

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ  
Федеральное государственное бюджетное образовательное учреждение  
высшего профессионального образования  
**«Вятский государственный университет»**  
Факультет автоматики и вычислительной техники  
Кафедра электронных вычислительных машин

Лабораторная работа № 2 по курсу  
«Технологии программирования»

Выполнил студент группы ИВТ-21 \_\_\_\_\_/Рзаев А. Э./  
Проверил доцент кафедры ЭВМ \_\_\_\_\_/Долженкова М. Л./

Киров 2017

## 1 Задание

Написать программу для работы с динамической структурой данных (двоичное дерево поиска), содержащей строку и массив чисел с плавающей запятой.

## 2 Работа программы

### 2.1 Занесение элемента

Память 1															
Адрес: 0x0079AB30															
0x0079AB30	01	00	00	00	01	00	00	00	3b	ed	d5	2d	60	00	00
0x0079AB48	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x0079AB60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x0079AB78	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x0079AB90	00	00	00	00	00	00	00	00	30	df	9f	14	ad	2e	00
0x0079ABA8	00	00	00	00	00	00	00	00	1c	00	00	00	01	00	00
0x0079ABC0	b8	ae	79	00	04	00	00	00	18	ac	79	00	02	00	00
0x0079ABD8	00	00	00	00	fd	fd	fd	fd	ab	ab	ab	ab	ab	ab	ab
0x0079ABF0	32	df	9f	16	a9	2e	00	1c	98	ae	79	00	00	00	00

- Указатель на предыдущий занятый элемент
- Указатель на следующий занятый элемент
- Указатель на имя файла подкачки
- Номер строки в файле подкачки
- Размер выделенного участка памяти
- Тип участка памяти
- Количество обращений
- Индикатор начала участка памяти
- Указатель на строку
- Длина строки
- Указатель на массив чисел
- Длина массива
- Указатель на левый дочерний узел
- Указатель на правый дочерний узел
- Указатель на родительский узел
- Индикаторы конца участка памяти

## 2.2 Удаление элемента

Память 1																
Адрес: 0x0079AB30																
0x0079AB30	01	00	00	00	01	00	00	00	3b	ed	d5	2d	60	00	00	00
0x0079AB48	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x0079AB60	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x0079AB78	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
0x0079AB90	00	00	00	00	00	00	00	00	2c	df	9c	0b	ad	2e	00	00
0x0079ABA8	ee	fe	ee	fe	ee	fe	ee	fe	ee	fe	ee	fe	ee	fe	ee	fe
0x0079ABC0	ee	fe	ee	fe	ee	fe	ee	fe	ee	fe	ee	fe	ee	fe	ee	fe
0x0079ABD8	ee	fe	ee	fe	ee	fe	ee	fe	ee	fe	ee	fe	ee	fe	ee	fe
0x0079ABF0	ee	fe	ee	fe	ee	fe	ee	fe	ee	fe	ee	fe	ee	fe	ee	fe

Произошло освобождение области памяти после удаления элемента из дерева.

## 3 Листинг программы

Листинг разработанной программы приведен в приложении А.

## 4 Вывод

В ходе выполнения лабораторной работы была разработана структура данных на динамической памяти – двоичное дерево поиска. Изучена структура элемента памяти кучи.

Приложение А  
(обязательное)  
Листинг программы

```
#include <bits/stdc++.h>

struct node
{
    char* str;
    size_t str_size;
    float* farr;
    size_t farr_size;

    node* left;
    node* right;
    node* parent;
};

struct tree {
    node root; // real root of tree is root.left
};

size_t leaks_counter = 0;

node* allocate_node(size_t str_size, size_t farr_size) {
    ++leaks_counter;

    node* ret = (node*)realloc(nullptr, sizeof(node));
    ret->str = (char*)realloc(nullptr, sizeof(char) * str_size);
    ret->farr = (float*)realloc(nullptr, sizeof(float) * farr_size);
    ret->str_size = str_size;
    ret->farr_size = farr_size;

    ret->left = ret->right = ret->parent = nullptr;

    std::fill_n(ret->str, str_size, '\0');
    std::fill_n(ret->farr, farr_size, 0.0f);
    return ret;
}

node* allocate_node(node* src) {
    node* ret = allocate_node(src->str_size, src->farr_size);

    std::copy(src->str, src->str + src->str_size, ret->str);
    std::copy(src->farr, src->farr + src->farr_size, ret->farr);

    return ret;
}

void deallocate_node(node* n) {
    --leaks_counter;

    realloc(n->str, 0);
    realloc(n->farr, 0);
    realloc(n, 0);
}

bool less_node(node* first, node* second) {
    if (first->str_size == second->str_size &&
        std::equal(first->str, first->str + first->str_size, second->str))
    { // if first->str == second->str
        return std::lexicographical_compare(
            first->farr, first->farr + first->farr_size,
```

```

        second->farr, second->farr + second->farr_size
    );
}
else {
    return std::lexicographical_compare(
        first->str, first->str + first->str_size,
        second->str, second->str + second->str_size
    );
}
}

node* find_node(tree* tr, node* key) {
    node* root = tr->root.left;
    while (root != nullptr) {
        if (less_node(key, root)) { // key < root
            root = root->left;
        }
        else if (less_node(root, key)) { // key > root
            root = root->right;
        }
        else { // key == root, success!
            return root;
        }
    }
    return root; // key not found
}

void remove_node(tree* tr, node* key) {
    node* c = find_node(tr, key);
    if (c == nullptr) { // no such key in tree, nothing to do
        return;
    }

    node* l = c->left,
        *r = c->right,
        *p = c->parent;

    bool key_is_less = p->left == c;

    if (l == nullptr && r == nullptr) {
        if (key_is_less) { // left child
            p->left = nullptr;
        }
        else { //right child
            p->right = nullptr;
        }
    }
    else if (l == nullptr && r != nullptr) {
        if (key_is_less) {
            p->left = r;
        }
        else {
            p->right = r;
        }
    }
    else if (l != nullptr && r == nullptr) {
        if (key_is_less) {
            p->left = l;
        }
        else {
            p->right = l;
        }
    }
    else {

```

```

        node* new_p = r;
        while (r->left != nullptr) {
            new_p = r->left;
        }
        l->parent = new_p;
        new_p->left = l;
        if (key_is_less) {
            p->left = r;
        }
        else {
            p->right = r;
        }
    }
    deallocate_node(c);
}

void insert_node(tree* tr, node* key) {
    bool key_is_less = true;
    node* root = tr->root.left;
    node* parent = &(tr->root);
    while (root != nullptr) {
        if (less_node(key, root)) { // key < root
            key_is_less = true;
            parent = root;
            root = root->left;
        }
        else if (less_node(root, key)) { // key > root
            key_is_less = false;
            parent = root;
            root = root->right;
        }
        else { // key == root, node exists, nothing to do
            return;
        }
    }
    // insert new node
    root = allocate_node(key);
    root->parent = parent;
    if (key_is_less) {
        parent->left = root;
    }
    else {
        parent->right = root;
    }
}

void deallocate_tree(node* root) {
    if (root == nullptr) {
        return;
    }
    deallocate_tree(root->left);
    deallocate_tree(root->right);
    root->left = root->right = nullptr;
    deallocate_node(root);
}

void allocate_tree(tree* tr) {
    tr->root.left = tr->root.right = tr->root.parent = nullptr;
}

void loop(tree* tr) {
    while (true) {
        std::cout << "1. Insert\n2. Find\n3. Remove\n0. Exit\n";
        int code; std::cin >> code;
    }
}

```

```

        if (code == 1) {
            std::string s; int n1, n2;
            std::cin >> n1 >> s >> n2;
            s.resize(n1);
            node* key = allocate_node(n1 + 1, n2);
            for (int i = 0; i < n1; ++i) {
                key->str[i] = s[i];
            }
            for (int i = 0; i < n2; ++i) {
                std::cin >> key->farr[i];
            }
            insert_node(tr, key);
            deallocate_node(key);
        }
        else if (code == 2) {
            std::string s; int n1, n2;
            std::cin >> n1 >> s >> n2;
            s.resize(n1);
            node* key = allocate_node(n1 + 1, n2);
            for (int i = 0; i < n1; ++i) {
                key->str[i] = s[i];
            }
            for (int i = 0; i < n2; ++i) {
                std::cin >> key->farr[i];
            }
            if (find_node(tr, key) != nullptr) {
                std::cout << "key exists\n";
            }
            else {
                std::cout << "key doesn't exist\n";
            }
            deallocate_node(key);
        }
        else if (code == 3) {
            std::string s; int n1, n2;
            std::cin >> n1 >> s >> n2;
            s.resize(n1);
            node* key = allocate_node(n1 + 1, n2);
            for (int i = 0; i < n1; ++i) {
                key->str[i] = s[i];
            }
            for (int i = 0; i < n2; ++i) {
                std::cin >> key->farr[i];
            }
            remove_node(tr, key);
            deallocate_node(key);
        }
        else if (code == 0) {
            break;
        }
    }

}

int main(void) {
    tree tr = { nullptr };
    loop(&tr);
    deallocate_tree(tr.root.left);
    std::cout << "leaks: " << leaks_counter << std::endl;
    return 0;
}

```