БУ ВО Ханты-Мансийского автономного округа – Югры
«Сургутский государственный университет»

Политехнический институт
Кафедра автоматики и компьютерных систем

ОТЧЁТ

по лабораторной работе №5

по дисциплине: «Алгоритмы и структуры данных»

Выполнил: студент группы №609-22,
Бельтюков Михаил Олегович
Принял: старший преподаватель кафедры АиКС
Назаров Евгений Владимирович

Сургут
2024г.

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.

Задание

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 и рандомизированного) от количества ключей (ключи – случайные и упорядоченные величины), полагая, что ключи целые числа.

3. Реализовать заданную функцию (в соответствии с вариантом: T – тип ключей, D – диапазон изменения значений ключей).

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

Таблица 1 — индивидуальный вариант

№	T	D	Функция
2	char	[а..я, А..Я]	Подсчет количества гласных в листьях

Ход работы

Для организации двоичного дерева поиска были написаны 2 класса - «узел» и «дерево».

```
class IntNode{
   public:
      int key;
      IntNode *left;
      IntNode *right;
      IntNode *parent;
```

```
        bool visited;
        int height;
        int size;
        void out();
        int get_size() {
            int _size = 1;
            if (left!= NULL)
                _size += left->size;
            if (right!= NULL)
                _size += right->size;
            return _size;
        }
        void fix_size() {
            size = 1;
            if (left!= NULL)
                size += left->size;
            if (right!= NULL)
                size += right->size;

        };
        IntNode(int key){
            left = NULL;
            right = NULL;
            parent = NULL;
            visited = 0;
            this->key = key;
            height = -1;
            size = 1;
        }
};
```

Листинг 1 — класс узел

```
class IntTree {
    private:
        int count;
    public:
        IntNode *root;
```

```
        // bin tree search
        static IntTree rand_tree_gen(int size);
        static IntTree sorted_tree_gen(int size);
        //random tree
        static IntTree rand_rand_tree_gen(int size);
        static IntTree rand_sorted_tree_gen(int size);

        void dfs(IntNode * a);
        void dfs_support(IntNode * a); // NULL visited nodes
        void out();
        void insert(IntNode * rt, int key);
        IntNode * find(IntNode * rt, int key);
        int remove(IntNode*, int key);
        IntNode * find_min(IntNode * node);

        IntTree() : root(NULL), count(0) {};
        IntTree(IntNode * rt) : root(rt), count(1) {};
        void find_height(IntNode * a, int * mx);
            // рандомизированное
        IntNode* rotate_right(IntNode* p);
        IntNode* rotate_left(IntNode* p);
        IntNode* insert_root(IntNode* p, int k);
        IntNode* insert_random(IntNode* p, int k);
};
```

Листинг 2 — класс дерево

```
IntNode * IntTree::find(IntNode * node, int key) {
   if (node == NULL)   return NULL;
   if (node->key == key)   return node;
   else if (node->key < key)   find(node->right, key);
   else    find(node->left, key);
   return NULL;
}
```

Листинг 3 — функция поиска узла по ключу

Для реализации функции удаления узла необходима функция поиска минимального элемента (в случае, когда у удаляемого узла 2 потомка).

```
IntNode * IntTree::find_min(IntNode * node) {
    if (node) {
        IntNode * temp = node;
        while (temp->left) {
            temp = temp->left;
        }
        return temp;
    }
    return NULL;
}
```

Листинг 4 — функция поиска минимального элемента в поддереве с корнем node

```
int IntTree::remove(IntNode * node, int key) {
    IntNode * removed = find(node, key);
    if (removed == NULL) return 0;
    char _key = removed->key;
    IntNode * l;
    IntNode * r;
    if (removed->left && removed->right) {

        IntNode * min = this->find_min(removed->right);
        if (removed->parent == NULL) {

            min->left = removed->left;
            if (min != removed->right) min->right = removed->right;
            removed->right->parent = min;
            removed->left->parent = min;
            min->parent = removed->parent;
            root = min;
        }
        else {
            removed->parent->right = min;
            min->left = removed->left;
            if (min != removed->right) min->right = removed->right;
```

```
            removed->left->parent = min;
            removed->right->parent = min;
            min->parent = removed->parent;
        }
        removed->left->fix_size();
        removed->right->fix_size();
        min->fix_size();


    }
    else if (removed->left) {
        removed->left->parent = removed->parent;
        if (removed->parent) {
            if (removed->parent->right == removed) removed->parent->right = removed->left;
            else removed->parent->left = removed->left;
        }
        else {
            root = removed->left;
            root->parent = NULL;
        }
        removed->left->fix_size();
    }
    else if (removed->right){
        removed->right->parent = removed->parent;
        if (removed->parent) {
            if (removed->parent->right == removed) removed->parent->right = removed->right;
            else removed->parent->left = removed->right;
        }
        else {
            root = removed->right;
            root->parent = NULL;
        }
        removed->right->fix_size();

    }
    else if (removed == root){
```

```
        root = NULL;
    }
    else if (removed->parent->left == removed){
        removed->parent->left = NULL;
        removed->parent->fix_size();
    }
    else if (removed->parent->right == removed) {
        removed->parent->right = NULL;
        removed->parent->fix_size();
    }
    count--;
    delete removed;
    return _key;
}
```

Листинг 5 — функция удаления узла из дерева

```
void IntTree::insert(IntNode * node, int key){
    IntNode * temp = node;
            if (node){
                    if (node->key - key == 0){
          // key already exist
                            return;
                    }else if (node->key < key){
                            if (node->right){


            temp = node;
                                    insert(node->right, key);
                            }
                            else{
                                    IntNode * new_node = new IntNode(key);
                                    new_node->parent = temp;
                                    node->right = new_node;
                                    count++;
            temp->fix_size();
            new_node->fix_size();
```

```
                                return;
                        }
                }else{
                        if (node->left){
        temp = node;
                                insert(node->left, key);
                        }
                        else{
                                IntNode * new_node = new IntNode(key);
        new_node->parent = temp;
                                node->left = new_node;
                                count++;
        temp->fix_size();
        new_node->fix_size();
                                return;
                        }
                }
        }else{
                root = new IntNode(key);
    root->fix_size();
                count++;
                return;
        }

}
```

Листинг 6 — функция вставки узла в дерево

Обход дерева реализован благодаря алгоритму поиска в глубину. При обходе дерева каждый узел выводиться, так организован вывод дерева. Также при помощи обхода реализован поиск глубины (высоты) дерева — если у узла нет родителя — его высота равна 1, в ином случае его высота равна высоте родителя + 1, при этом есть переменная хранящая максимальную высоту. Функция обхода дерева состоит из двух частей — первая функция обходит все узлы и обнуляет значение переменной, которая обозначает посещали узел или нет, вторая функция выполняет проход с целью действия над узлом.

```
void IntTree::dfs_support(IntNode * a){
	static int cnt = 0;
	if (a->visited){
		a->visited = 0;
		cnt++;
	}
	if (cnt != this->count && a->left)
		dfs_support(a->left);
	if (cnt != this->count && a->right)
		dfs_support(a->right);
}
```

Листинг 7 — Функция, отвечающая за выполнение первой части функции обхода

```
void IntTree::dfs(IntNode * a){
	if (a){
		dfs_support(root);
		static int cnt = 0;
		if (!a->visited){
			a->out();
			a->visited = 1;
			cnt += 1;
		}
		if (cnt != this->count && a->left)
			dfs(a->left);
		if(cnt != this->count && a->right)
			dfs(a->right);


		dfs_support(root);
	}
}
```

Листинг 8 — Функция, отвечающая за выполнение второй части функции обхода , где действие над узлом — его вывод

```
void IntTree::find_height(IntNode * a, int * mx) {
   if (a) {
      static int cnt = 0;
                        if (!a->visited){
                                 if (a->parent) {
         a->height = a->parent->height + 1;
        }
        else {
         a->height = 1;
        }
        *mx = *mx > a->height? *mx : a->height;
                                 a->visited = 1;
                                 cnt += 1;
                        }
                        if (cnt != this->count && a->left)
                                 find_height(a->left, mx);
                        if(cnt != this->count && a->right)
                                 find_height(a->right, mx);


                        dfs_support(root);
   }
}
```

Листинг 9 — Функция, отвечающая за выполнение второй части функции обхода , где действие над узлом — поиск высоты

Для реализации функции, строящей рандомизированное дерево необходимы функции вставки в корень, левого и правого поворотов вокруг узла.

```
IntNode* IntTree::rotate_right(IntNode* p) {
   IntNode* q = p->left;
   if (!q) return NULL;
   p->left = q->right;
   if (q->right)
      q->right->parent = p;
   q->right = p;


   if (p->parent) {
```

```
        if (p == p->parent->left)
            p->parent->left = q;
        else
            p->parent->right = q;
    }
    else {
        root = q;
    }
    p->parent = q;
    p->fix_size();
    q->fix_size();
    return q;
}
```

Листинг 10 — функция правого поворота

```
IntNode* IntTree::insert_root(IntNode* p, int k) {
            if (!p) {count++;return new IntNode(k);}

    if (p->key > k) {
        p->left = insert_root(p->left, k);
        p->fix_size();
        return rotate_right(p);

    }
    else {
        p->right = insert_root(p->right, k);
        p->fix_size();
        return rotate_left(p);
    }

}
```

Листинг 11 — функция вставки в корень дерева

```
IntNode* IntTree::insert_random(IntNode* p, int k) {
    if (!p) {count++;return new IntNode(k);}
    if (rand() % (p->size+1) == 0) {
        return insert_root(p, k);
```

```
    }
    if (p->key > k) {
        p->left = insert_random(p->left, k);
        p->left->parent = p;

    }
    else {
        p->right = insert_random(p->right, k);
        p->right->parent = p;
    }
    p->fix_size();
    return p;
}
```

Листинг 12 - Функция вставки, строящая рандомизированное дерево

Заполним дерево английскими буквами, для проверки работоспособности программы

```
before removing
--------------------------------
self    left    right   parent
--------------------------------
--------------------------------
   A    0000    a       0
--------------------------------

--------------------------------
   a    B       b       65
--------------------------------

--------------------------------
   B    0000    C       97
--------------------------------

--------------------------------
   C    0000    D       66
--------------------------------

--------------------------------
   D    0000    E       67
--------------------------------

--------------------------------
   E    0000    F       68
--------------------------------

--------------------------------
   F    0000    G       69
--------------------------------

--------------------------------
   G    0000    H       70
--------------------------------

--------------------------------
   H    0000    I       71
--------------------------------
```

Рисунок 1 — начало функции вывода дерева

```
221
222 ---------------------------------
223    x    0000    y       119
224 ---------------------------------
225
226 ---------------------------------
227    y    0000    z       120
228 ---------------------------------
229
230 ---------------------------------
231    z    0000    0000    121
232 ---------------------------------
233
234 count of nodes: 52
235
236 find nodes with keys z, a, J
```

Рисунок 2 — конец функции вывода дерева

Далее найдём узлы с ключами «z», «a», «j» в дереве и выведем их.

```
234 count of nodes: 52
235
236 find nodes with keys z, a, J
237 ---------------------------------
238    z    0000    0000    121
239 ---------------------------------
240
241 ---------------------------------
242    a    B       b       65
243 ---------------------------------
244
245 ---------------------------------
246    J    0000    K       73
247 ---------------------------------
248
```

Рисунок 3 — вывод найденных ключей

После чего удалим все узлы из дерева

```
48
49 after removing
50 ---------------------------------
51 self    left    right   parent
52 ---------------------------------
53 count of nodes: 0
```

Рисунок 4 — вывод дерева после удаления всех узлов

Таблица 2 — результат расчёта высоты дерева от кол-ва узлов в дереве

Тип ключей	Тип дерева	Кол-во элементов	Высота дерева
случайные	Двоичное дерево поиска	5000	29
		10000	27
		15000	30
		20000	30
		25000	34
упорядоченные	Двоичное дерево поиска	5000	5000
		10000	10000
		15000	15000
		20000	20000
		25000	25000
случайные	Рандомизированное дерево	5000	26
		10000	37
		15000	35
		20000	32
		25000	36
упорядоченные	Рандомизированное дерево	5000	27
		10000	34
		15000	31
		20000	35
		25000	32

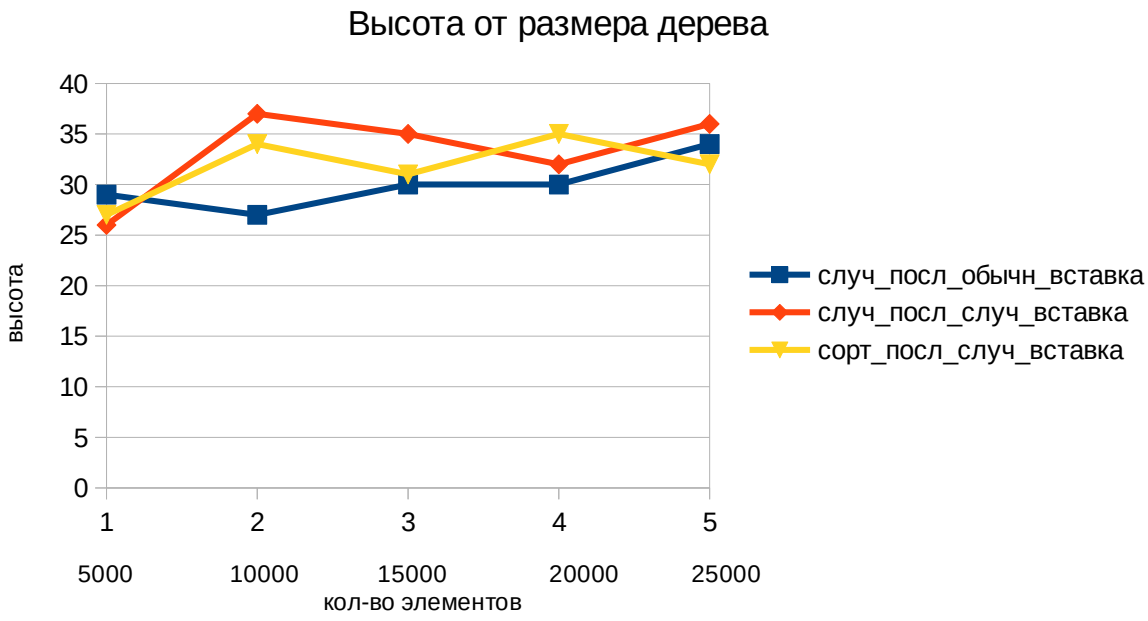

Рисунок 5 — первая часть графика зависимости высоты дерева от кол-ва элементов

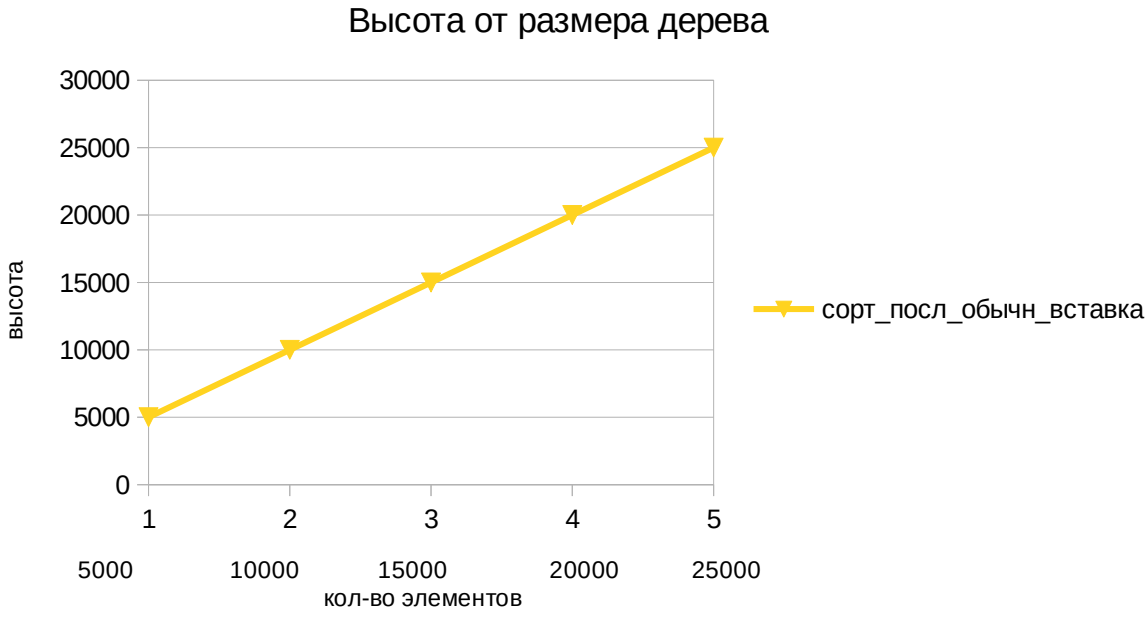

Рисунок 6 — вторая часть графика зависимости высоты дерева от кол-ва элементов

```
void CharTree::target_func(CharNode * a, int *sheets_vowel){
        if (a){
                dfs_support(root);
```

```
                static int cnt = 0;
                if (!a->visited){
        if (a->left == NULL && a->right == NULL &&  (strchr(vowel, a->key) != NULL)) (*sheets_vowel)++;
                        a->visited = 1;
                        cnt += 1;
                }
                if (cnt != this->count && a->left)
                        target_func(a->left, sheets_vowel);
                if(cnt != this->count && a->right)
                        target_func(a->right, sheets_vowel);


                dfs_support(root);
          }
}
```

Листинг 13 — функция для реализации задания индивидуального варианта

Для проверки работоспособности функции для реализации индивидуального варианта было заполнено 2 дерева, содержимое которых представлено на рисунке 7.

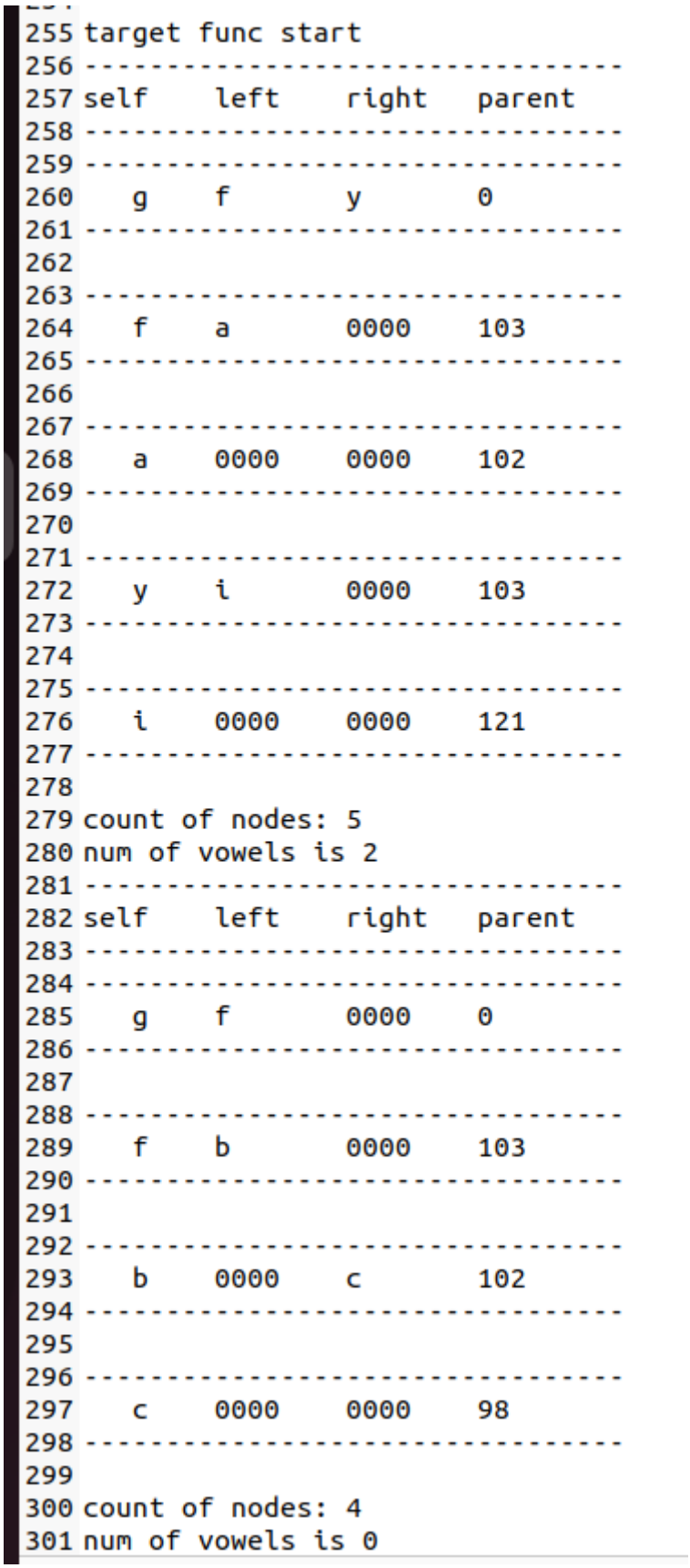

Рисунок 7 — проверка работоспособности функции для реализации задания индивидуального варианта

Анализ свойств алгоритмов

Таблица 3 — анализ алгоритмов взаимодействия в деревьях

Название	Тип дерева	среднее	худшее
Вставка	Двоичное дерево поиска	$O(\log n)$	$O(n)$
поиск		$O(\log n)$	$O(n)$
Удаление		$O(\log n)$	$O(n)$
Нахождение высоты		$O(n)$	$O(n)$
Вставка	Рандомизированное дерево	$O(\log n)$	$O(\log n)$
поиск		$O(\log n)$	$O(\log n)$
Удаление		$O(\log n)$	$O(\log n)$
Нахождение высоты		$O(n)$	$O(n)$

Исходя из графика зависимости высоты от кол-ва элементов можно заметить недостаток несбалансированного двоичного дерева поиска — в случае, когда ключи для вставки отсортированы дерево имеет высоту, равную количеству элементов, из-за чего вставка поиск и удаление будут работать с сложностью O(n), в то время как рандомизированное дерево имеет высоту, равную log(n) благодаря чему сложность данных функций будет O(log(n)).

Вывод

Изучены базовые алгоритмы работы с деревьями: построение, обход, поиска элемента, удаление элемента, подсчет количества узлов, нахождение высоты дерева, исследованы свойства деревьев, разница между сбалансированным двоичным деревом поиска и несбалансированным.