Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Российский химико-технологический университет имени Д.И. Менделеева»
Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 6

Выполнил студент группы.....КС-38.....(Нергарян Геворг Гарегинович)
Ссылка на репозиторий: .....(https://github.com/MUCTR-IKT-CPP/Nergaryan_KC-38_Algos)

Приняли: .....Пысин Максим Дмитриевич
.....Краснов Дмитрий Олегович
.....Лобанов Алексей Владимирович
.....Крашенинников Роман Сергеевич

Дата сдачи: …….23.05.2023……..

Оглавление

Описание задачи.

В рамках лабораторной работы необходимо изучить и реализовать бинарное дерево поиска и его самобалансирующийся вариант в лице AVL дерева.

Для проверки анализа работы структуры данных требуется провести 10 серий тестов.

- В каждой серии тестов требуется выполнять 20 циклов генерации и операций. При этом первые 10 работают с массивом заполненным случайным образом, во второй половине случаев, массив заполняется в порядке возрастания значений индекса, т.е. является отсортированным по умолчанию.
- Требуется создать массив, состоящий из $2^{(10 + i)}$ элементов, где i это номер серии.
- Массив должен быть помещен в оба вариант двоичных деревьев. При этому замеряется время, затраченное на всю операцию вставки всего массива.
- После заполнения массива, требуется выполнить 1000 операций поиска по обоим вариантам дерева, случайного числа в диапазоне генерируемых значений, замерев время на все 1000 попыток и вычислив время 1 операции поиска.
- Провести 1000 операций поиска по массиву, замерить требуемое время на все 1000 операций и найти время на 1 операцию.
- После, требуется выполнить 1000 операций удаления значений из двоичных деревьев, и замерить время, затраченное на все операции, после чего вычислить время на 1 операцию.
- После выполнения всех серий тестов, требуется построить графики зависимости времени затрачиваемого на операции вставки, поиска, удаления от количества элементов. При этом требуется разделить графики для отсортированного набора данных и заполненных со случайным распределением. Так же, для операции поиска, требуется так же нанести для сравнения график времени поиска для обычного массива.

Описание метода/модели.

Бинарное дерево - это структура данных, которая состоит из узлов, связанных между собой в виде иерархической структуры. Особенностью бинарного дерева является то, что каждый узел может иметь не более двух потомков - левого и правого.

Основные компоненты бинарного дерева:

1. Узел (Node): Каждый узел в бинарном дереве содержит информацию (значение) и ссылки на левого и правого потомков. Узлы можно представить в виде объектов или структур, содержащих данные и указатели на потомков.
2. Корень (Root): Корень бинарного дерева - это основной (верхний) узел дерева, от которого начинается построение и навигация по дереву. Корень не имеет родительского узла.
3. Левый и правый потомок: Каждый узел в бинарном дереве может иметь до двух потомков. Потомки, расположенные слева от родительского узла, называются левым потомком, а расположенные справа - правым потомком.
4. Лист (Leaf): Листом называется узел, который не имеет ни левого, ни правого потомка. Листья находятся в конце каждой ветви дерева.
5. Поддерево: Поддерево - это часть дерева, состоящая из узла и всех его потомков, включая их потомков.

Бинарные деревья широко применяются в компьютерных науках и информатике. Они обеспечивают эффективную структуру для хранения и организации данных, позволяют эффективно выполнять операции поиска, вставки и удаления элементов. Примеры применения бинарных деревьев включают поиск и сортировку данных, реализацию алгоритмов обхода дерева (например, прямой, обратный и симметричный обход) и построение арифметических выражений.

AVL-дерево - это особый тип сбалансированного бинарного дерева, которое обеспечивает эффективные операции вставки, удаления и поиска элементов. Оно получило свое название от своих создателей, Адельсона-Вельского и Ландиса.

Основная особенность AVL-дерева заключается в его сбалансированности. Сбалансированность означает, что высота левого и правого поддеревьев каждого узла может различаться не более чем на 1. Это свойство позволяет достичь высокой эффективности операций, так как гарантируется, что высота дерева будет ограничена логарифмически.

Каждый узел AVL-дерева имеет дополнительное поле, называемое "фактор баланса" (balance factor). Фактор баланса вычисляется как разность высоты правого поддерева и высоты левого поддерева. Значение фактора баланса может быть -1, 0 или 1 для сбалансированного дерева.

Операции вставки и удаления в AVL-дереве осуществляются таким образом, чтобы сохранить его сбалансированность. При вставке или удалении узла могут возникать ситуации, когда факторы баланса становятся неправильными и нарушается баланс дерева. В таких случаях выполняются повороты (rotations) - операции перебалансировки поддерева, которые изменяют структуру дерева, чтобы восстановить его сбалансированность.

Преимущества AVL-дерева:

1. Обеспечивает быстрые операции вставки, удаления и поиска элементов, имея высоту ограниченную логарифмически.
2. Гарантирует сбалансированность, что позволяет предотвратить худший случай времени выполнения операций на бинарном дереве.

Ограничения AVL-дерева:

1. В сравнении с другими типами бинарных деревьев, AVL-дерево требует дополнительных операций для поддержания баланса, что может приводить к небольшому снижению производительности.
2. Занимает больше памяти из-за дополнительного поля фактора баланса в каждом узле.

В целом, AVL-дерево является эффективной структурой данных для операций поиска, вставки и удаления, особенно в случаях, когда требуется гарантированная балансировка дерева и ограниченное время выполнения операций.

Выполнение задачи.

Код на Языке C++

```
#include <iostream>
#include <vector>
#include <cmath>
#include <algorithm>
#include <chrono>
#include <fstream>
#include <time.h>
#include <string>

using namespace std;

//Узел дерева
struct Node {
    int data;
    //левый\правый подузел
    Node* left;
    Node* right;
    //высота дерева
    int height;
    Node(int data) : data(data), left(nullptr), right(nullptr), height(1) {}
};



//Бинарное дерево
class BinarySearchTree {
private:
    Node* root;//дерево

    //Вставка
    void insert(Node* node, int data) { //node узел,data число для вставки
        //через сравнения числа со значением текущего узла находим место для вставки
        if (data < node->data) {
            if (node->left == nullptr) {
```

```
            node->left = new Node(data);
        }
        else {
            insert(node->left, data);
        }
    }
    else if (data > node->data) {
        if (node->right == nullptr) {
            node->right = new Node(data);
        }
        else {
            insert(node->right, data);
        }
    }
}

//Поиск
bool search(Node* node, int data) {
    //через сравнения числа со значением текущего узла делаем поиск числа в дереве
    if (node == nullptr) {
        return false;
    }
    else if (data < node->data) {
        return search(node->left, data);
    }
    else if (data > node->data) {
        return search(node->right, data);
    }
    else {
        return true;
    }
}
//Удаление
void remove(Node* node, int data) {
    if (node == nullptr) {
        return;
    }
```

```
        else if (data < node->data) {
            remove(node->left, data);
        }
        else if (data > node->data) {
            remove(node->right, data);
        }
        else {
            if (node->left == nullptr && node->right == nullptr) {
                node = nullptr;
            }
            else if (node->left != nullptr && node->right != nullptr) {
                node->data = minNode(node->right)->data;
                remove(node->right, node->data);
            }
            else {
                node = (node->left != nullptr) ? node->left : node->right;
            }
        }
    }

    //минимальный узел по левой стороне
    Node* minNode(Node* node) {
        return (node->left != nullptr) ? minNode(node->left) : node;
    }
    // удаления всех узлов в бинарном дереве, начиная с заданного узла
    void destroyTree(Node* node) {
        if (node != nullptr) {
            destroyTree(node->left);
            destroyTree(node->right);
            delete node;
        }
    }

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

    //Вставка
```

```
    void insert(int data) {
        if (root == nullptr) {
            root = new Node(data);
        }
        else {
            insert(root, data);
        }
    }

    //Поиск
    bool search(int data) {
        return search(root, data);
    }



    //Удаление
    void remove(int data) {
        remove(root, data);
    }

    ~BinarySearchTree() {
        destroyTree(root);
    }
};

class AVLTree {
private:
    Node* root;
    // Возвращение высоты
    int height(Node* node) {
        return node ? node->height : 0;
    }

    //Правка высоты
    void fixHeight(Node* node) {
        int hl = height(node->left);
        int hr = height(node->right);
```

```
    node->height = (hl > hr ? hl : hr) + 1;
}

//Правый и левый повороты
Node* rotateRight(Node* node)
{
    Node* left = node->left;
    node->left = left->right;
    left->right = node;
    fixHeight(node);
    fixHeight(left);
    return left;
}
Node* rotateLeft(Node* node)
{
    Node* right = node->right;
    node->right = right->left;
    right->left = node;
    fixHeight(node);
    fixHeight(right);
    return right;
}

//Фактор баланса
int balanceFactor(Node* node) {
    return height(node->right) - height(node->left);
}
//Балансировка
Node* balance(Node* node)
{
    fixHeight(node);
    if (balanceFactor(node) == 2)
    {
        if (balanceFactor(node->right) < 0)
            node->right = rotateRight(node->right);
        return rotateLeft(node);
    }
```

```
    if (balanceFactor(node) == -2)
    {
        if (balanceFactor(node->left) > 0)
            node->left = rotateLeft(node->left);
        return rotateRight(node);
    }
    return node;
}


Node* insert(Node* node, int data) {
    //Находим место для вставки числа
    if (node == nullptr)
        return new Node(data);
    if (data < node->data) {
        node->left = insert(node->left, data);
    }
    else
        node->right = insert(node->right, data);
    //отправляем в функцию балансировки
    return balance(node);
}

//Поиск по дереву
bool search(Node* node, int data) {
    //Если меньше текущего узла то отправляемся в его левый узел иначе в правый
    if (node == nullptr) {
        return false;
    }
    else if (data < node->data) {
        return search(node->left, data);
    }
    else if (data > node->data) {
        return search(node->right, data);
    }
    else {
        return true;
```

```
    }
  }

  //Поиск минимума
  Node* findMin(Node* node)
  {
    return (node->left != nullptr) ? findMin(node->left) : node;
  }

  //Удаление минимального элемента
  // По свойству АВЛ-дерева у минимального элемента справа либо подвешен единственный узел, либо там пусто
  Node* removeMin(Node* node)
  {
    if (node->left == nullptr)
      return node->right;
    node->left = removeMin(node->left);
    return balance(node);
  }

  // Удаление узла
  Node* remove(Node* node, int data)
  {
    if (node == nullptr)
      return 0;
    if (data < node->data)
      node->left = remove(node->left, data);
    else if (data > node->data)
      node->right = remove(node->right, data);
    else
    {
      Node* left = node->left;
      Node* right = node->right;
      delete node;
      if (right == nullptr)
        return left;
      Node* min = findMin(right);
```

```
            min->right = removeMin(right);
            min->left = left;
            return balance(min);
        }
        return balance(node);
    }

public:

    void insert(int data) {
        if (root == nullptr)
            root = new Node(data);
        else
            root = insert(root, data);
    }


    bool search(int data) {
        return search(root, data);
    }

    void remove(int data) {
        root = remove(root, data);
    }
};



vector<int> generateNums(int nums, bool sorted) {
    vector<int> values;
    for (int num = 0; num < nums; num++)
        values.push_back(1 + rand() % 1000);

    if (sorted)
        sort(values.begin(), values.end());
    return values;
}
```

```
void doForBinTree(ofstream& out, vector<int> values) {
    BinarySearchTree* bin_tree = new BinarySearchTree();
    chrono::high_resolution_clock::time_point start = chrono::high_resolution_clock::now();
    for (int num : values)
        bin_tree->insert(num);
    chrono::high_resolution_clock::time_point end = chrono::high_resolution_clock::now();
    chrono::duration<double> sec_diff = end - start;
    out << sec_diff.count() << " ";

    start = chrono::high_resolution_clock::now();
    for (int k = 0; k < 1000; k++) {
        int num = 1 + rand() % 1000;
        bin_tree->search(num);
    }
    end = chrono::high_resolution_clock::now();
    sec_diff = end - start;
    out << sec_diff.count() << " ";

    start = chrono::high_resolution_clock::now();
    for (int k = 0; k < 1000; k++) {
        int num = 1 + rand() % 1000;
        bin_tree->remove(num);
    }
    end = chrono::high_resolution_clock::now();
    sec_diff = end - start;
    out << sec_diff.count() << endl;
    delete bin_tree;
}

void doForAVLTree(ofstream& out, vector<int> values) {
    AVLTree* avl_tree = new AVLTree();
    chrono::high_resolution_clock::time_point start = chrono::high_resolution_clock::now();
    for (int num : values)
        avl_tree->insert(num);
    chrono::high_resolution_clock::time_point end = chrono::high_resolution_clock::now();
    chrono::duration<double> sec_diff = end - start;
```

```
    out << sec_diff.count() << " ";

    start = chrono::high_resolution_clock::now();
    for (int k = 0; k < 1000; k++) {
        int num = rand() % 1 + rand() % 1000;
        avl_tree->search(num);
    }
    end = chrono::high_resolution_clock::now();
    sec_diff = end - start;
    out << sec_diff.count() << " ";

    start = chrono::high_resolution_clock::now();
    for (int k = 0; k < 1000; k++) {
        int num = rand() % 1 + rand() % 1000;
        avl_tree->remove(num);
    }
    end = chrono::high_resolution_clock::now();
    sec_diff = end - start;
    out << sec_diff.count() << endl;
    delete avl_tree;
}

void doForVector(ofstream& out, vector<int> values) {
    chrono::high_resolution_clock::time_point start = chrono::high_resolution_clock::now();
    for (int k = 0; k < 1000; k++) {
        int num = rand() % 1 + rand() % 1000;
        find(values.begin(), values.end(), num);
    }
    chrono::high_resolution_clock::time_point end = chrono::high_resolution_clock::now();
    chrono::duration<double> sec_diff = end - start;
    out << sec_diff.count() << "\n";
}

int main()
{
    srand(time(0));
    chrono::high_resolution_clock::time_point start;
```

```
    chrono::high_resolution_clock::time_point end;
    chrono::duration<double> sec_diff;

    ofstream bin_out;
    bin_out.open("BinTree.txt");
    ofstream avl_out;
    avl_out.open("AVLTree.txt");
    ofstream arr_search;
    arr_search.open("Search.txt");

    vector<int> values;
    for (int test = 1; test <= 10; test++) {
        bin_out << "Test " << test << endl;
        avl_out << "Test " << test << endl;
        arr_search << "Test " << test << endl;

        int nums = pow(2, 10 + test);
        bool checker = false;
        for (int i = 0; i < 20; i++) {
            vector<int> values = (i < 10) ? generateNums(nums, false) : generateNums(nums, true);

            if (i > 9 && !checker) {
                checker = true;
                bin_out << "Sorted\n";
                avl_out << "Sorted\n";
                arr_search << "Sorted\n";
            }
            doForBinTree(bin_out, values);
            doForAVLTree(avl_out, values);
            doForVector(arr_search, values);
        }
    }
    bin_out.close();
    avl_out.close();
    arr_search.close();
}
```

В результате выполнения программы были получены значения времени. Графики получились следующими:

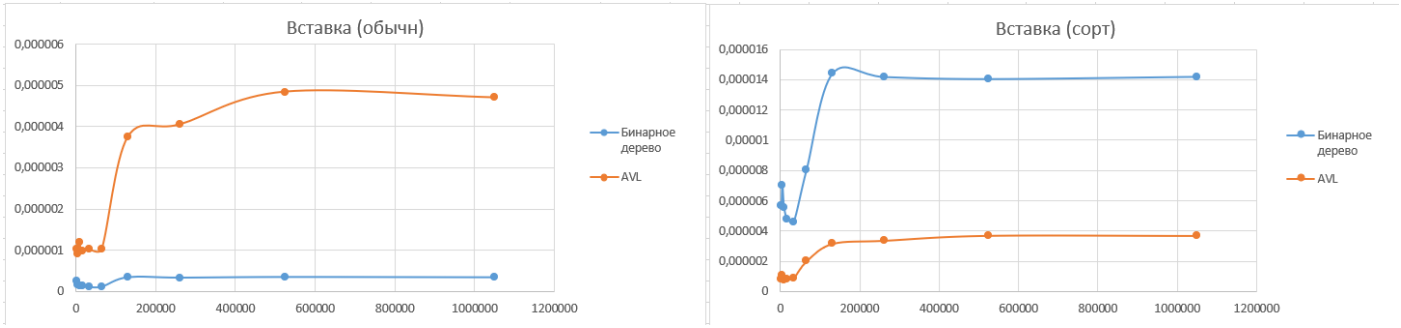

График 1, Вставка

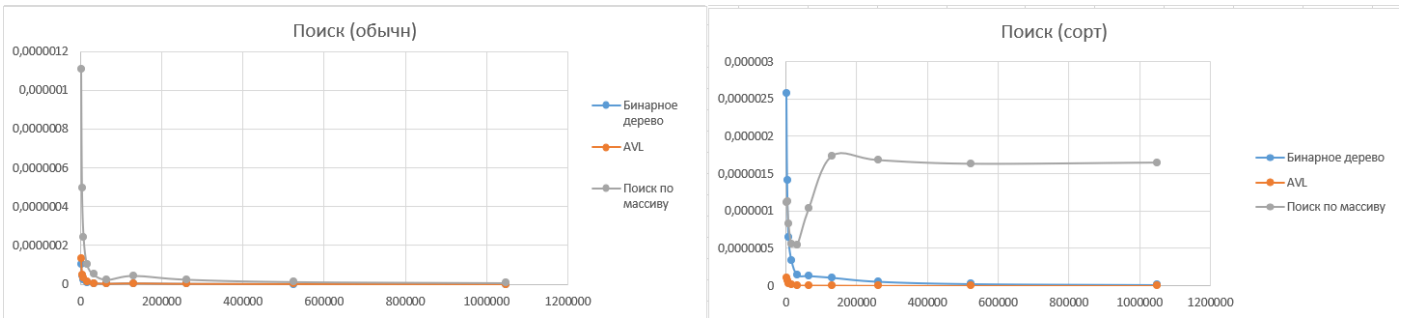

График 2, Поиск

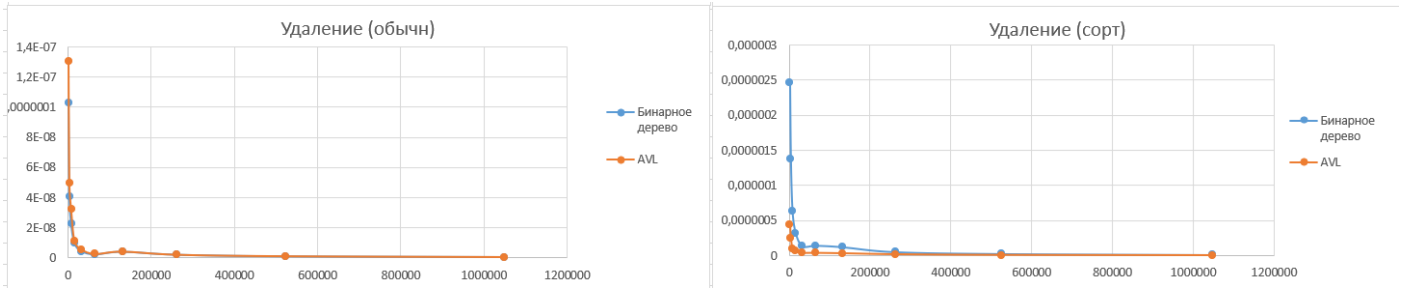

График 3, Удаление

Заключение.

В заключение, результаты исследования показывают, что выбор структуры данных для операций вставки, поиска и удаления зависит от особенностей данных, с которыми мы работаем. Обычное бинарное дерево без балансировки проявляет себя наилучшим образом при операциях вставки с использованием случайно сгенерированного массива, так как не требуется дополнительных операций для поддержания баланса. Однако, при работе с отсортированным массивом рекомендуется использовать AVL-дерево, чтобы гарантировать сбалансированность и обеспечить эффективность операций.

При выполнении поиска в обычном бинарном дереве и AVL-дереве время выполнения примерно одинаковое и значительно лучше, чем при поиске в массиве, особенно при случайно сгенерированном массиве. Однако, на отсортированном массиве предпочтительнее использовать AVL-дерево, так как классическое бинарное дерево может столкнуться с худшим случаем, требуя прохода по всем элементам до последнего.

При удалении элементов разница между обычным бинарным деревом и AVL-деревом не слишком велика при случайно сгенерированном массиве. Однако, на отсортированном массиве AVL-дерево демонстрирует лучшую производительность, особенно при меньшем количестве узлов.

Таким образом, для выбора подходящей структуры данных необходимо учитывать характеристики входных данных и требования к операциям, исходя из которых можно определить, какое дерево будет наиболее эффективным в конкретной ситуации.