**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра информационных систем**

отчет

**по курсовой работе**

**по дисциплине «Программирование»**

Тема: "AVL-Дерево"

|  |  |  |
| --- | --- | --- |
| Студент гр. 0324 |  | Гордиенко Т.Е. |
| Преподаватель |  | Глущенко А.Г |

Санкт-Петербург

2021

**Цель работы**: изучение свойств и организации сбалансированных деревьев; получение практических навыков в работе с АВЛ-деревьями; определение преимуществ и недостатков подобных структур данных; проведение сравнительной характеристики скорости вставки, удаления и поиска элемента в АВЛ-деревьях.

**Постановка задачи**:

Программа должна поддерживать весь описанный функционал:

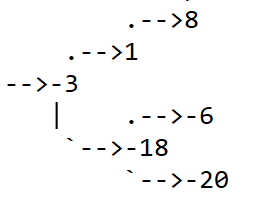
1.   Формирование АВЛ-дерева из N элементов:

a) пользователь вводит количество элементов N АВЛ-дерева, которое автоматически заполняется случайными числами (–99 до 99);

б) пользователь вводит в консоль элементы массива, N определяется автоматически по количеству введенных элементов;

Определение скорости формирования АВЛ-дерева.

2. Вывод в консоль AVL\_tree АВЛ-дерева.



3.   Определение скорости вставки, удаления и получения элемента  дерева.

4.   Определение скорости проверки на сбалансированность.

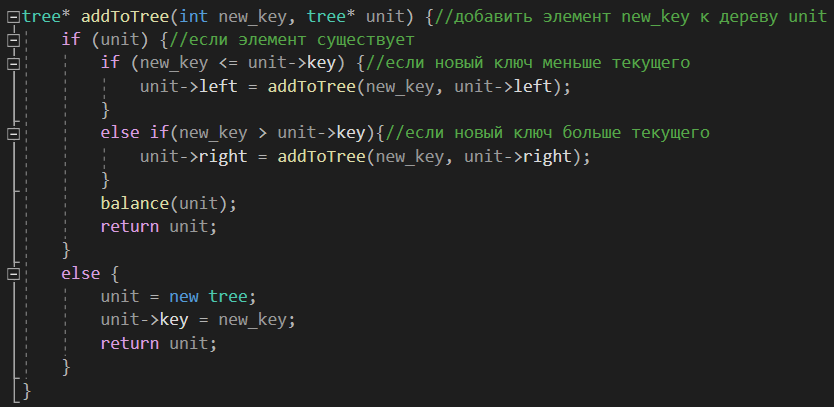
5.   Генерация заданий к практической работе по АВЛ-деревьям. Необходимо сгенерировать задания в файл output\_task в количестве вариантов, которые введет пользователь. В файл output\_key необходимо вывести короткие ответы к заданиям. В файл output\_ans необходимо вывести развернутые ответы к заданиям. Должны быть представлены задания на создание АВЛ-дерева из массива значений, удаление и вставка элементов в АВЛ-дерево.

Выполнение работы:

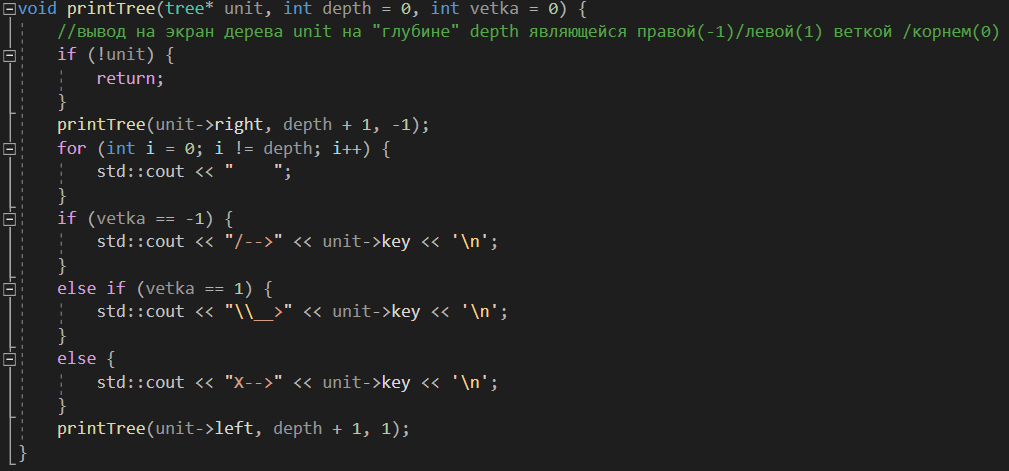
Для комфортного чтения кода, рекомендуется прейти по ссылке, но код так же есть в приложении «В» в конце.

Ссылка на код: <https://github.com/TimPingvin/AVL-Tree/blob/main/Code>

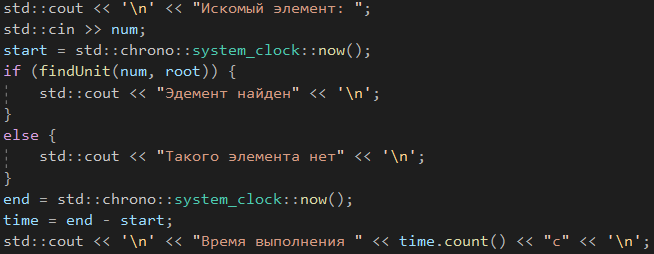
1. Для создания дерева используем функцию добавления к дереву, с условием, если переданное дерево пустое (нулевой указатель), то выделяем память под новый элемент и возвращаем указатель на него.



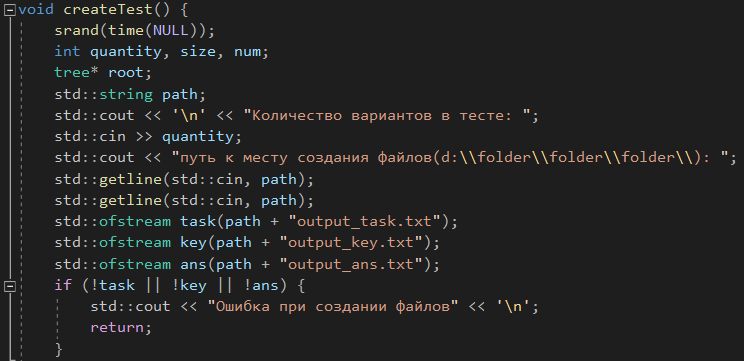
1. Для вывода дерева в консоль будем использовать эту функцию



Для корректного вывода всего дерева, вызываем её только с аргументом unit – корнем дерева. Depth – счётчик глубины для понимания, как далеко находится элемент от корня. Vetka – условный признак, по которому определяется каким потомком является дерево (0 – корень, 1 – меньший потомок, -1 – больший потомок)

1. Для определения скоростей выполнения задач по поиску и удалению используем библиотеку <chrono> 

Для удаления и прочих задач, принцип тот же самый – запоминаем время, до вызова функции и после её завершения, вычисляем разность, выводим.

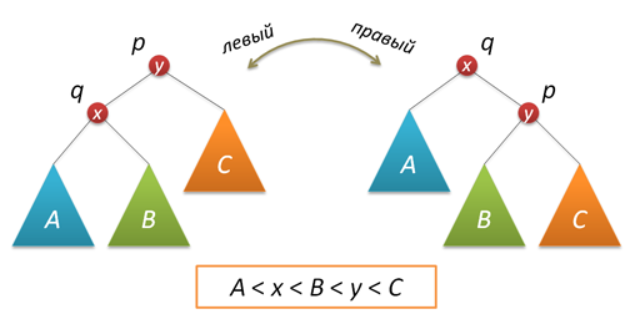
1. Аналогично 3-му пункту
2. Для формирования теста, сначала создаём три файла

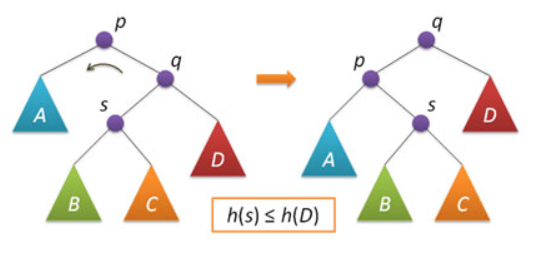
Task – вопросы, key – ответы, ans – промежуточные результаты

При генерации заданий одновременно, одновременно производим и решение этих же заданий. К примеру при формировании задания по созданию дерева, сразу создаём это дерево, и результаты записываем в ans. При завершении задания, конечный ответ записываем в key.

1. Прочая проделанная работа:

При добавлении\удалении элемента, автоматически вызывается “balance()”. Это функция по балансировке дерева, в случае необходимости. Таким образом, при всём желании пользователя, он не получит несбалансированное дерево. Теория при балансировке:





Приложение В

|  |
| --- |
| #include <iostream> |
|  | #include <string> |
|  | #include <time.h> |
|  | #include <fstream> |
|  | #include <chrono> |
|  |  |
|  | int max(int num1, int num2) { |
|  | if (num1 > num2) { |
|  | return num1; |
|  | } |
|  | else { |
|  | return num2; |
|  | } |
|  | } |
|  |  |
|  | struct tree { |
|  | int key; |
|  | int height = 1; |
|  | tree\* left = NULL; |
|  | tree\* right = NULL; |
|  | }; |
|  |  |
|  | int heightTree(tree\* unit) {//величина дерева |
|  | if (!unit) { |
|  | return 0; |
|  | } |
|  | else { |
|  | return unit->height; |
|  | } |
|  | } |
|  |  |
|  | void streamTree(std::ofstream \*file, tree\* unit, int depth = 0, int vetka = 0) { |
|  | if (!unit) { |
|  | return; |
|  | } |
|  | streamTree(file, unit->right, depth + 1, -1); |
|  | for (int i = 0; i != depth; i++) { |
|  | \*file << " "; |
|  | } |
|  | if (vetka == -1) { |
|  | \*file << "/-->" << unit->key << '\n'; |
|  | } |
|  | else if (vetka == 1) { |
|  | \*file << "\\\_\_>" << unit->key << '\n'; |
|  | } |
|  | else { |
|  | \*file << "X-->" << unit->key << '\n'; |
|  | } |
|  | streamTree(file, unit->left, depth + 1, 1); |
|  | } |
|  |  |
|  | void printTree(tree\* unit, int depth = 0, int vetka = 0) { |
|  | //вывод на экран дерева unit на "глубине" depth являющейся правой(-1)/левой(1) веткой /корнем(0) |
|  | if (!unit) { |
|  | return; |
|  | } |
|  | printTree(unit->right, depth + 1, -1); |
|  | for (int i = 0; i != depth; i++) { |
|  | std::cout << " "; |
|  | } |
|  | if (vetka == -1) { |
|  | std::cout << "/-->" << unit->key << '\n'; |
|  | } |
|  | else if (vetka == 1) { |
|  | std::cout << "\\\_\_>" << unit->key << '\n'; |
|  | } |
|  | else { |
|  | std::cout << "X-->" << unit->key << '\n'; |
|  | } |
|  | printTree(unit->left, depth + 1, 1); |
|  | } |
|  |  |
|  | void leftTurn(tree\* unit) {//левое вращение (если для текущего дерева величина ПРАВОГО поддерева больше величины левого на 2 |
|  | tree\* temp\_tree; |
|  | int temp\_num; |
|  | if (heightTree(unit->right->right) >= heightTree(unit->right->left)) {//если правое дерево правого потомка больше либо равно левому |
|  | //мылое левое вращение |
|  | temp\_tree = unit->right->right;//запоминаем адресс большего поддерева правого потомка |
|  | temp\_num = unit->key;//запоминаем ключ старого корня |
|  | unit->key = unit->right->key;//в корень помещаем ключь правого поддерева |
|  | unit->right->key = temp\_num;//в ключь правого поддерева помещаем ключ корня |
|  | unit->right->right = unit->right->left;//меняем указатель правой ветки правого поддерева на левую ветку |
|  | unit->right->left = unit->left;//левая ветка правого поддерева указывает на левое поддерево |
|  | unit->left = unit->right;//теперь левая ветка корня указывает на старый корень |
|  | unit->right = temp\_tree; |
|  | unit->left->height = max(heightTree(unit->left->left), heightTree(unit->left->right)) + 1; |
|  | unit->height = max(heightTree(unit->left), heightTree(unit->right)) + 1; |
|  | } |
|  | else { |
|  | //большое левое вращение |
|  | temp\_num = unit->key; |
|  | unit->key = unit->right->left->key; |
|  | unit->right->left->key = temp\_num;//меняем ключи корня и левого поддерева правого потомка местами |
|  | temp\_tree = unit->right->left;//запоминаем адресс левого поддерева правого потомка (теперь здесь ключ старого корня) |
|  | unit->right->left = unit->right->left->right;//меняем левоего поддерево правого потомка на правое поддерево его левого потомка |
|  | temp\_tree->right = temp\_tree->left;//корректируем старый корень |
|  | temp\_tree->left = unit->left; |
|  | unit->left = temp\_tree;//левая ветка нового корня теперь указывает на старый корень |
|  | unit->left->height = max(heightTree(unit->left->left), heightTree(unit->left->right)) + 1;//корректировка высот |
|  | unit->right->height = max(heightTree(unit->right->left), heightTree(unit->right->right)) + 1; |
|  | unit->height = max(heightTree(unit->left), heightTree(unit->right)) + 1; |
|  | } |
|  | } |
|  |  |
|  | void rightTurn(tree\* unit) {//правое вращение (если для текущего дерева величина ЛЕВОГО поддерева больше величины правого на 2 |
|  | tree\* temp\_tree; |
|  | int temp\_num; |
|  | if (heightTree(unit->left->left) >= heightTree(unit->left->right)) {//если левое поддерево левого потомка выше или равно правому поддереву |
|  | //малый правый поворот |
|  | temp\_num = unit->key; |
|  | unit->key = unit->left->key; |
|  | unit->left->key = temp\_num; |
|  | temp\_tree = unit->left->left; |
|  | unit->left->left = unit->left->right; |
|  | unit->left->right = unit->right; |
|  | unit->right = unit->left; |
|  | unit->left = temp\_tree; |
|  | unit->right->height = max(heightTree(unit->right->left), heightTree(unit->right->right)) + 1; |
|  | unit->height = max(heightTree(unit->left), heightTree(unit->right)) + 1; |
|  | } |
|  | else {//большой правый поворот |
|  | temp\_num = unit->key; |
|  | unit->key = unit->left->right->key; |
|  | unit->left->right->key = temp\_num; |
|  | temp\_tree = unit->left->right; |
|  | unit->left->right = unit->left->right->left; |
|  | temp\_tree->left = temp\_tree->right; |
|  | temp\_tree->right = unit->right; |
|  | unit->right = temp\_tree; |
|  | unit->left->height = max(heightTree(unit->left->left), heightTree(unit->left->right)) + 1;//корректировка высот |
|  | unit->right->height = max(heightTree(unit->right->left), heightTree(unit->right->right)) + 1; |
|  | unit->height = max(heightTree(unit->left), heightTree(unit->right)) + 1; |
|  | } |
|  | } |
|  |  |
|  | void balance(tree\* unit) {//сбалансировать дерево unit |
|  | unit->height = max(heightTree(unit->left), heightTree(unit->right)) + 1;//корректировка высоты |
|  | if (heightTree(unit->left) - heightTree(unit->right) < -1) {//если коэф баланса меньше -1 (правое дерево сильно выше левого) |
|  | leftTurn(unit); |
|  | balance(unit); |
|  | } |
|  | else if (heightTree(unit->left) - heightTree(unit->right) > 1){//если коэф баланса больше 1 (левое сильно больше правого) |
|  | rightTurn(unit); |
|  | balance(unit); |
|  | } |
|  | } |
|  |  |
|  | tree\* addToTree(int new\_key, tree\* unit) {//добавить элемент new\_key к дереву unit |
|  | if (unit) {//если элемент существует |
|  | if (new\_key <= unit->key) {//если новый ключ меньше текущего |
|  | unit->left = addToTree(new\_key, unit->left); |
|  | } |
|  | else if(new\_key > unit->key){//если новый ключ больше текущего |
|  | unit->right = addToTree(new\_key, unit->right); |
|  | } |
|  | balance(unit); |
|  | return unit; |
|  | } |
|  | else { |
|  | unit = new tree; |
|  | unit->key = new\_key; |
|  | return unit; |
|  | } |
|  | } |
|  |  |
|  | tree\* removeFromTree(int num, tree\* unit) {//удалить из дерева unit элемент с ключом num |
|  | tree\* temp\_tree, \*temp\_tree2; |
|  | if (!unit) {//если дошли до нулевого указателя |
|  | return unit; |
|  | } |
|  | else if (num < unit->key) {//если удаляемое число меньше, чем ключ текущего дерева |
|  | unit->left = removeFromTree(num, unit->left);//вызываем удаления для левого поддерева и присваиваем результат удаления |
|  | balance(unit);//проверяем на баланс и корректируем, если необходимо |
|  | return unit;//возвращаем указатель на это дерево |
|  | } |
|  | else if (num > unit->key) {//если удаляемое число больше, чем ключ текущего дерева |
|  | unit->right = removeFromTree(num, unit->right);//вызываем удаление для правого поддерева и присваиваем результат удаления |
|  | balance(unit);//проверяем на баланс и корректируем, если необходимо |
|  | return unit;//возвращаем указатель на это дерево |
|  | } |
|  | else {//если ключ совпадает с удаляемым значением |
|  | if (unit->height == 1) {//если дерево не имеет потомков |
|  | delete unit;//удаляем дерево; |
|  | return NULL;//возвращаем нулевой указатель |
|  | } |
|  | else if(unit->left && unit->right){//если есть оба потомка |
|  | if (!unit->right->left) {//если правое поддерево не имеет левых потомков (является наименьшим) |
|  | //меняем ключи местами с удалением элемента и перебалансировкой |
|  | temp\_tree = unit->right; |
|  | unit->key = unit->right->key; |
|  | unit->right = unit->right->right; |
|  | delete temp\_tree; |
|  | balance(unit); |
|  | } |
|  | else {//если у правого поддерева есть меньший потомок |
|  | //меняем ключи удалаяемого элемента с меньшим потомком, и удаляем ненужный ключь |
|  | temp\_tree = unit->right; |
|  | int count = 0; |
|  | while (temp\_tree->left->left) { |
|  | temp\_tree = temp\_tree->left; |
|  | count++; |
|  | } |
|  | unit->key = temp\_tree->left->key; |
|  | temp\_tree2 = temp\_tree->left; |
|  | temp\_tree->left = temp\_tree->left->right; |
|  | delete temp\_tree2; |
|  | //балансируем деревья, которые были между правым потомком элемента и наименьшим |
|  | while (count > -1) { |
|  | temp\_tree = unit->right; |
|  | for (int i = 0; i != count; i++) { |
|  | temp\_tree = temp\_tree->left; |
|  | } |
|  | balance(temp\_tree); |
|  | count--; |
|  | } |
|  | balance(unit); |
|  | } |
|  | return unit; |
|  | } |
|  | else {//если есть только один потомок |
|  | if (unit->left) {//если есть левый |
|  | temp\_tree = unit->left; |
|  | delete unit;//удаляем текущее дерево |
|  | return temp\_tree;//возвращаем указатель на левое поддерево |
|  | } |
|  | else {//если есть правый |
|  | temp\_tree = unit->right; |
|  | delete unit;//удаляем текущее дерево |
|  | return temp\_tree;//возвращаем указатель на правое поддерево |
|  | } |
|  | } |
|  | } |
|  | } |
|  |  |
|  | tree\* removeTree(tree\* root) { |
|  | if (!root) { |
|  | return NULL; |
|  | } |
|  | else { |
|  | root->left = removeTree(root->left); |
|  | root->right = removeTree(root->right); |
|  | delete root; |
|  | return NULL; |
|  | } |
|  | } |
|  |  |
|  | tree\* findUnit(int num, tree\* unit) {//поиск элемента с ключом num |
|  | if (!unit) {//если поступил нулевой указатель |
|  | return NULL; |
|  | } |
|  | else { |
|  | if (unit->key == num) {//если ключи совпадают |
|  | return unit; |
|  | } |
|  | else { |
|  | if (num > unit->key) {//если искомое больше ключа |
|  | return findUnit(num, unit->right); |
|  | } |
|  | else {//если искомое меньше ключа |
|  | return findUnit(num, unit->left); |
|  | } |
|  | } |
|  | } |
|  | } |
|  |  |
|  | tree\* createRandTree(int size) {//дерево из size рандомных чисел |
|  | tree\* root = new tree; |
|  | srand(time(NULL)); |
|  | root->key = rand() % 100 - rand() % 100; |
|  | for (int i = 1; i != size; i++) { |
|  | addToTree(rand() % 100 - rand() % 100, root); |
|  | } |
|  | return root; |
|  | } |
|  |  |
|  | tree\* inputTree() {//дерево из введённых пользователем эелементов |
|  | tree\* root = NULL; |
|  | int num; |
|  | std::cout << "элементы дерева (через пробел, после последнего поставьте точку): "; |
|  | std::cin >> num; |
|  | while (!std::cin.fail()) { |
|  | root = addToTree(num, root); |
|  | std::cin >> num; |
|  | } |
|  | std::cin.ignore(1024); |
|  | std::cin.clear(); |
|  | return root; |
|  | } |
|  |  |
|  | tree\* readTreeFromFile() {//чтение элементов дерева из файла |
|  | tree\* root = NULL; |
|  | std::string path; |
|  | int num; |
|  | std::cout << "Путь до файла(d:\\1\\файл.txt): "; |
|  | getline(std::cin, path); |
|  | std::ifstream file(path); |
|  | if (!file) { |
|  | std::cout << "Ошибка чтения файла" << '\n'; |
|  | return root; |
|  | } |
|  | else { |
|  | file >> num; |
|  | while (!file.eof() && !file.fail()) { |
|  | addToTree(num, root); |
|  | file >> num; |
|  | } |
|  | file.close(); |
|  | return root; |
|  | } |
|  | } |
|  |  |
|  | bool areBalance(tree\* root) {//Проверка на сбалансированность (без корректировки) |
|  | if (!root) { |
|  | return true; |
|  | } |
|  | else { |
|  | if (areBalance(root->left) && areBalance(root->right)) { |
|  | if (-2 > heightTree(root->left) - heightTree(root->right) < 2) { |
|  | return true; |
|  | } |
|  | else { |
|  | return false; |
|  | } |
|  | } |
|  | else { |
|  | return false; |
|  | } |
|  | } |
|  | } |
|  |  |
|  | void createTest() { |
|  | srand(time(NULL)); |
|  | int quantity, size, num; |
|  | tree\* root; |
|  | std::string path; |
|  | std::cout << '\n' << "Количество вариантов в тесте: "; |
|  | std::cin >> quantity; |
|  | std::cout << "путь к месту создания файлов(d:\\folder\\folder\\folder\\): "; |
|  | std::getline(std::cin, path); |
|  | std::getline(std::cin, path); |
|  | std::ofstream task(path + "output\_task.txt"); |
|  | std::ofstream key(path + "output\_key.txt"); |
|  | std::ofstream ans(path + "output\_ans.txt"); |
|  | if (!task || !key || !ans) { |
|  | std::cout << "Ошибка при создании файлов" << '\n'; |
|  | return; |
|  | } |
|  | for (int i = 0; i != quantity; i++) { |
|  | root = NULL; |
|  | task << "Вариант " << i + 1 << '\n'; |
|  | task << "Создать AVLдерево из: "; |
|  | ans << "Вариант " << i + 1 << '\n'; |
|  | ans << "Создание дерева" << '\n'; |
|  | ans << "=====================================" << '\n'; |
|  | key << "Вариант " << i + 1 << '\n'; |
|  | size = rand() % 8 + 7; |
|  | for (int j = 0; j != size; j++) { |
|  | num = rand() % 10; |
|  | task << num << " "; |
|  | root = addToTree(num, root); |
|  | streamTree(&ans, root); |
|  | ans << "=====================================" << '\n'; |
|  | } |
|  | key << "Созданное дерево" << '\n'; |
|  | streamTree(&key, root); |
|  | num = rand() % 10; |
|  | task << "Добавить элемент " << num << '\n'; |
|  | root = addToTree(num, root); |
|  | ans << "Добавляем элемент " << num << '\n'; |
|  | streamTree(&ans, root); |
|  | ans << "=====================================" << '\n'; |
|  | key << "Дерево после добавления элемента " << num << '\n'; |
|  | streamTree(&key, root); |
|  | num = rand() % 10; |
|  | task << "Удалить элемент " << num << '\n' << '\n'; |
|  | if (findUnit(num, root)) { |
|  | ans << "Элемент найден, удаляем его и производим корректировку, если необходимо" << '\n'; |
|  | root = removeFromTree(num, root); |
|  | streamTree(&ans, root); |
|  | } |
|  | else { |
|  | ans << "Элемента нет, так что бьём баклуши" << '\n'; |
|  | } |
|  | key << "Дерево после удаления элемента " << num << '\n'; |
|  | streamTree(&key, root); |
|  | removeTree(root); |
|  | } |
|  | task.close(); |
|  | ans.close(); |
|  | key.close(); |
|  | } |
|  |  |
|  | int main() { |
|  | setlocale(NULL, ""); |
|  | tree\* root = NULL; |
|  | int choise = 1; |
|  | int num; |
|  | std::string trash; |
|  | std::chrono::system\_clock::time\_point start, end; |
|  | std::chrono::duration <double> time; |
|  | while (1) { |
|  | std::cout << '\n' << "Выберите пункт:" << '\n'; |
|  | std::cout << "1) Создать дерево" << '\n'; |
|  | std::cout << "2) Вывод дерева" << '\n'; |
|  | std::cout << "3) Удаление/получение эелемента дерева" << '\n'; |
|  | std::cout << "4) Проверка на сбалансированность" << '\n'; |
|  | std::cout << "5) Генерация теста" << '\n'; |
|  | std::cout << "0) Выход" << '\n'; |
|  | std::cin >> choise; |
|  | std::cout << "Был выбран" << choise << '\n'; |
|  | switch (choise) { |
|  | case 1://создание дерева |
|  | while (choise) { |
|  | if (root) { |
|  | root = removeTree(root); |
|  | } |
|  | std::cout << '\n' << "1) Создать из n рандомных чисел" << '\n'; |
|  | std::cout << "2) Создать из введённых чисел" << '\n'; |
|  | std::cout << "3) Создать из чисел из файла" << '\n'; |
|  | std::cin >> choise; |
|  | switch (choise) { |
|  | case 1: |
|  | std::cout << '\n' << "Количество элементов дерева: "; |
|  | std::cin >> num; |
|  | start = std::chrono::system\_clock::now(); |
|  | root = createRandTree(num); |
|  | end = std::chrono::system\_clock::now(); |
|  | time = end - start; |
|  | std::cout << '\n' << "Время выполнения: " << time.count() << "c" << '\n'; |
|  | choise = 0; |
|  | break; |
|  | case 2: |
|  | start = std::chrono::system\_clock::now(); |
|  | root = inputTree(); |
|  | end = std::chrono::system\_clock::now(); |
|  | time = end - start; |
|  | std::getline(std::cin, trash); |
|  | std::cout << '\n' << "Время выполнения: " << time.count() << "c" << '\n'; |
|  | choise = 0; |
|  | break; |
|  | case 3: |
|  | start = std::chrono::system\_clock::now(); |
|  | root = readTreeFromFile(); |
|  | end = std::chrono::system\_clock::now(); |
|  | time = end - start; |
|  | std::cout << '\n' << "Время выполнения: " << time.count() << "c" << '\n'; |
|  | choise = 0; |
|  | break; |
|  | default: |
|  | std::cout << "Попробуй ещё раз" << '\n'; |
|  | choise = 1; |
|  | break; |
|  | } |
|  | } |
|  | choise = 1; |
|  | break; |
|  | case 2://вывод дерева в консоль |
|  | printTree(root); |
|  | break; |
|  | case 3://поиск/удаление |
|  | while (choise) { |
|  | std::cout << '\n' << "1) Поиск элемента" << '\n'; |
|  | std::cout << "2) Удаление элемента" << '\n'; |
|  | std::cin >> choise; |
|  | switch (choise) { |
|  | case 1: |
|  | std::cout << '\n' << "Искомый элемент: "; |
|  | std::cin >> num; |
|  | start = std::chrono::system\_clock::now(); |
|  | if (findUnit(num, root)) { |
|  | std::cout << "Эдемент найден" << '\n'; |
|  | } |
|  | else { |
|  | std::cout << "Такого элемента нет" << '\n'; |
|  | } |
|  | end = std::chrono::system\_clock::now(); |
|  | time = end - start; |
|  | std::cout << '\n' << "Время выполнения " << time.count() << "c" << '\n'; |
|  | choise = 0; |
|  | break; |
|  | case 2: |
|  | std::cout << '\n' << "Удаляемый элемент: "; |
|  | std::cin >> num; |
|  | start = std::chrono::system\_clock::now(); |
|  | root = removeFromTree(num, root); |
|  | end = std::chrono::system\_clock::now(); |
|  | time = end - start; |
|  | std::cout << '\n' << "Время выполнения " << time.count() << "c" << '\n'; |
|  | choise = 0; |
|  | break; |
|  | default: |
|  | std::cout << "try again" << '\n'; |
|  | choise = 3; |
|  | break; |
|  | } |
|  | } |
|  | choise = 3; |
|  | break; |
|  | case 4://проверка на сбалансированность |
|  | start = std::chrono::system\_clock::now(); |
|  | if (areBalance(root)) { |
|  | std::cout << '\n' << "Дерево сбалансированно" << '\n'; |
|  | } |
|  | else { |
|  | std::cout << '\n' << "Дерево НЕ сбалансированно" << '\n'; |
|  | } |
|  | end = std::chrono::system\_clock::now(); |
|  | time = end - start; |
|  | std::cout << '\n' << "Время выполнения " << time.count() << "c" << '\n'; |
|  | break; |
|  | case 5://формирование теста |
|  | createTest(); |
|  | break; |
|  | case 0: |
|  | return 0; |
|  | break; |
|  | default: |
|  | std::cout << '\n' << "Попробуй ещё раз"; |
|  | break; |
|  | } |
|  | } |
|  | return 1; |
|  | } |