if data < node.data:
```

```
                node = node.left
            else: # data > node.data
                node = node.right
        if node: # wezel o zadanym kluczu jest w drzewie
            root = splay_splay(root, node)
        else: # wezla o zadanym kluczu nie ma w drzewie
            root = splay_splay(root, parent)
    return root # jesli drzewo jest puste
```

6.3. Wstawianie węzła

Listing 6.3 przedstawia iteracyjną wersję algorytmu wstawiania węzła do drzewa splay. Składa się on z iteracyjnego wstawienia do drzewa BST, po którym następuje operacja *splay* na wstawionym węźle. Funkcja zwraca korzeń drzewa i wstawiony węzeł [22].

Listing 6.3. Wstawianie węzła do drzewa splay.

```
def splay_insert(root, new_node):
    # poczatkowo wstawiamy wezel jak do drzewa BST
    root, _ = bst_insert(root, new_node)
    # nastepnie wykonujemy metode splay
    root = splay_splay(root, new_node)
    return root, new_node
```

6.4. Usuwanie węzła

Listing 6.4 przedstawia algorytm usuwania węzła z drzewa splay. Jeśli usuwany węzeł znajduje się w drzewie, to najpierw wykonywana jest na nim operacja splay 6.1, a następnie węzeł ten zostaje usunięty algorytmem z drzewa BST i zwrócony przez metodę. W przeciwnym wypadku zostaje zwrócona wartość None [21].

Listing 6.4. Usuwanie węzła z drzewa splay.

```
def splay_delete(root, node):
    if node:
        root = splay_splay(root, node)
        root, _ = bst_delete(root, node)
        return root, root
    return root, node
```

7. Podsumowanie

W ramach pracy przygotowano implementacje w języku Python dla czterech rodzajów drzew binarnych. Podstawowym rodzajem drzew są drzewa BST, na których można wykonywać operacje wstawiania, usuwania, wyszukiwania elementów, oraz wyszukiwania poprzednika, następnika, elementu najmniejszego i największego. Jeżeli drzewa BST są zrównoważone, to podane operacje są realizowane wydajnie. W pesymistycznych przypadkach tak nie jest, dlatego następne trzy rodzaje drzew wprowadzają różne modyfikacje poprawiające sytuację. Mamy drzewa czerwono-czarne, drzewa AVL, oraz drzewa splay.

W pracy przygotowano dwa rodzaje implementacji dla każdego rodzaju drzew. Pierwsza implementacja to cztery klasy, po jednej dla każdego rodzaju drzewa binarnego. Klasa BinarySearchTree jest klasą bazową dla pozostałych trzech klas: RedBlackTree, AVLTree, SplayTree. W kodzie obiektowym łatwo można prześledzić jakie zmiany wprowadza nowy rodzaj drzewa, a które operacje są dziedziczone bez zmian.

Druga implementacja to zestaw funkcji, które w zamierzeniu mają zastępować typowy pseudokod z podręczników informatyki. Nazwy funkcji mają odpowiedni przedrostek, stanowiący rodzaj przestrzeni nazw dla każdego rodzaju drzewa. Tutaj również można łatwo prześledzić pochodzenie danej operacji na drzewie binarnym.

Oba rodzaje implementacji zostały pokryte testami jednostkowymi z użyciem standardowego modułu unittest.

Przygotowane implementacje mogą zostać wzbogacone tak, aby pomóc w rozwiązywaniu innych problemów. Przykładowo drzewa czerwono-czarne mogą stać się bazą dla struktury danych umożliwiającej wyznaczanie statystyk pozycyjnych na zbiorze dynamicznym. Ta struktura danych nosi nazwę *drzewa statystyk pozycyjnych* [2]. Drzewa czerwono-czarne są także używane do stworzenia *drzew przedziałowych*, które implementują operacje na dynamicznych zbiorach przedziałów na osi liczbowej [2]. Możemy przykładowo szybko znajdować przedziały mające część wspólną z danym przedziałem.

A. Klasy dla węzłów drzew

Moduł treenodes zawiera klasy reprezentujące węzły dla różnych rodzajów drzew binarnych opisywanych w pracy. W przypadku drzew czerwono-czarnych skorzystano z modułu termcolor do kolorowania węzłów przy ich wyświetlaniu na konsoli. Zawartość stałych RED, BLACK dobrano tak, aby dobrze współpracowały z modułem termcolor.

Listing A.1. Moduł treenodes.

```
#!/usr/bin/python

from termcolor import colored

class Node:

    def __init__(self, data=None, left=None, right=None, parent=None):
        self.data = data
        self.left = left
        self.right = right
        self.parent = parent

    def __str__(self):
        return str(self.data)

RED, BLACK = 'red', None

class RBTNode(Node):

    def __init__(self, data=None, left=None, right=None, parent=None, color=RED):
        Node.__init__(self, data, left, right, parent)
        self.color = color

    def __str__(self):
        return colored(Node.__str__(self), self.color)

class AVLNode(Node):

    def __init__(self, data=None, left=None, right=None, parent=None, balance=0):
        Node.__init__(self, data, left, right, parent)
        self.balance = balance    #-1,0,+1

    def __str__(self):
        return "{}({})".format(self.data, self.balance)
```

B. Testy algorytmów

Na wykresie B.1 przedstawiona jest zależność czasu od ilości węzłów podczas tworzenia drzew, gdy dane posortowane są rosnąco. Algorytm DSW zwiększa czas tworzenia drzewa BST, ale zyskujemy na szybkości innych operacji, które można wykonać na drzewie. Najszybciej budowane jest drzewo splay, zaś drzewa RBT i AVL powstają w czasie około dwa razy dłuższym.

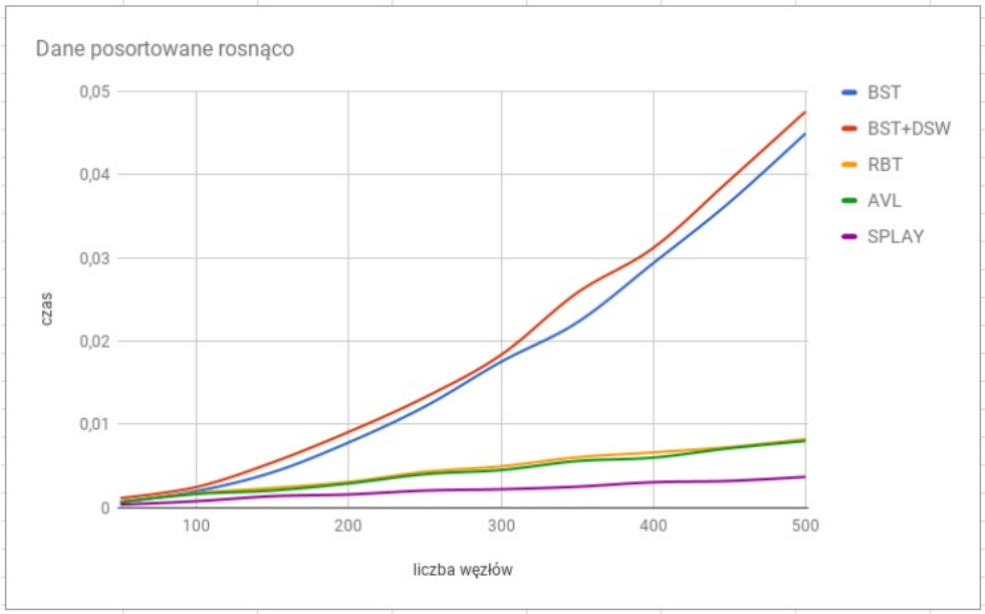
Na wykresie B.2 przedstawiona jest wysokość drzew zbudowanych z danych posortowanych rosnąco. W tym zestawieniu drzewa BST i splay wypadają fatalnie, są mocno niezrównoważone, czego należało oczekiwać. Mimo niewielkiej różnicy w wysokości tworzonych drzew, drzewa RBT wypadają odrobinę słabiej na tle idealnych drzew BST i AVL.

Na wykresie B.3 przedstawiona jest zależność czasu od liczby węzłów podczas tworzenia drzew z danych w kolejności losowej. Widać ponownie, że wywołanie algorytmu DSW zwiększa czas tworzenia drzewa BST. Najdłużej są budowane drzewa splay, zaś czasy dla drzew RBT i AVL ponownie są porównywalne.

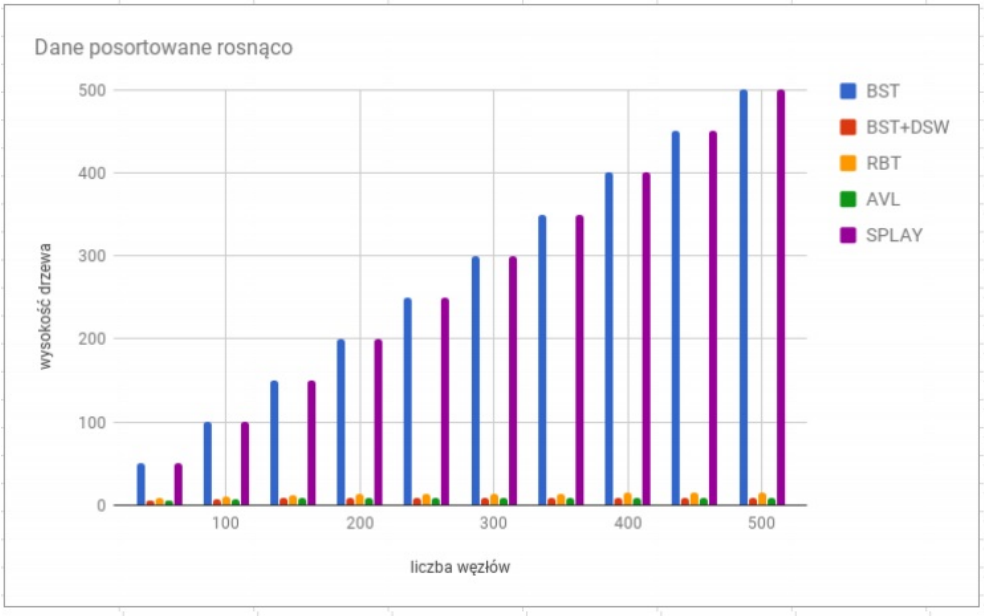
Na wykresie B.4 przedstawiona jest wysokość drzew zbudowanych z danych ułożonych w kolejności losowej. W tym zestawieniu drzewa BST i drzewa splay ponownie wypadają najgorzej. Tym razem idealne drzewo BST w każdym z przypadków ma najmniejszą wysokość. Drzewa RBT i AVL są mniej więcej porównywalne pod tym względem, przy czym lekką przewagę mają drzewa AVL.

W tabeli B.1 zaprezentowano czasy trzykrotnego wyszukania tego samego wylosowanego klucza (27). W drzewie jest 1000 węzłów. Dane przed wstawieniem do drzewa były uporządkowane rosnąco. Iteracyjne czasy dla BST + DSW oraz RBT są podobne (korzystają z tej samej funkcji), z czego można wnioskować, że są najlepiej zbalansowane dla takich danych. Wyraźnie widać, że pierwsze wyszukanie w drzewie splay trwa znacznie dłużej niż drugie i trzecie, kiedy to szukany klucz znajduje się w korzeniu.

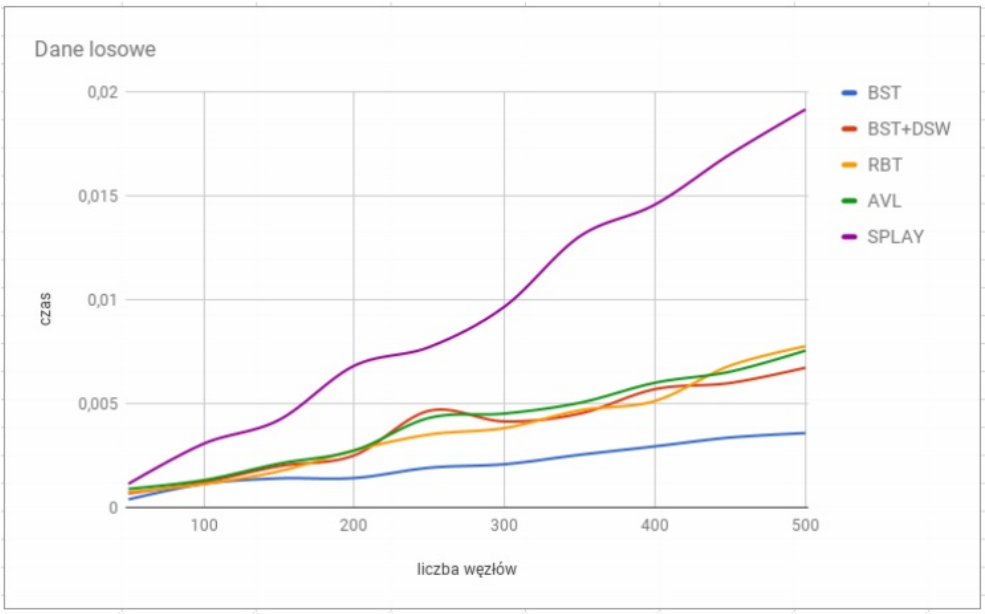
W tabeli B.2 zaprezentowano czasy trzykrotnego wyszukania tego samego wylosowanego klucza (579). W drzewie jest 1000 węzłów. Dane przed wstawieniem do drzewa były ułożone losowo. Z porównania danych wynika, że dla danych ułożonych losowo drzewo BST stało się całkiem dobrym drzewem. Dla takich danych drzewa RBT i AVL w wyszukiwaniu iteracyjnym dają podobne wyniki. Znowu zauważalna jest różnica między czasem pierwszego wyszukania klucza, a drugim i trzecim wyszukaniem w drzewie splay.



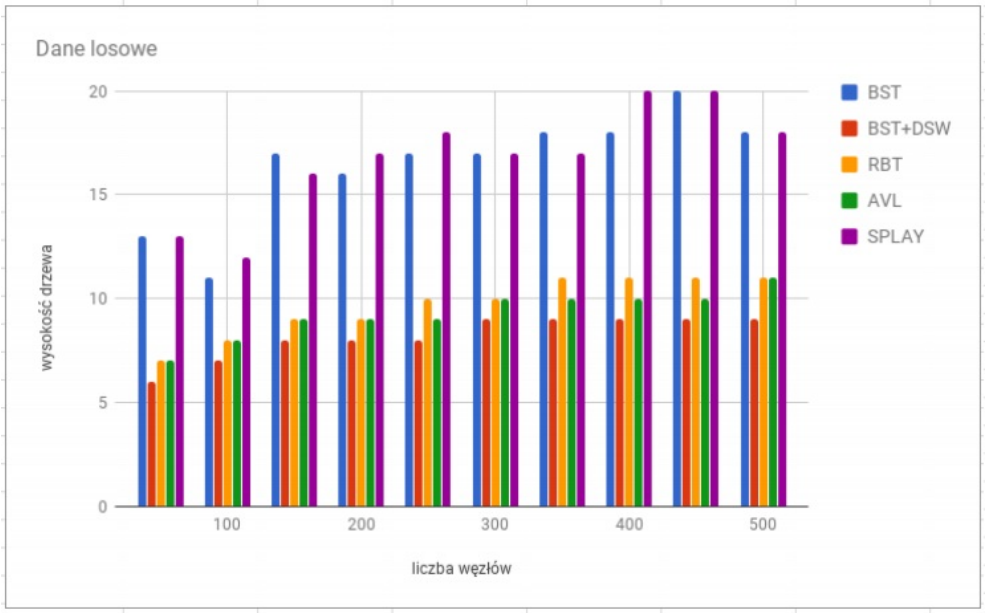
Rysunek B.1. Zależność czasu od liczby węzłów podczas tworzenia drzew, gdy dane są posortowane rosnąco.



Rysunek B.2. Wysokość drzew zbudowanych z danych posortowanych rosnąco.



Rysunek B.3. Zależność czasu od liczby węzłów podczas tworzenia drzew, gdy dane są w kolejności losowej.



Rysunek B.4. Wysokość drzew zbudowanych z danych w kolejności losowej.

Tabela B.1. Trzykrotne wyszukanie tego samego wylosowanego klucza (912) - dane przed wstawieniem do drzewa były uporządkowane. W drzewie jest 1000 węzłów. Dane w tabeli są podane w mikrosekundach.

BST (rekurencja)	793	625	609
BST (iteracja)	2057	2293	2211
BST + DSW (rekurencja)	8	154	11
BST + DSW (iteracja)	27	27	36
RBT (rekurencja)	7	27	7
RBT (iteracja)	113	42	81
AVL (rekurencja)	35	47	21
AVL (iteracja)	105	19	103
splay	1102	12	15

Tabela B.2. Trzykrotne wyszukanie tego samego wylosowanego klucza (579) - dane przed wstawieniem do drzewa zostały pomieszane. W drzewie jest 1000 węzłów. Dane w tabeli są podane w mikrosekundach.

BST (rekurencja)	28	13	22
BST (iteracja)	62	66	57
BST + DSW (rekurencja)	10	13	26
BST + DSW (iteracja)	50	33	34
RBT (rekurencja)	25	58	25
RBT (iteracja)	46	50	61
AVL (rekurencja)	70	38	70
AVL (iteracja)	47	71	83
splay	257	18	18

Bibliografia

[1] Python Programming Language - Official Website, `https://www.python.org/`.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Wprowadzenie do algorytmow*, Wydawnictwo Naukowe PWN, Warszawa 2012.

[3] Robert Sedgewick, *Algorytmy w C++*, Wydawnictwo RM, Warszawa 1999.

[4] Gavin Powell, *Beginning Database Design*, Wiley, Indianapolis 2006.

[5] Elvis C. Foster, Shripad Godbole, *Database Systems: A Pragmatic Approach*, Apress, United States 2016.

[6] William McAllister, *Data Structures and Algorithms Using Java*, Jones and Bartlett Publishers, Sudbury 2009.

[7] Dick Grune,Ceriel J.H. Jacobs, *Parsing Techniques: A Practical Guide*, Springer, 2008.

[8] Jeff Edmonds, *How to Think About Algorithms*, Cambridge University Press, 2008.

[9] James L. Hein, *Discrete Mathematics*, Jones & Barlett Learning, 2003.

[10] K R Venugopal, K G Srinivasa, P M Krishnaraj, *File Structures Using C++*, McGraw-Hill, New Delhi 2009.

[11] Ray Lischner, *Exploring C++ 11*, Apress, Berkeley 2013.

[12] M. de Berg, O. Cheong, M. van Krefeld, M. Overmars, *Computational Geometry: Algorithms and Applications*, Springer, 3rd edition.

[13] D. Brackeen, B. Barker, L. Vanhelsuwe, *Developing Games in Java*, New Riders, 2004.

[14] Wikipedia, Tree traversal, 2017, `https://en.wikipedia.org/wiki/Tree_traversal`.

[15] Wikipedia, Binarne drzewo poszukiwań, 2017, `https://pl.wikipedia.org/wiki/Binarne_drzewo_poszukiwan`.

[16] Jerzy Wałaszek, *Algorytmy i struktury danych*, *Równoważenie drzewa BST – algorytm DSW*, 2017, `http://eduinf.waw.pl/inf/alg/001_search/0116.php`.

[17] Wikipedia, Algorytm DSW, 2017, `https://pl.wikipedia.org/wiki/Algorytm_DSW`.

[18] Wikipedia, Drzewo czerwono-czarne, 2017, `https://pl.wikipedia.org/wiki/Drzewo_czerwono-czarne`.

[19] Wikipedia, Drzewo AVL, 2017, `https://pl.wikipedia.org/wiki/Drzewo_AVL`.

[20] *An Introduction to Binary Search Trees and Balanced Trees. Libavl Binary Search Tree Library. Volume 1: Source Code. Version 2.0.2 by Ben Pfaff.*

[21] Wikipedia, Drzewo splay, 2017, `https://pl.wikipedia.org/wiki/Drzewo_splay`.

[22] Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, *Data Structures and Algorithms in Java*, Wiley, Sixth Edition, 2014.