Report: "An Introduction to Applied Dendrology"

Introduction

Dendrology, the study of trees, is applied in various fields, and understanding the efficiency of different tree-based data structures is crucial in computer science. In this report, we explore the implementation of a binary tree-based container for integers and empirically compare the cumulative running times for inserting keys. The comparison includes:

a) Insertion into a binary tree in the original random order.
b) Insertion into a binary tree in the best-case order, achieving a perfectly balanced tree.
c) Insertion into the
`std::set` library solution.

Task 1: Binary Tree Implementation and Analysis

1.1 Binary Tree Implementation

▼ A binary tree structure was implemented to store integers. The insertion process involves adding keys in random order, creating an unbalanced tree, and then rearranging keys to achieve a perfectly balanced tree.

```
struct TreeNode {
    int key;
    int height;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int k) : key(k), height(1), left(nullptr), right
};

class BinaryTree {
private:
```

```
    TreeNode* root;

    TreeNode* insert(TreeNode* node, int key) {
        if (node == nullptr) {
            return new TreeNode(key);
        }

        if (key < node->key) {
            node->left = insert(node->left, key);
        } else if (key > node->key) {
            node->right = insert(node->right, key);
        } else {
            // Ignore duplicate keys
            return node;
        }

        return node;
    }

    void inOrderTraversal(TreeNode* node) {
        if (node != nullptr) {
            inOrderTraversal(node->left);
            std::cout << node->key << " ";
            inOrderTraversal(node->right);
        }
    }

public:
    BinaryTree() : root(nullptr) {}

    void insertKey(int key) {
        root = insert(root, key);
    }

    void traverseInOrder() {
        inOrderTraversal(root);
```

```
    }
};
```

1.2 Random and Best-case Scenarios

▼ For random scenarios, a set of random keys is generated and inserted into the binary tree. In best-case scenarios, keys are rearranged before insertion to obtain a tree structure with consecutive levels built downwards.

```
    std::random_device rd;
  std::mt19937 gen(rd());


  std::cout << "size \t binary_tree \t avl_tree \tbestcase_bi



  for (int m = 10; m < 16; m++) {
      int dataSize = static_cast<int>(std::pow(2, m)) - 1;
      std::uniform_int_distribution<> dist(1, dataSize);
      long binaryTreeTime = 0;
      long avlTreeTime = 0;
      long stdSetTime = 0;
      long AVLBestCaseTimes = 0;

      BinaryTree binaryTree;
      AVLTree avlTree;
      AVLTree AVLBestCaseTree;
      std::set<int> stdSetContainer;

      std::vector<int> randomKeys(dataSize);
      for (int i = 0; i < dataSize; i++) {
          randomKeys[i] = dist(gen);
      }
```

```
    std::vector<int>bestCaseKeys(dataSize);
    int value = 1;

    getSortedBalancedVector(bestCaseKeys, 0, dataSize - 1,

    binaryTreeTime += runKeyInsertionBinaryTree(binaryTree,
    avlTreeTime += runKeyInsertionAVLTree(avlTree, randomKe

    auto best_CASE_time_start = std::chrono::high_resoluti
    runKeyInsertionAVLTree(AVLBestCaseTree, bestCaseKeys);
    auto best_CASE_time_End = std::chrono::high_resolution_
    AVLBestCaseTimes += std::chrono::duration_cast<std::chr

    auto stdSetStart = std::chrono::high_resolution_clock:
    for (int key : randomKeys) {
        stdSetContainer.insert(key);
    }
    auto stdSetEnd = std::chrono::high_resolution_clock::no
    stdSetTime += std::chrono::duration_cast<std::chrono::n
```

1.3 Empirical Results

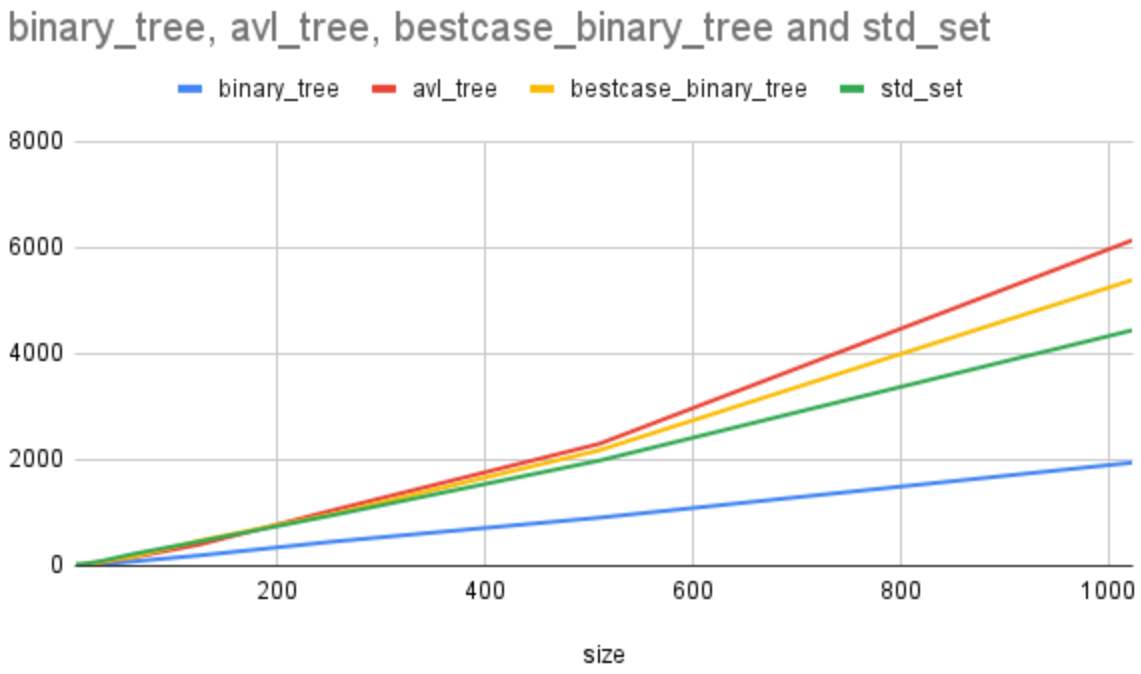
▼ The cumulative running times for each scenario are recorded and presented for different data sizes.

```
std::cout
                << dataSize << " \t\t "
                << binaryTreeTime / iterations << " \t\t\t "
                << avlTreeTime / iterations << " \t\t "
                << AVLBestCaseTimes / iterations << " \t\t\t
                << stdSetTime / iterations << std::endl;
```

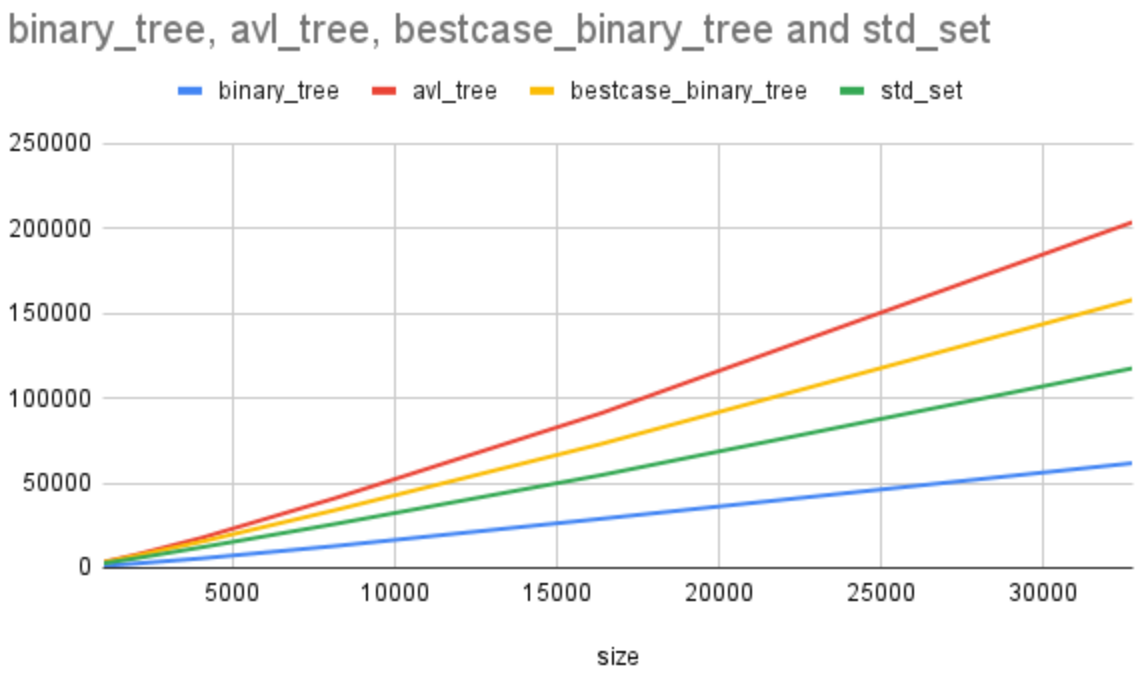
1.4 Graphical Representation

Graphs depicting the performance of each scenario for varying data sizes are provided below:

Small sizes y axis is in nanoseconds:



Big sizes and here y axis in microseconds:



And all the sizes are the numbers n = 2^m - 1.

Task 2: Introduction of AVL Tree

2.1 AVL Tree Implementation

▼ To enhance the analysis, an AVL tree, a self-balancing binary search tree, was introduced. AVL trees automatically maintain a balanced structure during insertions.

```
class AVLTree {
private:
    TreeNode* root;

    int getHeight(TreeNode* node) {
        return (node != nullptr) ? node->height : 0;
    }

    int getBalanceFactor(TreeNode* node) {
        return (node != nullptr) ? getHeight(node->left) - ge
    }
```

```
void updateHeight(TreeNode* node){
    if (node != nullptr){
        node->height = 1 + std::max(getHeight(node->left)
    }
}

TreeNode* rotateRight(TreeNode* y) {
    TreeNode* x = y->left;
    TreeNode* T2 = x->right;

    x->right = y;
    y->left = T2;

    updateHeight(y);
    updateHeight(x);

    return x;
}

TreeNode* rotateLeft(TreeNode* x) {
    TreeNode* y = x->right;
    TreeNode* T2 = y->left;

    y->left = x;
    x->right = T2;

    updateHeight(y);
    updateHeight(x);

    return y;
}

TreeNode* insert(TreeNode* node, int key) {
    if (node == nullptr) {
        return new TreeNode(key);
```

```
        }

        if (key < node->key) {
            node->left = insert(node->left, key);
        } else if (key > node->key) {
            node->right = insert(node->right, key);
        } else {
            // Ignore duplicate keys
            return node;
        }

        updateHeight(node);

        int balance = getBalanceFactor(node);

        // Left Heavy
        if (balance > 1) {
            if (key < node->left->key) {
                return rotateRight(node);
            } else {
                if (node->left->right == nullptr) {
                    return rotateRight(node);
                } else {
                    node->left = rotateLeft(node->left);
                    return rotateRight(node);
                }
            }
        }

        // Right Heavy
        if (balance < -1) {
            if (key > node->right->key) {
                return rotateLeft(node);
            } else {
                if (node->right->left == nullptr) {
                    return rotateLeft(node);
```

```
                } else {
                    node->right = rotateRight(node->right);
                    return rotateLeft(node);
                }
            }
        }

        return node;
    }

    void inOrderTraversal(TreeNode* node) {
        if (node != nullptr) {
            inOrderTraversal(node->left);
            std::cout << node->key << " ";
            inOrderTraversal(node->right);
        }
    }

public:
    AVLTree() : root(nullptr) {}

    void insertKey(int key) {
        root = insert(root, key);
    }

    void traverseInOrder() {
        inOrderTraversal(root);
    }
};
```

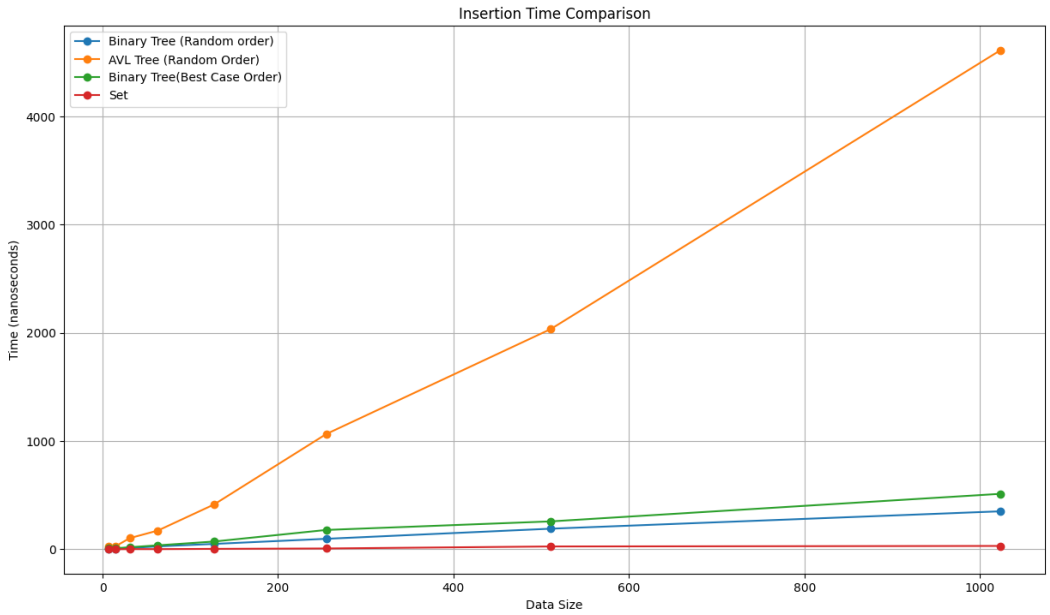
2.2 Enhanced Test Setup

▼ The test setup was enhanced by including the AVL tree in the comparison, evaluating its performance alongside the binary tree and `std::set`

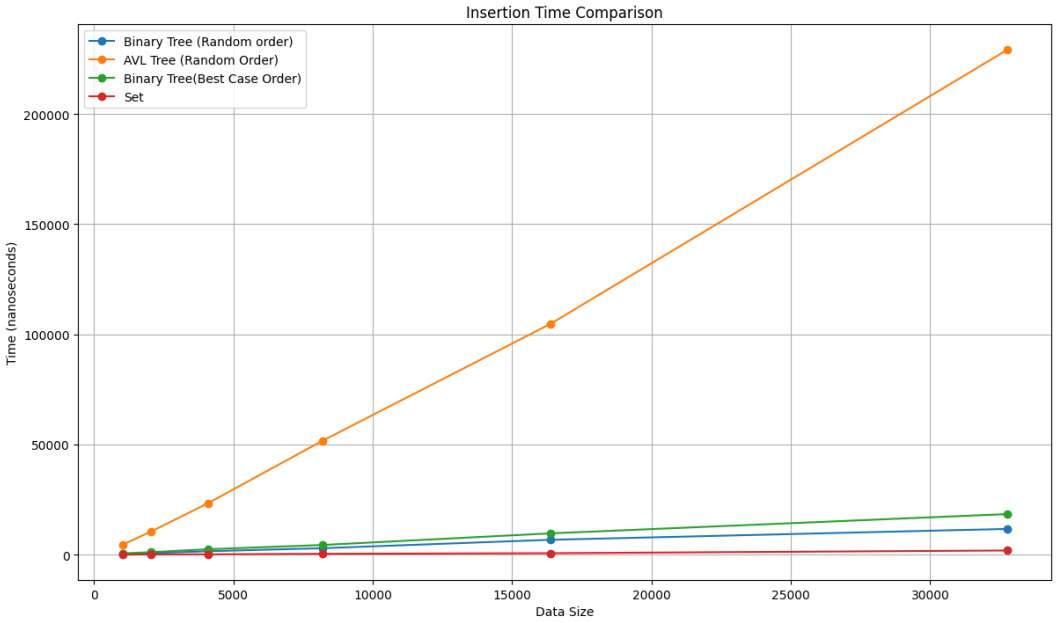
```
auto best_CASE_time_start = std::chrono::high_resolution_cloc
runKeyInsertionAVLTree(AVLBestCaseTree, bestCaseKeys);
auto best_CASE_time_End = std::chrono::high_resolution_clock
AVLBestCaseTimes += std::chrono::duration_cast<std::chrono::r
```

P.S. Here are the graphs of the same algorithm but in python:

Small sizes:



Large sizes:



As we can see the graph result are different, probably it is because of the difference in language

Conclusion

In conclusion, this experiment provides valuable insights into the performance of different tree structures for key insertion. The inclusion of an AVL tree demonstrates the advantages of a self-balancing mechanism, offering improved efficiency compared to unbalanced binary trees. This empirical analysis, accompanied by graphical representations, contributes to the understanding of applied dendrology in computer science, aiding in the selection of appropriate data structures for specific scenarios.