МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Отчет по выполнению практического задания № 7, часть 1

Тема:

«Балансировка дерева поиска»

Дисциплина: «Структуры и алгоритмы обработки данных»

Выполнил студент: Лисовский И.В

Группа: ИКБО-21-23

Москва – 2024

СОДЕРЖАНИЕ

1 ЦЕЛЬ

Разработать и реализовать алгоритм балансировки дерева поиска (AVL-дерево) на языке C++. Оценить эффективность использования различных методов балансировки, сравнить их по временным затратам на операции вставки, удаления и поиска, а также проанализировать влияние баланса на производительность структуры данных.

2 ЗАДАНИЕ

2.1 Формулировка задачи

Составить программу создания двоичного дерева поиска и реализовать процедуры для работы с деревом согласно варианту. Процедуры оформить в виде самостоятельных режимов работы созданного дерева. Выбор режимов производить с помощью пользовательского(иерархического ниспадающего) меню. Провести полное тестирование программы на дереве размером n=10элементов, сформированном вводом с клавиатуры. Тест-примеры определить самостоятельно. Результаты тестирования в виде скриншотов экранов включить в отчет по выполненной работе. Сделать выводы о проделанной работе, основанные на полученных результатах. Оформить отчет с подробным описанием созданного дерева, принципов программной реализации алгоритмов работы с деревом, описанием текста исходного кода и проведенного тестирования программы.

Вариант №19:

Вариант	Тип значения узла	Тип дерева	Реализовать алгоритмы								
			Вставка элемента	Прямой обход	Обратный обход	Симметричный обход	Обход в ширину	Найти сумму значений листьев	Найти среднее арифметическое всех узлов	Найти длину пути от корня до заданного значения	Найти высоту дерева
19	Строка – город	АВЛ-дерево	+ (и балансировка)		+	+				+	+

2.2 Математическая модель решения задачи 1

1. **Определение структуры данных:**

Структура узла дерева поиска (AVL-дерево) содержит данные, указатели на левого и правого потомка, а также высоту узла.

```
struct Node
{
    string data;
    Node* left = nullptr;
    Node* right = nullptr;
    int height = 1;
    Node(string data)
    {
        this->data = data;
    }
};
```

Листинг 1.1 — Структура узлов дерева

2. **Получение высоты узла:**

В этой секции описывается функция, которая вычисляет высоту узла дерева. Высота узла представляет собой длину самого длинного пути от данного узла до его наиболее глубокого листового потомка. Таблица строится следующим образом:

```
int getHeight(Node* node)
{
    if (node == nullptr)
        return 0;
    return node->height;
}
```

Листинг 1.2 — Получение высоты узла

3. **Получение коэффициента баланса:**

Коэффициент баланса узла используется для определения его состояния. Он показывает, насколько сильно узел смещён влево или вправо, и позволяет понять, требуется ли балансировка.

```
int getBalanceFactor(Node* node)
{
    if (node == nullptr)
        return 0;
    return getHeight(node->left) - getHeight(node->right);
}
```

Листинг 1.3 — Функция для нахождения Баланс-фактора

4. **Вращения:**

Вращения — это ключевые операции для поддержания сбалансированности AVL-дерева. Существует четыре основных типа вращений, которые выполняются в зависимости от коэффициента баланса узлов. Это левое вращение, правое вращение, а также комбинированные левое-правое и правое-левое вращения.

Правое вращение выполняется, когда левое поддерево узла высоко, что может привести к дисбалансу. Оно используется для изменения структуры дерева так, чтобы корень стал правым потомком левого поддерева.

```
Node* rightRotate(Node* y)
{
    Node* x = y->left;
    Node* T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(getHeight(y->left), getHeight(y->right)) + 1;
    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

    // Return new root
    return x;
}
```

Листинг 1.4 — Функция для правого вращения

Левое вращение выполняется, когда правое поддерево узла высоко и необходимо скорректировать баланс. Это вращение делает правое поддерево новым корнем узла.

```
Node* leftRotate(Node* x)
{
    Node* y = x->right;
    Node* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;
    y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

    // Return new root
    return y;
}
```

Листинг 1.5 — Функция для левого вращения

Комбинированные вращения происходят в случаях, когда требуется выполнить два вращения. Левое-правое вращение: выполняется, если левое поддерево узла имеет правое поддерево, которое высоко.

```
if (balance > 1 && key > node->left->data)
{
    node->left = leftRotate(node->left);
    return rightRotate(node);
}
```

Листинг 1.6 — Функция для LR вращения

Правое-левое вращение: выполняется, если правое поддерево узла имеет левое поддерево, которое высоко.

```
// Right Left Case
if (balance < -1 && key < node->right->data)
{
    node->right = rightRotate(node->right);
    return leftRotate(node);
}
```

Листинг 1.7 — Функция для RL вращения

5. **Вставка узла:**

Вставка узла в AVL-дерево включает в себя обычную вставку в бинарное дерево поиска, обновление высоты узлов и проверку баланса, чтобы определить, требуется ли выполнить вращение для восстановления сбалансированности дерева. Алгоритм следующий: Если узел пустой (`nullptr`), создаем новый узел с заданным ключом. Сравниваем ключи и вставляем в левое или правое поддерево. Высота узла обновляется на основе высоты его детей. Рассчитываем баланс узла, который показывает, насколько сбалансировано дерево. Если баланс больше 1 и ключ меньше ключа левого сына, выполняем правое вращение. Если баланс меньше -1 и ключ больше ключа правого сына, выполняем левое вращение. Для

случаев «левый правый» и «правый левый» выполняем соответствующие двойные вращения.

```
Node* insert(Node* node, string key)
{
    // 1. Perform normal BST insertion
    if (node == nullptr)
        return new Node(key);

    if (key < node->data)
        node->left = insert(node->left, key);
    else if (key > node->data)
        node->right = insert(node->right, key);
    else // Duplicate keys not allowed
        return node;

    // 2. Update height of this ancestor node
    node->height = 1 + max(getHeight(node->left), getHeight(node->right));

    // 3. Get the balance factor
    int balance = getBalanceFactor(node);

    // 4. Balance the node if it is unbalanced

    // Left Left Case
    if (balance > 1 && key < node->left->data)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->data)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->data)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->data)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
    // Return the (unchanged) node pointer
    return node;
}
```

Листинг 1.8 — Функция для левого вращения

2.3 Код программы с комментариями

```
#include <iostream>
#include <string>
#include <windows.h>
using namespace std;

struct Node
{
    string data;
    Node* left = nullptr;
    Node* right = nullptr;
    int height = 1;
    Node(string data)
    {
        this->data = data;
    }
};

int getHeight(Node* node)
{
    if (node == nullptr)
        return 0;
    return node->height;
}

int getBalanceFactor(Node* node)
{
    if (node == nullptr)
        return 0;
    return getHeight(node->left) - getHeight(node->right);
}

Node* rightRotate(Node* y)
{
    Node* x = y->left;
    Node* T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(getHeight(y->left), getHeight(y->right)) + 1;
    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;

    // Return new root
    return x;
}

Node* leftRotate(Node* x)
{
    Node* y = x->right;
    Node* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(getHeight(x->left), getHeight(x->right)) + 1;
    y->height = max(getHeight(y->left), getHeight(y->right)) + 1;

    // Return new root
    return y;
```

```
}

Node* insert(Node* node, string key)
{
    // 1. Perform normal BST insertion
    if (node == nullptr)
        return new Node(key);

    if (key < node->data)
        node->left = insert(node->left, key);
    else if (key > node->data)
        node->right = insert(node->right, key);
    else // Duplicate keys not allowed
        return node;

    // 2. Update height of this ancestor node
    node->height = 1 + max(getHeight(node->left), getHeight(node->right));

    // 3. Get the balance factor
    int balance = getBalanceFactor(node);

    // 4. Balance the node if it is unbalanced

    // Left Left Case
    if (balance > 1 && key < node->left->data)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->data)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->data)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->data)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
    // Return the (unchanged) node pointer
    return node;
}

void printTree(Node* node, int space = 0, int height = 10)
{
    if (node == nullptr)
        return;

    space += height;

    // Process right child first
    printTree(node->right, space);

    cout << endl;
    for (int i = height; i < space; i++)
        cout << " ";
    cout << node->data << "\n";

    // Process left child
    printTree(node->left, space);
```

```
}

void inOrderTraversal(Node* node)
{
    if (node == nullptr)
        return;
    inOrderTraversal(node->left);
    cout << node->data << " ";
    inOrderTraversal(node->right);
}

void postOrderTraversal(Node* node)
{
    if (node == nullptr)
        return;
    postOrderTraversal(node->left);
    postOrderTraversal(node->right);
    cout << node->data << " ";
}

int pathLength(Node* node, string value)
{
    if (node == nullptr)
        return -1;
    if (node->data == value)
        return 0;
    else if (value < node->data)
    {
        int left = pathLength(node->left, value);
        if (left >= 0)
            return left + 1;
        else
            return -1;
    }
    else
    {
        int right = pathLength(node->right, value);
        if (right >= 0)
            return right + 1;
        else
            return -1;
    }
}

int treeHeight(Node* node)
{
    return getHeight(node);
}

int main()
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    Node* parent = nullptr;
    char menu;
    parent = insert(parent, "Moscow");
    parent = insert(parent, "Kazan");
    parent = insert(parent, "Kaliningrad");
    parent = insert(parent, "Pskov");
    parent = insert(parent, "Paris");
    parent = insert(parent, "Tombov");
    parent = insert(parent, "Evpatoria");
    string data;
    while (true)
    {
```

```
        cout << "1. Add 2. Print 3. In-order traversal 4. Post-order traversal 5.
Path length to value 6. Tree height\nCommand: ";
        cin >> menu;
        switch (menu)
        {
        case '1':
            cout << "data? : ";
            cin >> data;
            parent = insert(parent, data); // Assign the returned node to parent
            cout << "Successfully! Here is the tree:\n\n\n";
            printTree(parent);
            cout << "\n\n\n";
            break;
        case '2':
            cout << "Tree (upper - right, lower - left)\n\n\n";
            printTree(parent);
            cout << "\n\n\n";
            break;
        case '3':
            cout << "In-order traversal: ";
            inOrderTraversal(parent);
            cout << "\n\n\n";
            break;
        case '4':
            cout << "Post-order traversal: ";
            postOrderTraversal(parent);
            cout << "\n\n\n";
            break;
        case '5':
            cout << "Value? : ";
            cin >> data;
            {
                int length = pathLength(parent, data);
                if (length >= 0)
                    cout << "Length of path from root to " << data << " is " <<
length << "\n\n\n";
                else
                    cout << "Value not found in tree.\n\n\n";
            }
            break;
        case '6':
            cout << "Height of the tree is: " << treeHeight(parent) << "\n\n\n";
            break;
        default:
            cout << "Invalid command\n\n\n";
            break;
        }
    }
    return 0;
}
```

Листинг 2 — Код программы

2.4 Тестирование программы

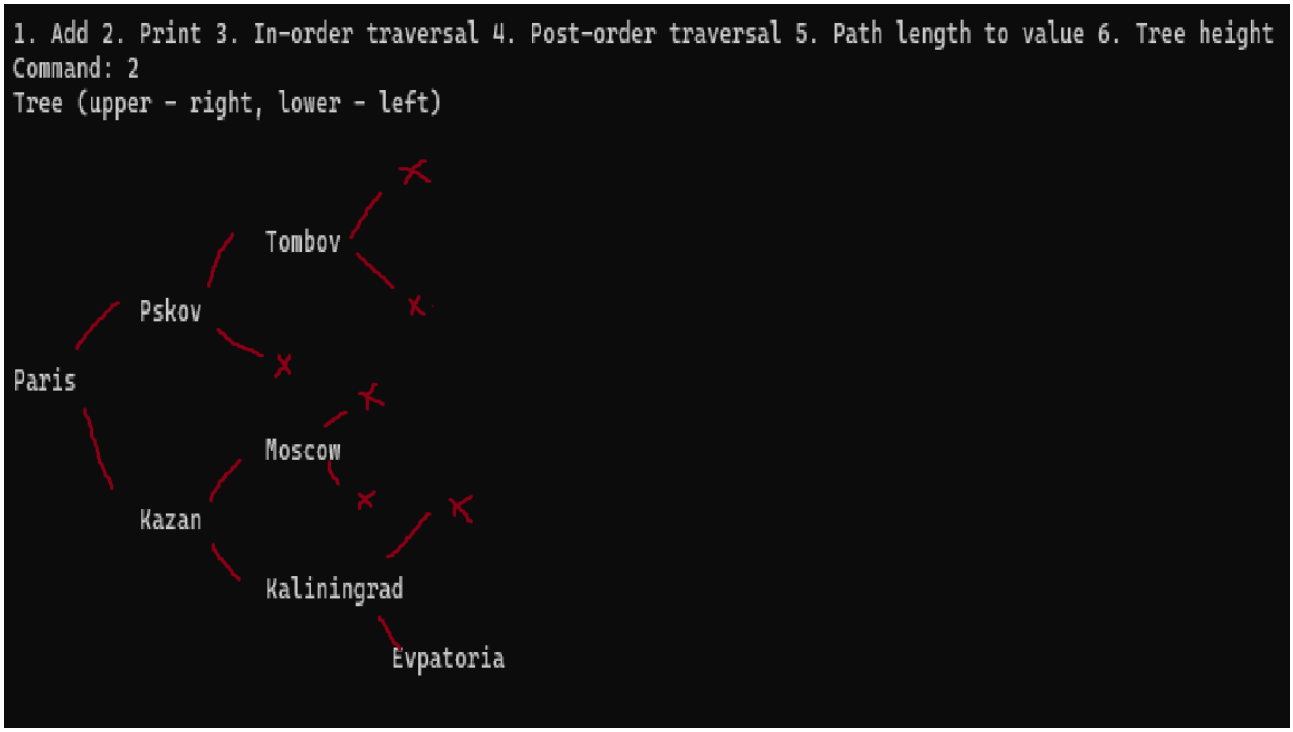

Рисунок 1 — Тестирование вывода дерева

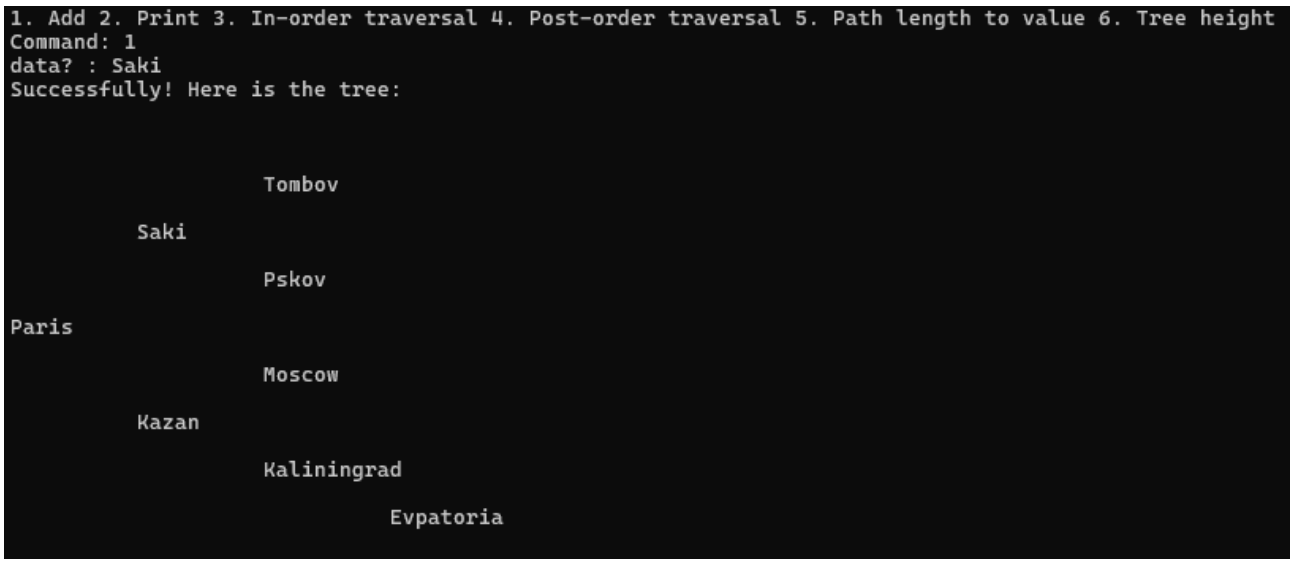

Рисунок 2 — Тестирование добавления узла

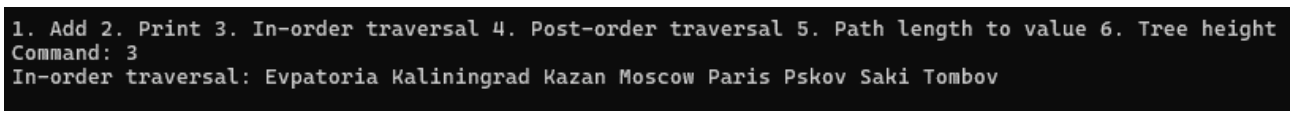

Рисунок 3 — Тестирование симметричного обхода дерева

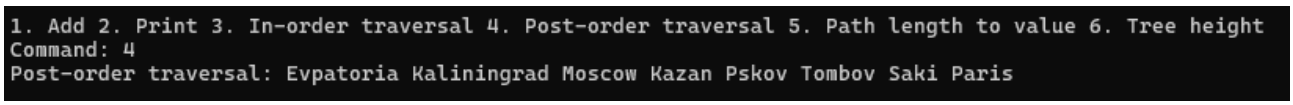

Рисунок 4 — Тестирование обратного обхода дерева

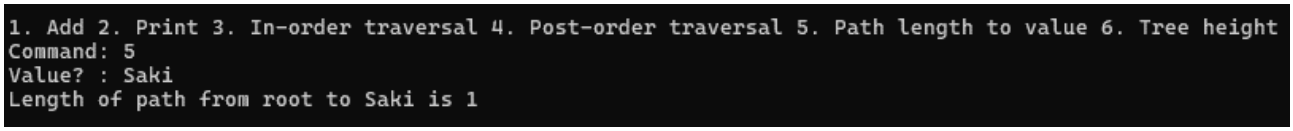

Рисунок 5 — Тестирование вывода длины пути до узла

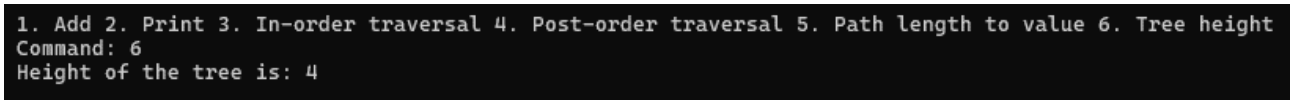

Рисунок 6 — Тестирование вывода высоты дерева

3 ВЫВОД

В данной работе была исследована и реализована структура данных AVL-дерево, которая представляет собой самобалансирующееся бинарное дерево поиска. Основными целями работы были:

1. **Разработка алгоритма вставки узлов**: Реализованный алгоритм обеспечивает сбалансированную вставку, что позволяет поддерживать логарифмическое время выполнения операций вставки, удаления и поиска. Благодаря использованию ротаций, AVL-дерево сохраняет свои свойства даже после модификаций.
2. **Анализ эффективности алгоритма**:
    - **Временная сложность**: Временная сложность основных операций (вставка, удаление, поиск) в AVL-дереве составляет O(logn), где n — количество узлов в дереве. Это обеспечивается поддержанием сбалансированности дерева, что минимизирует его высоту.
3. **Сравнение с другими структурами данных**: В отличие от простых бинарных деревьев поиска, которые могут деградировать до линейной структуры (в худшем случае), AVL-деревья гарантируют более высокую эффективность за счёт автоматической балансировки. Заведомо сбалансированные деревья обеспечивают стабильную производительность по времени.
4. **Преимущества и недостатки**:
    - Основные преимущества заключаются в быстром выполнении операций благодаря сбалансированности.
    - Недостатком может быть сложность реализации ротаций и поддержание баланса при частых изменениях.

Работа продемонстрировала практическую реализацию AVL-дерева с использованием языка C++, а также успешное применение основных принципов работы с деревьями. Исходя из проведённого анализа, можно заключить, что

AVL-дерево является высокоэффективной структурой данных для хранения и обработки отсортированных данных с минимальными временными затратами на основные операции.

Таким образом, изучение и внедрение алгоритмов, обеспечивающих балансировку деревьев, является важным направлением в области структур данных и алгоритмов, позволяющим достигать высокого уровня производительности в приложениях, требующих частого доступа и модификации хранимой информации.

4 ЛИТЕРАТУРА

1. Вирт Н. Алгоритмы и структуры данных. Новая версия для Оберона, 2010.
2. Кнут Д. Искусство программирования. Тома 1-4, 1976-2013.
3. Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих, 2017.
4. Кормен Т.Х. и др. Алгоритмы. Построение и анализ, 2013.
5. Лафоре Р. Структуры данных и алгоритмы в Java. 2-е изд., 2013.
6. Макконнелл Дж. Основы современных алгоритмов. Активный обучающий метод. 3-е доп. изд., 2018.
7. Скиена С. Алгоритмы. Руководство по разработке, 2011.
8. Хайнеман Д. и др. Алгоритмы. Справочник с примерами на C, C++, Java и Python, 2017.
9. Гасфилд Д. Строки, деревья и последовательности в алгоритмах. Информатика и вычислительная биология, 2003.

По языку C++:

10. Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд., 2016.
11. Павловская Т.А. C/C++. Программирование на языке высокого уровня, 2003.
12. Прата С. Язык программирования C++. Лекции и упражнения. - 6-е изд., 2012.
13. Седжвик Р. Фундаментальные алгоритмы на C++, 2001-2002
14. Хортон А. Visual C++ 2010. Полный курс, 2011.
15. Шилдт Г. Полный справочник по C++. 4-е изд., 2006.