­­Бюджетное учреждение высшего образования   
Ханты-Мансийского автономного округа   
«Сургутский государственный университет»

Политехнический институт

Кафедра автоматики и компьютерных систем

**Отчет**

по лабораторной работе № 5

по дисциплине «Алгоритмы и структуры данных»

Выполнил: студент группы 609-21,

Шумилов И.Д.

Принял: старший преподаватель кафедры АиКС

Назаров Е.В.

2024 г.

**Цель работы:** изучить базовые алгоритмы работы с деревьями: построение, обход, поиска элемента, удаление элемента, подсчет количества узлов, нахождение высоты дерева, исследовать свойства деревьев, закрепить навыки структурного программирования.

**Общее задание на работу:**

1. Реализовать функции вставки, поиска и удаления узла, обхода дерева, вставки в корень, вывода дерева на экран, нахождения высоты дерева и количества узлов, а также функцию вставки, строящую рандомизированное дерево.

2. Построить зависимости высоты деревьев (обычного бинарного поиска – БДП и рандомизированного – РБДП) от количества ключей (ключи – случайные и упорядоченные величины), полагая, что ключи целые числа.

3. Реализовать заданную функцию (в соответствии с вариантом).

4. Составить отчет, в котором привести графики полученных зависимостей, анализ свойств алгоритмов, листинги функций вставки, поиска, удаления узла с комментариями и выводы по работе.

**Код программы**

Листинг 1. Класс узла.

class Node {

public:

Node() : key(0), size(1), left(nullptr), right(nullptr), parent(nullptr) {};

Node(int key) : size(1), left(nullptr), right(nullptr), parent(nullptr) { this->key = key; };

int key;

int size;

Node\* left;

Node\* right;

Node\* parent;

};

Листинг 2. Класс дерева.

class Tree {

public:

Node\* root;

Tree() : root(nullptr) {};

Node\* insert(int key);

Node\* insert(Node\* root, int key);

Node\* insert(Node\* root, Node\* parent, int key);

Node\* insertAtRoot(Node\* node, int key);

Node\* randomizedInsert(Node\* node, int key);

Node\* find(Node\* node, int key);

Node\* getInorderSucc(Node\* node);

Node\* remove(Node\* node, int key);

void preorderTraversal(Node\* node, void func(Node\*));

void postorderTraversal(Node\* node, void func(Node\*));

void print();

int getHeight(Node\* node);

int countNodes(Node\* node);

int individual(Node\* node);

int getSize(Node\* node);

void fixSize(Node\* node);

Node\* rotateRight(Node\* node);

Node\* rotateLeft(Node\* node);

static Tree genTree(int size, bool random);

static Tree genRandomizedTree(int size, bool random);

};

Листинг 3. Функция обычной вставки узла в дерево.

Node\* Tree::insert(Node\* node, Node\* parent, int key) {

if (!node) {

Node\* newNode = new Node(key);

newNode->parent = parent;

if (!root)

root = newNode;

return newNode;

}

if (key < node->key) {

node->left = insert(node->left, node, key);

} else if (key > node->key) {

node->right = insert(node->right, node, key);

}

return node;

}

Листинг 4. Функция вставки узла в корень дерева.

Node\* Tree::insertAtRoot(Node\* node, int key) {

if (!node) {

Node\* newNode = new Node(key);

if (!root)

root = newNode;

return newNode;

}

if (key < node->key) {

node->left = insertAtRoot(node->left, key);

return rotateRight(node);

} else {

node->right = insertAtRoot(node->right, key);

return rotateLeft(node);

}

}

Листинг 5. Функция рандомизированной вставки в дерево.

Node\* Tree::randomizedInsert(Node\* node, int key) {

if (!node) {

Node\* newNode = new Node(key);

if (!root)

root = newNode;

return newNode;

}

if (rand() % (node->size + 1) == 0)

return insertAtRoot(node, key);

if (key < node->key)

node->left = randomizedInsert(node->left, key);

else

node->right = randomizedInsert(node->right, key);

fixSize(node);

return node;

}

Листинг 6. Функция поиска узла.

Node\* Tree::find(Node\* node, int key) {

if (!node) return nullptr;

else if (node->key == key) return node;

else if (key < node->key) return find(node->left, key);

else return find(node->right, key);

}

Листинг 7. Функция удаления узла.

Node\* Tree::remove(Node\* node, int key) {

// Выход, если дерево пустое или узел не существует

if (!root || !node) return nullptr;

// Поиск удаляемого узла

if (key < node->key) {

node->left = remove(node->left, key);

return node;

} else if (key > node->key) {

node->right = remove(node->right, key);

return node;

}

// Последующий код выполняется только если ключ удаляемого узла равен переданного узла

// В случае если один потомок он удаляется и устанавливаются связи

if (!node->left) {

Node\* tmp = node->right;

delete node;

return tmp;

} else if (!node->right) {

Node\* tmp = node->left;

delete node;

return tmp;

}

// В случае если два потомка, то проводятся сложные махинации

Node\* succ = getInorderSucc(node->right);

node->key = succ->key;

if (succ->parent->left == succ)

succ->parent->left = succ->right;

else

succ->parent->right = succ->right;

delete succ;

return node;

}

Листинг 8. Функция вывода дерева на экран.

void Tree::print() {

preorderTraversal(root, [](Node\* node) {

int lk = 0, rk = 0, pk = 0;

if (node->left) lk = node->left->key;

if (node->right) rk = node->right->key;

if (node->parent) pk = node->parent->key;

if (node->parent == nullptr) std::cout << "=============root=============" << std::endl;

else std::cout << "-------------node-------------" << std::endl;

std::cout << "this\t" << "left\t" << "right\t" << "parent\t" << std::endl;

std::cout << node->key << "\t" << lk << "\t" << rk << "\t" << pk << "\t\n" << std::endl;

});

}

Листинг 9. Функция нахождения высоты дерева.

int Tree::getHeight(Node\* node) {

if (!node) return 0;

int leftHeight = getHeight(node->left);

int rightHeight = getHeight(node->right);

return 1 + std::max(leftHeight, rightHeight);

}

Листинг 8. Функция нахождения количества узлов.

int Tree::countNodes(Node\* node) {

if (!node)

return 0;

else

return 1 + countNodes(node->left) + countNodes(node->right);

}

**Экспериментальные данные:**

Таблица 2 – Зависимость высот бинарных и рандомизированных деревьев от количества узлов и от порядка их добавления

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Количество узлов | Высота дерева | | | |
| Случайные ключи | | Упорядоченные ключи | |
| БДП | РБДП | БДП | РБДП |
| 1000 | 24 | 22 | 1000 | 20 |
| 2000 | 26 | 24 | 2000 | 23 |
| 3000 | 27 | 24 | 3000 | 23 |
| 4000 | 29 | 27 | 4000 | 25 |
| 5000 | 29 | 25 | 5000 | 26 |
| 6000 | 30 | 26 | 6000 | 25 |
| 7000 | 30 | 26 | 7000 | 25 |
| 8000 | 30 | 28 | 8000 | 28 |
| 9000 | 31 | 27 | 9000 | 29 |
| 10000 | 31 | 27 | 10000 | 30 |

Рис. 1 – График зависимости высоты дерева от количества узлов при случайной генерации ключей

Проанализировав график зависимости, можно прийти к следующим выводам:

1. Рандомизированные деревья получаются вполне сбалансированными, и даже более сбалансированными, чем обычные деревья поиска со случайными значениями.
2. В случае вставки упорядоченных значений в БДП его высота становится прямо пропорциональной количеству узлов, что означает линейную зависимость для вставки, поиска и удаления, в сравнение логарифмической зависимости для рандомизированных деревьев.

Рандомизированные деревья крайне простые в реализации, но они имеют и свои минусы. Например, для каждого узла приходится хранить еще одно дополнительное целочисленное значение, в котором хранится размер поддерева. Пусть время работы и логарифмическое, но константа пропорциональности вероятнее всего немаленькая, и сверх этого в нем используются случайные числа что не гарантирует того, что дерево окажется сбалансированным, однако уже при размере в 10000 вероятность этого настолько мала, что столкнуться с этим при работе практически невозможно.

**Анализ свойств алгоритмов**

Таблица 3 – Анализ свойств алгоритмов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Алгоритм | Бинарное дерево поиска | | Рандомизированное дерево | |
| Среднее | Худшее | Среднее | Худшее |
| Вставка | O(log n) | O(n) | O(log n) | O(log n) |
| Поиск | O(log n) | O(n) | O(log n) | O(log n) |
| Удаление | O(log n) | O(n) | O(log n) | O(log n) |

Можно заметить, что алгоритмы рандомизированного дерева и в среднем, и в худшем случаях имеют логарифмическую зависимость, а обычное дерево поиска в худшем случае имеет линейную зависимость. Это обуславливается тем, что если в БДП вставлять ключи в упорядоченном порядке, то получится обычный список.

**Индивидуальное задание на лабораторную работу**

Таблица 1 – Описание варианта индивидуального задания

|  |  |  |  |
| --- | --- | --- | --- |
| № | Тип | Диапазон значений | Функция |
| 7 | int | [1000; 2000] | Подсчет суммы четных чисел во внутренних узлах |

Листинг 10. Функция, реализующая индивидуальное задание.

int Tree::individual(Node\* node) {

if (!node)

return 0;

else if (node->key % 2 == 0 && (node->left || node->right))

return node->key + individual(node->left) + individual(node->right);

else

return individual(node->left) + individual(node->right);

}

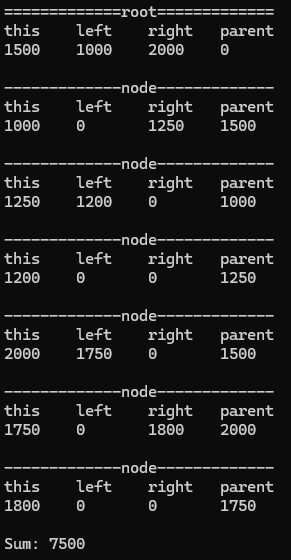


Рис. 2 – Результат выполнения функции индивидуального задания

Проверим: внешними узлами являются все, кроме узлов с ключами 1200 и 1800, так как у них нет ни одного потомка. Таким образом результат должен быть равен 1500 + 1000 + 1250 + 2000 + 1750 = 7500. Функция работает корректно.

**Вывод:** были изучены и реализованы базовые алгоритмы работы с деревьями: построение, обход, поиска элемента, удаление элемента, подсчет количества узлов, нахождение высоты дерева, исследовать свойства деревьев, также были закреплены навыки структурного программирования.