



Politechnika Poznańska

Informatyka rok I semestr 2

L10, Piątek 11:45 - 13:15

Algorytmy i Struktury Danych

Prowadzący: Dominik Piotr Witczak

Sprawozdanie nr 2

Drzewa przeszukiwań binarnych
BST i drzewa samobalansujące AVL

Autor:

Dominik Fischer 164176

Oliwer Miller 163544

Rok akademicki 2024/2025

Wprowadzenie

Celem niniejszego sprawozdania jest analiza i porównanie dwóch struktur danych: binarnego drzewa poszukiwań BST oraz samobalansującego się drzewa AVL. Obie struktury służą do przechowywania danych w sposób umożliwiający szybkie wyszukiwanie, wstawianie i usuwanie elementów. W praktyce jednak ich wydajność może się znacznie różnić.

W ramach projektu zaimplementowano oba typy drzew, a następnie dokonano pomiarów czasu wykonywania podstawowych operacji dla zestawów danych o różnej wielkości i strukturze.

Tworzenie drzewa BST

Klasa *BSTNode* reprezentuje pojedynczy węzeł drzewa BST. Składa się z klucza *key* i odnośników do lewego i prawego potomka.

Klasa *BST* reprezentuje drzewo BST. Korzeń drzewa jest zapisany w *root*. Funkcja *insert* służy do dodawania nowych węzłów, wykorzystuje do tego *_insert*, które rekurencyjnie wstawia węzeł po lewej lub prawej stronie, zależnie od jego wartości.

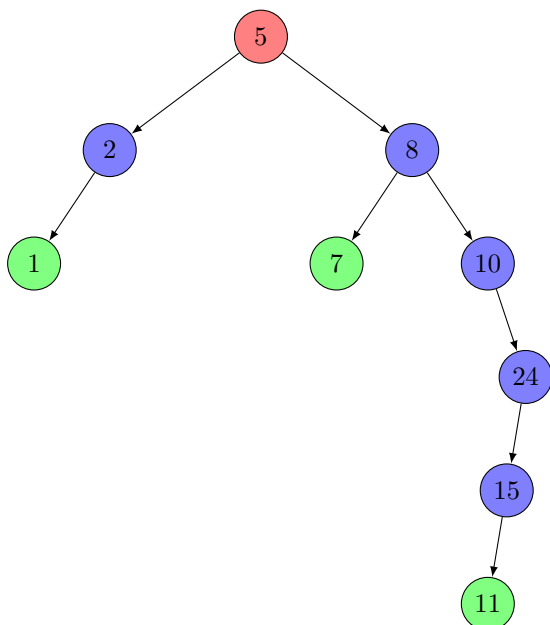
Poniżej drzewo BST utworzone z liczb:

5 8 2 5 10 24 15 11 7 1 2

Kolorem czerwonym został zaznaczony korzeń

Kolorem niebieskim węzły wewnętrzne

Kolorem zielonym liście



Terminal

```
class BSTNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

class BST:
    def __init__(self):
        self.root = None

    def insert(self, key):
        self.root = self._insert(self.root,
                                   key)

    def _insert(self, node, key):
        if not node:
            return BSTNode(key)
        if key < node.key:
            node.left = self._insert(node.left,
                                      key)
        elif key > node.key:
            node.right = self._insert(node.
                                       right, key)
        return node
```

Tworzenie drzewa AVL

Klasa *AVLNode* ma dodatkowo przypisaną wysokość węzła *height*, wstępnie równa 1, dla przypadku liścia. Przydaje się ona do utrzymania zrównoważenia.

Funkcja *build_from_sorted()* buduje drzewo AVL z posortowanej listy, zgodnie z metodą połowienia binarnego.

Funkcje *get_height()* i *update_height()* mają za zadanie obliczanie i aktualizowanie wysokości węzła, a funkcja *get_balance()* zwraca współczynnik zrównoważenia.

Następne są dwie funkcje *rotate_right* i *rotate_left* odpowiadające za rotacje w prawo i w lewo, używające wcześniej wspomnianej funkcji *update_height()*. Funkcja *insert()* jest odpowiedzialna za wstawianie nowego węzła do drzewa. Wykorzystuje do tego *__insert()* działające rekurencyjnie. Po wstawieniu węzła, aktualizowana jest wysokość i sprawdzany jest balans, w razie potrzeby później mogą zostać wykonane rotacje.

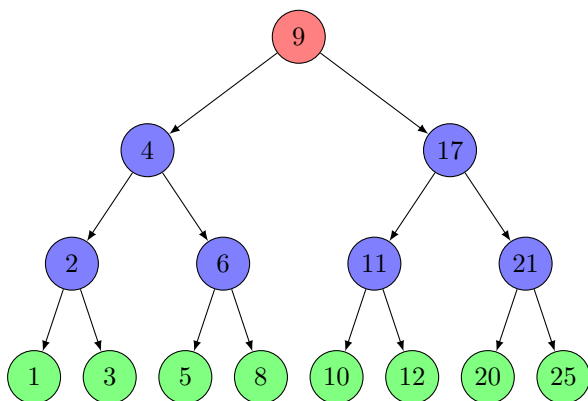
Poniżej drzewo AVL utworzone z liczb:

6 2 4 5 3 10 9 8 17 21 20 25 12 11 1 2

Kolorem czerwonym został zaznaczony korzeń

Kolorem niebieskim węzły wewnętrzne

Kolorem zielonym liście

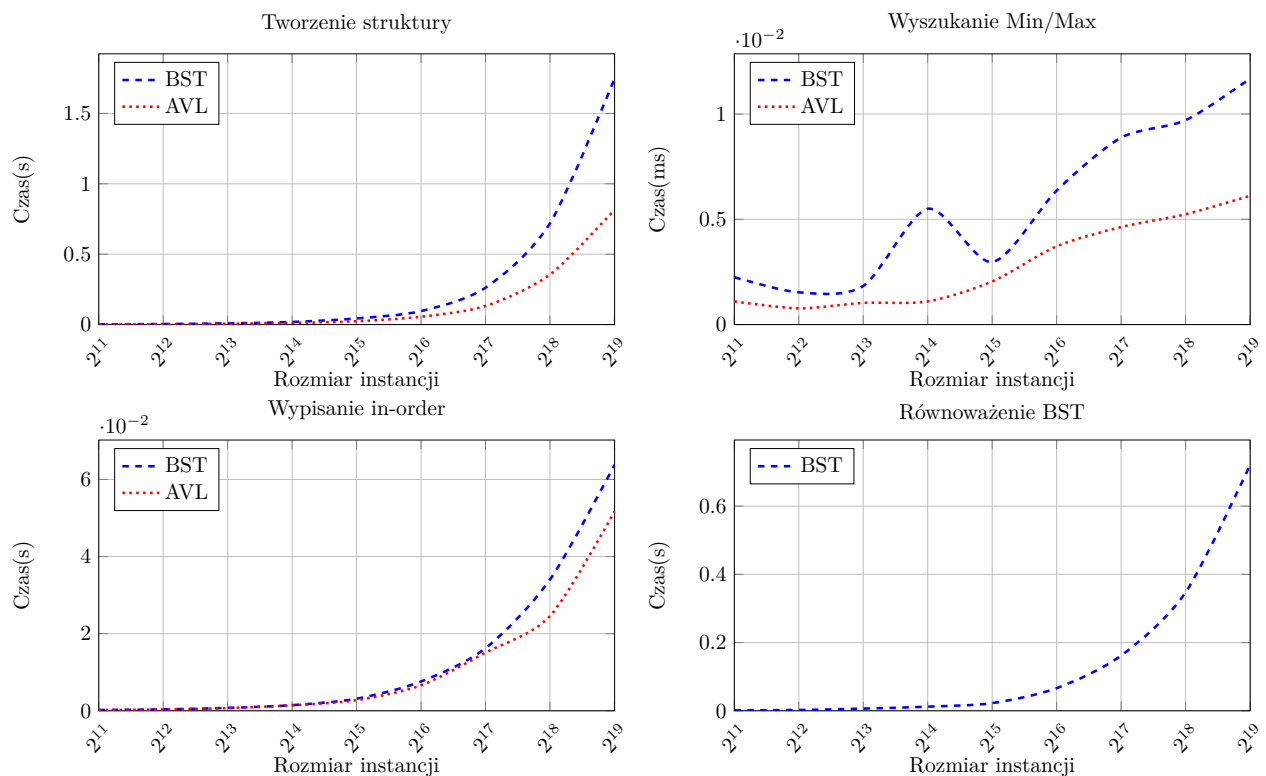


Terminal

```
class AVLNode:
    def __init__(self, key):
        self.key = key
        self.height = 1
        self.left = None
        self.right = None

class AVL:
    def __init__(self):
        self.root = None
    def build_from_sorted(self, values):
        def build(l, r):
            if l > r:
                return None
            mid = (l + r) // 2
            node = AVLNode(values[mid])
            node.left = build(l, mid - 1)
            node.right = build(mid + 1, r)
            node.height = 1 + max(self.get_height(node.left), self.get_height(node.right))
            return node
        self.root = build(0, len(values) - 1)
    def get_height(self, node):
        return node.height if node else 0
    def get_balance(self, node):
        return self.get_height(node.left) - self.get_height(node.right) if node else 0
    def update_height(self, node):
        node.height = 1 + max(self.get_height(node.left), self.get_height(node.right))
    def rotate_right(self, y):
        x = y.left
        T2 = x.right
        x.right = y
        y.left = T2
        self.update_height(y)
        self.update_height(x)
        return x
    def rotate_left(self, x):
        y = x.right
        T2 = y.left
        y.left = x
        x.right = T2
        self.update_height(x)
        self.update_height(y)
        return y
    def insert(self, key):
        self.root = self._insert(self.root, key)
    def _insert(self, node, key):
        if not node:
            return AVLNode(key)
        if key < node.key:
            node.left = self._insert(node.left, key)
        elif key > node.key:
            node.right = self._insert(node.right, key)
        else:
            return node
        self.update_height(node)
        balance = self.get_balance(node)
        if balance > 1 and key < node.left.key:
            return self.rotate_right(node)
        if balance < -1 and key > node.right.key:
            return self.rotate_left(node)
        if balance > 1 and key > node.left.key:
            node.left = self.rotate_left(node.left)
            return self.rotate_right(node)
        if balance < -1 and key < node.right.key:
            node.right = self.rotate_right(node.right)
            return self.rotate_left(node)
        return node
```

Porównanie czasów wykonania operacji



Rysunek 1: Wykresy tworzenia, wyszukiwania min/max, wypisania in-order, równoważenia

Wnioski

Drzewo samobalansujące AVL wypada lepiej, jeśli chodzi o czas wykonania powyższych operacji. Dzieje się to dzięki automatycznemu równoważeniu zastosowanym w drzewie AVL. Drzewo BST przy losowych danych może mieć dużą wysokość, co pogarsza wydajność operacji, inną niekorzystną cechą drzewa BST jest to, że musi być dodatkowo równoważone.