Laboratoria, piątek 9:45
Grupa L9
Informatyka, WIiT
Jędrzej Ogrodowski 160229
Maksymilian Norkiewicz 160267

Algorytmy i Struktury Danych
Prowadzący mgr inż. Dominik Witczak

Sprawozdanie projekt 2
Drzewa przeszukiwań binarnych BST i drzewa samobalansujące

1 Wstęp

W dzisiejszym świecie informatyki, efektywne zarządzanie danymi stanowi kluczowy element w projektowaniu oprogramowania. Jednym z najważniejszych aspektów tego zarządzania jest wybór odpowiedniej struktury danych, która umożliwi szybkie wyszukiwanie, dodawanie i usuwanie danych. W tym kontekście, drzewa BST (Binary Search Tree) oraz drzewa AVL odgrywają istotną rolę.

Celem projektu była implementacja drzew BST i AVL oraz badanie efektywnosci operacji wykonywanych na tych drzewach. Implementacja została napisana w językach C++, Python oraz Bash. Testy benchmark zostały wykonane na komputerze Apple Macbook Air z procesorem M2.

Projekt dostępny jest w całości na platofrmie Github.

2 Tworzenie drzewa BST

Drzewo BST tworzone jest z ciągu liczb.

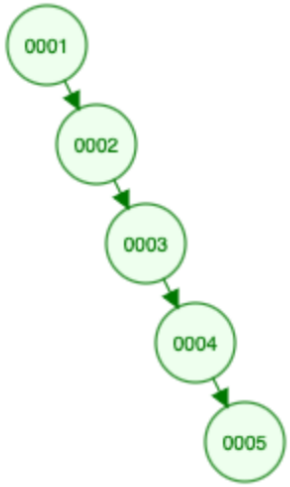
```
NodeBST *insertBST(NodeBST *root, int key)
{
    if (root == nullptr)
    {
        return createNodeBST(key);
    }

    if (key < root->key)
    {
```

```
        root->left = insertBST(root->left, key);
    }

    else if (key >= root->key)
    {
        root->right = insertBST(root->right, key);
    }

    return root;
}
```



Rysunek 1: Drzewo BST (zdegenerowane)

3 Tworzenie drzewa AVL

Drzewo AVL w tej implementacji tworzone jest poprzez zastosowanie algorytmu przeszukiwania połówkowego.

```
NodeAVL *createAVL(std::vector<int> &arr, int start, int end)
{
    if (start > end)
        return nullptr;

    int mid = (start + end) / 2;
    NodeAVL *root = createNodeAVL(arr[mid]);
```

```
    root->left = createAVL(arr, start, mid - 1);
    root->right = createAVL(arr, mid + 1, end);

    return root;
}
```

Kluczowym w tej sytuacji jest utrzymanie optymalnej wysokości drzewa, dlatego dodane zostały funkcje, które w tym celu wykonują rotacje.

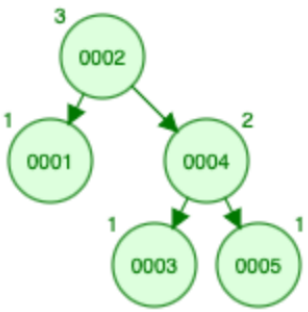
```
int balanceCoef = balanceCoefficient(root);

// LL case
if (balanceCoef > 1 && key < root->left->key)
{
    return rightRotation(root);
}

// RR case
if (balanceCoef < -1 && key > root->right->key)
{
    return leftRotation(root);
}

// LR case
if (balanceCoef > 1 && key > root->left->key)
{
    root->left = leftRotation(root->left);
    return rightRotation(root);
}

// RL case
if (balanceCoef < -1 && key < root->right->key)
{
    root->right = rightRotation(root->right);
    return leftRotation(root);
}
```



Rysunek 2: Drzewo AVL

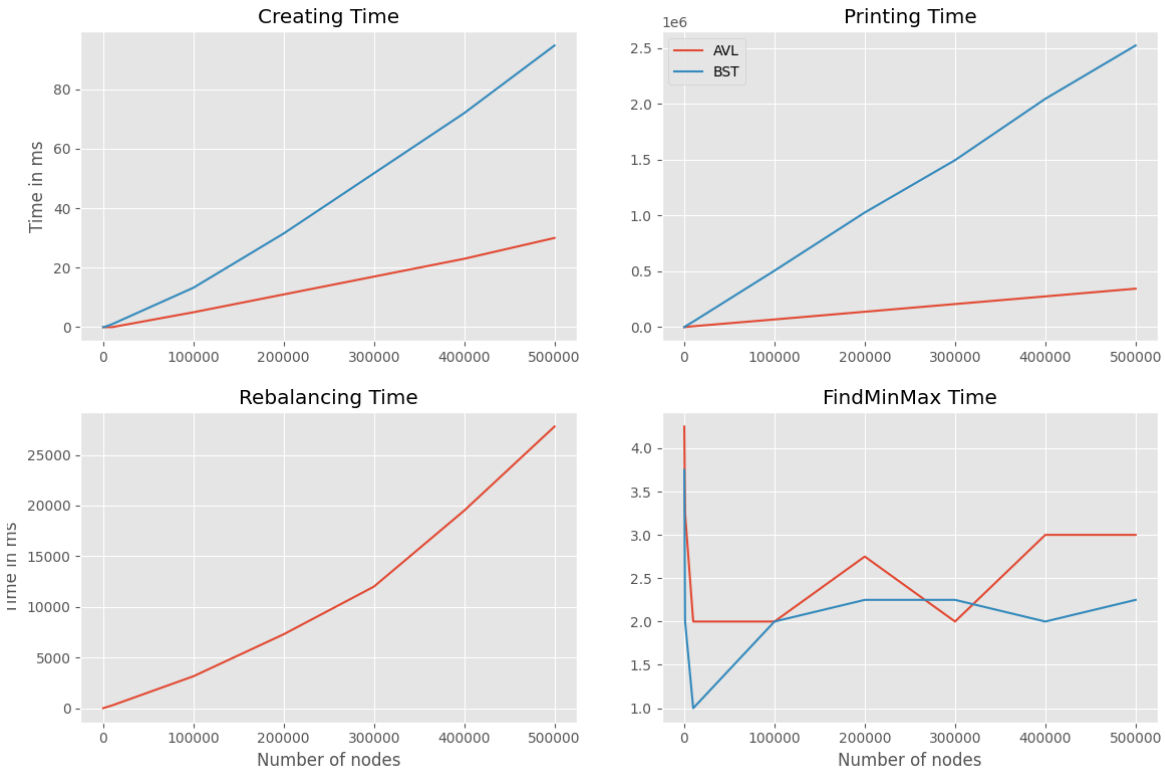
4 Badanie efektywności operacji

W projekcie zostały zaimplementowanie operacje:

- Tworzenie drzewa
- Wyszukiwane elementu w drzewie
- Wypisanie elementów drzewa
- Usuwanie z drzewa pojedyńczych elementów lub całego drzewa
- Równoważenie drzewa

Pomiary czasu zostały zrobione za pomocą biblioteki std::chrono. Każdy zmierzony czas zapisywany jest do pliku. Pliki wejściowe zostały wygenerowane w instancjach 100, 1000, 10000, 100000, 200000, 300000, 400000 500000. Uruchomienie benchmarku jest wykonane jako skrypt Bash.

Pomiary widoczne na wykresach sa uśrednione wartości 4 pomiarów.



Rysunek 3: Pomiary poszczególnych operacji

Pomiary zgadzają się ze złożonościami teorytycznie omawianymi na wykładzie. Z powodu implementacji mogą jednak wystąpić błędy pomiarowe.

5 Podsumowanie

Przeprowadzenie projektu związanego z implementacją oraz analizą drzew BST i AVL dostarczyło cennych wniosków dotyczących efektywności obu tych struktur danych.
Drzewa BST, dzięki swojej budowie, mogą być bardzo efektywnymi strukturami danych. Należy pamiętać, że w przypadku drzew BST, dobranie nieodpowiednych danych prowadzi do przypadku pesymistycznego. Problem ten rozwiązuje zastosowanie drzewa AVL.
Analiza wyników pomiarów pozwoliła na lepsze zrozumienie charakterystyki obu struktur danych oraz ich odpowiednich zastosowań.