МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №3
по дисциплине «Алгоритмы и структуры данных»
Тема: Реализация и исследование АВЛ-деревьев.

Студент гр. 3341 Костромитин М.М.

Преподаватель Иванов Д.В.

Санкт-Петербург
2024

Цель работы

Цель данной лабораторной работы заключается в углубленном изучении структуры данных, а именно AVL-дереве, и реализации основных операций с ним. В ходе работы необходимо:

1. Реализовать функции проверки, является ли дерево АВЛ-деревом, а также функции для нахождения разницы между связанными узлами и вставки новых узлов.

2. Провести исследование, реализовав функции удаления узлов (любого, максимального и минимального), и сравнить время и количество операций, необходимых для выполнения этих действий, с теоретическими оценками на различных объемах данных.

3. При подготовке к очной защите создать визуализацию дерева, что позволит наглядно продемонстрировать работу реализованных функций.

Задание

В предыдущих лабораторных работах вы уже проводили исследования и эта не будет исключением. Как и в прошлые разы лабораторную работу можно разделить на две части:

1) решение задач на платформе moodle

2) исследование по заданной теме

В заданиях в качестве подсказки будет изложена основная структура данных (класс узла) и будет необходимо реализовать несколько основных функций: проверка дерева (является ли оно АВЛ деревом), нахождение разницы между связными узлами, вставка узла.

В качестве исследования нужно самостоятельно:

реализовать функции удаления узлов: любого, максимального и минимального

сравнить время и количество операций, необходимых для реализованных операций, с теоретическими оценками (очевидно, что проводить исследования необходимо на разных объемах данных)

Также для очной защиты необходимо подготовить визуализацию дерева.

В отчете помимо проведенного исследования необходимо приложить код всей получившей структуры: класс узла и функции.

Выполнение работы

Реализация AVL-дерева: class AVLtree

Процесс реализации AVL-дерева включает следующие этапы:

1. class Node - узел содержит значение value, высоту height, и указатели на левое и правое поддеревья (left и right).

Node(int val) - конструктор, который инициализирует узел с заданным значением и высотой 1, считая его новым узлом (листьевым, без потомков).

Методы класса:

1. Node* insert(int val, Node* root) – вставка элемента.

Вставляет новое значение в дерево. Если корень nullptr, создается новый узел. Рекурсивно находит подходящее место для вставки и сбалансирует дерево после вставки.

2. void remove(int value) – удаление элемента.

Рекурсивно ищется узел с заданным значением. Если есть правый потомок, заменяем удаленный узел минимальным элементом из правого поддерева (для поддержания упорядоченности). После удаления вызывается балансировка.

3. Node* balance(Node* parent) – балансировка дерева.

Балансирует дерево, обновляя высоты узлов и выполняя необходимые вращения (левое или правое) для поддержания свойств AVL дерева.

4. void updateHeight(Node* root) - обновляет высоту узла, устанавливая её как максимальную высоту между левым и правым потомками плюс один.

Повороты дерева:

5. Node* smallLeftRotation(Node* root) - Проводит малый левый поворот вокруг узла root. Изменяет ссылки на поддеревья и обновляет высоты узлов.
6. Node* smallRightRotation(Node* root) - Проводит малый правый поворот вокруг узла root. Аналогично малому левому повороту, изменяет ссылки и обновляет высоты.

7. Node* bigLeftRotation(Node* root) - Проводит большой левый поворот, сначала выполняя малый правый поворот для правого поддерева, а затем малый левый поворот для текущего узла.
8. Node* bigRightRotation(Node* root) - Проводит большой правый поворот, сначала выполняя малый левый поворот для левого поддерева, а затем малый правый поворот для текущего узла.

Обходы и визуализация:

Реализованы следующие обходы:

void inOrder(Node* root, std::vector<int>& result) — изменяет массив result, который соответствует in-order проходу.

void preOrder(Node* root, std::vector<int>& result) — изменяет массив result, который соответствует pre-order проходу.

void postOrder(Node* root, std::vector<int>& result) — изменяет массив result, который соответствует post-order проходу.

Реализован метод print выводит дерево в консоль в виде иерархической структуры.

Тестирование

Оценка сложности алгоритмов:

Лучший случай — O(log n)

Средний случай — O(log n)

Худший случай — O(log n)

Результаты тестирования

```
1000 100
Inserting - 100: Base - 1000: Time - 0.089 ms
```

```
10000 1000
Inserting - 1000: Base - 10000: Time - 0.76 ms
```

```
100000 10000
Inserting - 10000: Base - 100000: Time - 11.151 ms
```

```
1000000 100000
Inserting - 100000: Base - 1000000: Time - 93.289 ms
```

```
10000000 1000000
Inserting - 1000000: Base - 10000000: Time - 833.663 ms
```

Выводы

В ходе выполнения лабораторной работы была создана структура данных AVL-дерева, и проведены замеры основных операций для различных объемов данных. На основе полученных результатов можно выделить следующие выводы:

1. Эффективность AVL-дерева: Исследование подтвердило, что AVL-дерево представляет собой высокоэффективный инструмент для выполнения операций вставки и удаления. Среднее и худшее время выполнения этих операций не превышает O(log n), что соответствует теоретическим ожиданиям.

2. Скорость выполнения операций: Операции вставки и удаления по значению выполнялись за миллисекунды, даже при увеличении объема данных до 100000 элементов.

3. Влияние размера данных на производительность: С увеличением количества элементов время выполнения операций, конечно, увеличивалось, однако оставалось на приемлемом уровне. В частности, время, затрачиваемое на удаление минимальных и максимальных значений, значительно возросло при работе с объемами данных до 100000 элементов. Это указывает на важность учета структуры дерева при выполнении подобных операций, и можно предположить, что на данный рост также повлияли особенности реализации балансировки и использование рекурсивных методов.

В целом, результаты лабораторной работы подтверждают высокую производительность AVL-деревьев как ключевого элемента в области структур данных и эффективной обработки информации.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Node.h

```
#ifndef NODE
#define NODE


class Node {
public:
int val;
Node* left;
Node* right;
int height;
Node(int val, Node* left = nullptr, Node* right = nullptr) :
      val(val), left(left), right(right), height(1)
{}
};

#endif
```

Название файла: AVLtree.hpp

```
#ifndef AVLtree
#define AVLtree
#include <iostream>
#include <vector>
#include <algorithm>
#include "NODE.hpp"


class AVLtree {
private:
Node* root;
public:
AVLtree();


AVLtree(std::vector<int> list);


    ~AVLtree();


int getHeight(Node* root);



void updateHeight(Node* root);
```

```
int heightDiff(Node* root);

bool checkIsAVL(Node* root);

int minNodeDiff(Node* root);

Node* createTree(std::vector<int>& list, int begin, int end);

Node* balance(Node* root);

Node* insert(int val, Node* root);

Node* smallLeftRotation(Node* root);

Node* bigLeftRotation(Node* root);

Node* smallRightRotation(Node* root);

Node* bigRightRotation(Node* root);

void inOrder(Node* root, std::vector<int>& result);

void preOrder(Node* root, std::vector<int>& result);

void postOrder(Node* root, std::vector<int>& result);
```

```
    Node* searchMin(Node* root);


    Node* remove(int val, Node* root);

    Node* removeMax(Node* root);


    Node* removeMin(Node* root);


    Node* getRoot();


    void print(Node* root, bool isRoot, bool isRight, std::string
space);
    };


    #endif
```

Название файла: AVLtree.cpp

```
    #include "AVLtree.hpp"


    AVLtree::AVLtree(): root(nullptr){}

    AVLtree::AVLtree(std::vector<int> list){
    std::vector<int> sorted_vector = list;
    std::sort(sorted_vector.begin(), sorted_vector.end());
    root = createTree(sorted_vector, 0, sorted_vector.size() - 1);
    }

    int AVLtree::getHeight(Node* root) {
    if (root != nullptr)
         return root->height;
    return 0;
    }


    void AVLtree::updateHeight(Node* root) {
    root->height   =   std::max(getHeight(root->left),   getHeight(root-
>right)) + 1;
    }


    int AVLtree::heightDiff(Node* root) {
    return (getHeight(root->left) - getHeight(root->right));
```

```
}


bool AVLtree::checkIsAVL(Node* root) {
if (abs(getHeight(root->left) - getHeight(root->right)) > 1) {
	return false;
}
if (root->left != nullptr) {
	if (!checkIsAVL(root->left))
		return false;
}
if (root->right != nullptr) {
	if (!checkIsAVL(root->right))
		return false;
}
return true;
}

int AVLtree::minNodeDiff(Node* root) {
int result = 100000000;
if (root->left != nullptr) {
	result = std::min(abs(root->val - root->left->val),
minNodeDiff(root->left));
}
if (root->right != nullptr) {
	if (result > std::min(abs(root->val - root->right->val),
minNodeDiff(root->right)))
		result = std::min(abs(root->val - root->right->val),
minNodeDiff(root->right));
}
return result;
}


Node* AVLtree::createTree(std::vector<int>& list, int begin, int
end) {
if (begin > end)
	return nullptr;
int half = begin + (end - begin) / 2;
Node* new_root = new Node(list[half]);
new_root->left = createTree(list, begin, half - 1);
new_root->right = createTree(list, half + 1, end);
return balance(new_root);
}

Node* AVLtree::balance(Node* root) {
updateHeight(root);
if (heightDiff(root) == 2) {
	if (heightDiff(root->left) < 0) {
		return bigRightRotation(root);
	}
	else if (heightDiff(root->left) > 0) {
		return smallRightRotation(root);
	}
}
else if (heightDiff(root) == -2) {
	if (heightDiff(root->right) > 0) {
		return bigLeftRotation(root);
```

```
      }
      else if (heightDiff(root->right) < 0) {
            return smallLeftRotation(root);
      }
}
return root;
}


Node* AVLtree::insert(int val, Node* root) {
if (root == nullptr)
      return new Node(val);
if (val < root->val) {
      root->left = insert(val, root->left);
}
else if (val > root->val) {
      root->right = insert(val, root->right);
}
else {
      return root;
}
return balance(root);
}


Node* AVLtree::smallLeftRotation(Node* root) {
Node* rootRight = root->right;
root->right = rootRight->left;
rootRight->left = root;

updateHeight(root);
updateHeight(rootRight);

return rootRight;
}


Node* AVLtree::bigLeftRotation(Node* root) {
root->right = smallRightRotation(root->right);
root = smallLeftRotation(root);

return root;
}


Node* AVLtree::smallRightRotation(Node* root) {
Node* rootLeft = root->left;
root->left = rootLeft->right;
rootLeft->right = root;

updateHeight(root);
updateHeight(rootLeft);

return rootLeft;
}


Node* AVLtree::bigRightRotation(Node* root) {
```

```
root->left = smallLeftRotation(root->left);
root = smallRightRotation(root);

return root;
}


void AVLtree::inOrder(Node* root, std::vector<int>& result) {
if (root == nullptr)
      return;
inOrder(root->left, result);
result.push_back(root->val);
inOrder(root->right, result);

}


void AVLtree::preOrder(Node* root, std::vector<int>& result) {
if (root == nullptr)
      return;
result.push_back(root->val);
preOrder(root->left, result);
preOrder(root->right, result);
}


void AVLtree::postOrder(Node* root, std::vector<int>& result) {
if (root == nullptr)
      return;
postOrder(root->left, result);
postOrder(root->right, result);
result.push_back(root->val);
}


Node* AVLtree::searchMin(Node* root) {
while (root->left != nullptr) {
      root = root->left;
}
return root;
}


Node* AVLtree::remove(int val, Node* root) {
if (root == nullptr)
      return nullptr;
if (val < root->val) {
      root->left = remove(val, root->left);
}
else if (val > root->val) {
      root->right = remove(val ,root->right);
}
else {
      Node* left = root->left;
      Node* right = root->right;
      delete root;
      if (right == nullptr)
            return left;
```

```
            Node* minim = searchMin(right);
            minim->right = removeMin(right);
            minim->left = left;
            return balance(minim);
      }
      return balance(root);
      }


      Node* AVLtree::removeMax(Node* root) {
      if (root->right == nullptr)
            return root->left;
      root->right = removeMax(root->right);
      return balance(root);
      }


      Node* AVLtree::removeMin(Node* root) {
      if (root->left == nullptr)
            return root->right;
      root->left = removeMax(root->left);
      return balance(root);
      }


      Node* AVLtree::getRoot() {
      return this->root;
      }


      void AVLtree::print(Node* root, bool isRoot, bool isRight,
std::string space) {
      if (root != nullptr) {
            if (isRoot) {
                  std::cout << "Root - " << root->val << '\n';
            }
            else {
                  std::cout << space << (isRight ? "R - " : "L - ") <<
root->val << '\n';
            }
            print(root->left, false, false, space + (isRight ? "   ": "
"));
            print(root->right, false, true, space + (isRight ? "   ": "
"));
      }
      }
```

Название файла: test.cpp

```
      float measureInsertTime(std::vector<int>& vec1, std::vector<int>&
vec2) {
      AVLtree avl;


      for (int& i : vec1) {
            avl.insert(i, avl.getRoot());
```

```
    }

    auto begin = std::chrono::high_resolution_clock::now();
    for (int& i : vec2) {
        avl.insert(i, avl.getRoot());
    }
    auto end = std::chrono::high_resolution_clock::now();

    auto duration = std::chrono::duration_cast<std::chrono::microseconds>(end - begin);
    return duration.count() / 1000.0;
    }



    int main() {

    srand(time(0));

    int size, insert;
    std::cin >> size >> insert;
    size = std::max(0, size);
    insert = std::max(0, insert);

    std::vector<int> a(size), b(size);
    for (int i = 0; i < size; i++) {
        a[i] = rand() % 1000;
    }
    for (int i = 0; i < insert; i++) {
        b[i] = rand() % 1000;
    }

    std::cout << "Inserting - " << insert << ": Base - " << size << ": Time - " << measureAVL(a, b) << " ms" << std::endl;



    return 0;
    }
```

Название файла: main.cpp

```
#include "AVLtree.hpp"

int main() {


std::vector<int> array = {6, 11 ,4 ,3, 45, 1};

AVLtree tree(array);
tree.print(tree.getRoot(), true, false, "");
std::cout << '\n';



tree.insert(5, tree.getRoot());
tree.insert(10, tree.getRoot());
tree.insert(2, tree.getRoot());
tree.insert(7, tree.getRoot());
tree.print(tree.getRoot(), true, false, "");

return 0;
}
```