

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

ОТЧЕТ
по лабораторной работе №5
по дисциплине «Алгоритмы и структуры данных»
Тема: Бинарное дерево поиска

Студент гр. 9382

Дерюгин Д.А.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Изучить случайное бинарное дерево поиска. Написать реализацию случайного БДП, реализовать удаление элементов из этого дерева.

Задание.

Вариант 8.

БДП: случайное* БДП; действие: 1+2б

1) По заданной последовательности элементов Elem построить структуру данных определённого типа – БДП или хеш-таблицу;

2) Выполнить одно из следующих действий:

Предусмотреть возможность повторного выполнения с другим элементом.

б) Для построенной структуры данных проверить, входит ли в неё элемент e типа Elem, и если входит, то удалить элемент e из структуры данных (первое обнаруженное вхождение). Предусмотреть возможность повторного выполнения с другим элементом.

Основные теоретические положения.

Бинарное дерево поиска - бинарное дерево, для которого выполняются условия:

Оба поддерева - бинарные деревья поиска

У всех узлов левого поддерева значения ключей меньше, нежели значения ключа родительского дерева

У всех узлов правого поддерева значения ключей больше, нежели значения ключа родительского дерева

Структура же случайного БДП полностью зависит от того порядка, в котором элементы расположены во входной последовательности.

Описание алгоритма.

На вход подается последовательность символом. В цикле перебираем все элементы этой последовательности. Каждый элемент сравниваем с узлом дерева. Если элемент меньше узла, значит рекурсивно вызываем левое поддерево данного узла. Если элемент больше данного узла, рекурсивно

вызываем правое поддерево данного узла. Если поддерева нет, создаем его и записываем данный элемент в это поддерево.

Удаление узла.

Рекурсивно ищем удаляемый узел(если удаляемый узел меньше главного узла, идем в левое поддерево, если удаляемый узел больше главного узла, идем в правое поддерево). Когда нашли удаляем элемент, смотрим на количество таких элементов, если кол-во больше 1, просто вычитаем единицу из данной переменной. Если же кол-во равно 1:

Если узел - лист, просто удаляем его;

Если узел имеет одно поддерево(правое или левое) просто удаляем его и перевешиваем на это место его поддерево;

Если узел имеет два поддерева:

Ищем минимальный элемент в правом поддереве;

Если этот элемент не имеет поддерево:

Переставляем его на место удаляемого элемента

Если этот элемент имеет поддерево:

Этот элемент переставляем на место удаляемого элемента

Его поддерево переставляем на его место.

Структуры данных.

RandomBinarySearchTree - Структура, которая является случайным БДП.

RandomBinarySearchTree* left - левое Случайное БДП

RandomBinarySearchTree* right- правое Случайное БДП

RandomBinarySearchTree* parent - родитель

Int count - количество вхождений данного элемента

Char data - сам элемент.

Описание функций.

RandomBinarySearchTree* createTree(char data, RandomBinarySearchTree* randomBinarySearchTree, RandomBinarySearchTree* parent) - Функция, которая создает узел случайного бинарного дерева поиска

char data - Данные узла

RandomBinarySearchTree* randomBinarySearchTree - главное дерево, в которое будет вставлен новый узел

RandomBinarySearchTree* parent - родитель нового узла

Возвращает новый узел

RandomBinarySearchTree*searchLowest(RandomBinarySearchTree* randomBinarySearchTree, RandomBinarySearchTree* temp) - функция поиска самого маленького узла(нужно для удаления элемента).

RandomBinarySearchTree* randomBinarySearchTree - дерево, в котором ищем минимальный узел

RandomBinarySearchTree* temp - копия узла, который удаляем

Возвращает новое дерево

void removeElem(RandomBinarySearchTree* randomBinarySearchTree) - функция удаления элемента.

RandomBinarySearchTree* randomBinarySearchTree - узел, который нужно удалить

void search(char elem, RandomBinarySearchTree* randomBinarySearchTree) - функция поиска удаляемого элемента.

char elem - элемент, который нужно удалить

RandomBinarySearchTree* randomBinarySearchTree - само дерево

void printLKP(RandomBinarySearchTree* randomBinarySearchTree) - Вывод в консоль дерева в обходе КЛП.

RandomBinarySearchTree* randomBinarySearchTree - дерево, которое нужно вывести

void removeElement(RandomBinarySearchTree* randomBinarySearchTree, int mode) - функция, которая считывает удаляемые элементы.

RandomBinarySearchTree* randomBinarySearchTree - дерево, из которого нужно удалить элемент)

int mode - показывает, откуда брать удаляемые символы(из консоли или из файла)

void enterTree() - функция ввода дерева

Тестирование.

Входные данные	Выходные данные
Дерево: a Удаляемый элемент: a	empty tree
Дерево: bac Удаляемый элемент: a	a < b move to left tree Create new leaf with data: a c > b move to right tree Create new leaf with data: c If you want to stop press '+' Enter remove element: a LKP of tree: b[1]c[1]
Дерево: jffaljsighrkjairogjapw Удаляемый элемент: sdgrypwerkgal	LKP of tree: a[3]f[2]g[2]h[1]i[2]j[4]k[1]l[1]o[1]p[1]r[2]w[1] There no this element LKP of tree: a[3]f[2]g[2]h[1]i[2]j[4]k[1]l[1]o[1]p[1]r[2]w[1] LKP of tree: a[3]f[2]g[1]h[1]i[2]j[4]k[1]l[1]o[1]p[1]r[2]w[1] LKP of tree: a[3]f[2]g[1]h[1]i[2]j[4]k[1]l[1]o[1]p[1]r[1]w[1] There no this element LKP of tree: a[3]f[2]g[1]h[1]i[2]j[4]k[1]l[1]o[1]p[1]r[1]w[1] LKP of tree:

	a[3]f[2]g[1]h[1]i[2]j[4]k[1]l[1]o[1]r[1]w[1] LKP of tree: a[3]f[2]g[1]h[1]i[2]j[4]k[1]l[1]o[1]r[1] There no this element LKP of tree: a[3]f[2]g[1]h[1]i[2]j[4]k[1]l[1]o[1]r[1] LKP of tree: a[3]f[2]g[1]h[1]i[2]j[4]k[1]l[1]o[1] LKP of tree: a[3]f[2]h[1]i[2]j[4]k[1]l[1]o[1] LKP of tree: a[3]f[2]h[1]i[2]j[4]l[1]o[1] LKP of tree: a[2]f[2]h[1]i[2]j[4]l[1]o[1] LKP of tree: a[2]f[2]h[1]i[2]j[4]o[1]
Дерево: Удаляемый элемент: k	empty tree

Выводы.

В результате данной лабораторной работы была реализована структура случайного БДП, а также удаление элемента из этого дерева.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <fstream>
using namespace std;

struct RandomBinarySearchTree {
    RandomBinarySearchTree* left;// left subtree
    RandomBinarySearchTree* right;// right subtree
    RandomBinarySearchTree* parent;// parent of tree
    int count;// count of char
    char data;// value
};

RandomBinarySearchTree* createTree(char data, RandomBinarySearchTree*
randomBinarySearchTree, RandomBinarySearchTree* parent) {
    //create new element of tree
    if (randomBinarySearchTree == nullptr) {
        cout<<"Create new leaf with data: "<<data<<endl;
        randomBinarySearchTree = new RandomBinarySearchTree;
        randomBinarySearchTree->data = data;
        randomBinarySearchTree->count = 1;
        randomBinarySearchTree->left = nullptr;
        randomBinarySearchTree->right = nullptr;
        randomBinarySearchTree->parent = parent;
    }
    else if (data < randomBinarySearchTree->data) {
        cout<<data<<" < "<< randomBinarySearchTree->data<<" move to left
tree\n";
        randomBinarySearchTree->left = createTree(data, randomBinarySearchTree-
>left, randomBinarySearchTree);
    }
    else if (data == randomBinarySearchTree->data) {
        cout<<data<<" = "<< randomBinarySearchTree->data<<" move there\n";
        randomBinarySearchTree->count++;
    }
    else {
        cout<<data<<" > "<< randomBinarySearchTree->data<<" move to right
tree\n";
        randomBinarySearchTree->right = createTree(data, randomBinarySearchTree-
>right, randomBinarySearchTree);
    }
}
```

```

    }
    return randomBinarySearchTree;
}

RandomBinarySearchTree* searchLowest(RandomBinarySearchTree*
randomBinarySearchTree, RandomBinarySearchTree* temp) {
    // if there is left subtree
    if (randomBinarySearchTree->left) searchLowest(randomBinarySearchTree->left,
temp);
    // if subtree hasn't right subtree
    else if (!randomBinarySearchTree->right) {
        temp->data = randomBinarySearchTree->data;
        temp->count = randomBinarySearchTree->count;
        if (randomBinarySearchTree->parent->right->data ==
randomBinarySearchTree->data)
            randomBinarySearchTree->parent->right = nullptr;
        else if (randomBinarySearchTree->parent->left->data ==
randomBinarySearchTree->data)
            randomBinarySearchTree->parent->left = nullptr;
        return temp;
        //if left subtree has right subtree
    } else if (randomBinarySearchTree->right) {
        temp->data = randomBinarySearchTree->data;
        temp->count = randomBinarySearchTree->count;
        randomBinarySearchTree->data = randomBinarySearchTree->right->data;
        randomBinarySearchTree->count = randomBinarySearchTree->right->count;
        randomBinarySearchTree->left = randomBinarySearchTree->right->left;
        randomBinarySearchTree->right = randomBinarySearchTree->right->right;
        return temp;
    }
}

void removeElem(RandomBinarySearchTree* randomBinarySearchTree) {
    if (!randomBinarySearchTree->parent && randomBinarySearchTree->count ==1) {
        delete randomBinarySearchTree;
        cout<<"empty tree";
        exit(1);
    }
    // if there is more than 1 element with this value
    if (randomBinarySearchTree->count > 1) randomBinarySearchTree->count--;
    // remove leaf

```



```

        else if (!randomBinarySearchTree->left && !randomBinarySearchTree->right &&
randomBinarySearchTree->count == 1) {
            if (randomBinarySearchTree->parent->right->data ==
randomBinarySearchTree->data)
                randomBinarySearchTree->parent->right = nullptr;
            else if (randomBinarySearchTree->parent->left->data ==
randomBinarySearchTree->data)
                randomBinarySearchTree->parent->left = nullptr;
        }

//if remove node has only left subtree
else if (randomBinarySearchTree->left && !randomBinarySearchTree->right) {
    randomBinarySearchTree->data = randomBinarySearchTree->left->data;
    randomBinarySearchTree->count = randomBinarySearchTree->left->count;
    randomBinarySearchTree->left = randomBinarySearchTree->left->left;
    if (randomBinarySearchTree->left)
        randomBinarySearchTree->right = randomBinarySearchTree->left->right;
}

// if remove node has only right subtree
else if (!randomBinarySearchTree->left && randomBinarySearchTree->right) {
    randomBinarySearchTree->data = randomBinarySearchTree->right->data;
    randomBinarySearchTree->count = randomBinarySearchTree->right->count;
    randomBinarySearchTree->left = randomBinarySearchTree->right->left;
    randomBinarySearchTree->right = randomBinarySearchTree->right->right;
}

//if remove leaf has left and right subtree
else if (randomBinarySearchTree->left && randomBinarySearchTree->right) {
    RandomBinarySearchTree* temp = randomBinarySearchTree;
    randomBinarySearchTree = searchLowest(randomBinarySearchTree->right,
temp);
}

}

void search(char elem, RandomBinarySearchTree* randomBinarySearchTree) {
    // cannot find element in tree
    if (!randomBinarySearchTree) {
        cout<<"There no this element\n";
        return;
    }
    if (elem < randomBinarySearchTree->data)
        search(elem,
randomBinarySearchTree->left);

```

```

        else if (elem > randomBinarySearchTree->data) search(elem,
randomBinarySearchTree->right);
        else removeElem(randomBinarySearchTree);
    }

void printLKP(RandomBinarySearchTree* randomBinarySearchTree) {
    if (randomBinarySearchTree->left) printLKP(randomBinarySearchTree->left);
    cout<<randomBinarySearchTree->data<<"["<<randomBinarySearchTree->count<<"]";
    if (randomBinarySearchTree->right) printLKP(randomBinarySearchTree->right);
}

void removeElement(RandomBinarySearchTree* randomBinarySearchTree, int mode) {
    char removeElem;// remove element
    string tmp;// rubbish
    if (mode) {
        do {
            cout<<"If you want to stop press '+'\nEnter remove element:\n";
            cin>>removeElem;
            search(removeElem, randomBinarySearchTree);
            cout<<"LKP of tree:\n";
            printLKP(randomBinarySearchTree);
            cout<<"\n";//
        } while (removeElem != '+');
    } else {
        ifstream fin;
        fin.open("input.txt");
        //if cannot open file
        if (!fin.is_open()) {
            cout<<"Cannot open file";
            exit(1);
        }
        getline(fin, tmp);
        do {
            //reading file char by char
            fin>>removeElem;
            if (!fin.eof()) {
                search(removeElem, randomBinarySearchTree);
                cout<<"LKP of tree:\n";
                printLKP(randomBinarySearchTree);
                cout<<"\n";
            }
        }
    }
}

```

```

        } while (!fin.eof());

        fin.close();//close file
    }

}

void enterTree() {
    auto* randomBinarySearchTree = new RandomBinarySearchTree;// create tree
    string rawTree;    // string of tree
    string path = "input.txt"; // path to input file
    int typeOfInput;// 1 if console
    cout<<"Enter '1' if you wanna write down binary tree in console otherwise
write down any letter or number:\n";
    cin>>typeOfInput;
    //input from console
    if (typeOfInput == 1){
        cout<<"Enter tree:\n";
        cin>>rawTree;
        //if tree empty
        if (rawTree.empty())
            return;
    }
    else {
        //open file
        ifstream fin;
        fin.open(path);
        //if cannot open file
        if (!fin.is_open()) {
            cout<<"Cannot open file";
            exit(1);
        }
        //reading file line by line
        getline(fin, rawTree);
        if (rawTree.empty()) {
            cout<<"empty tree";
            exit(1);
        }
        fin.close();//close file
    }
    //create main node

```

```

randomBinarySearchTree->data = rawTree[0];
randomBinarySearchTree->count = 1;
randomBinarySearchTree->left = nullptr;
randomBinarySearchTree->right = nullptr;
randomBinarySearchTree->parent = nullptr;
// loop for adding element in tree
for (int i = 1; i < rawTree.length(); i++) {
    randomBinarySearchTree = createTree(rawTree[i], randomBinarySearchTree,
randomBinarySearchTree);
}
if (typeOfInput == 1)
    removeElement(randomBinarySearchTree, 1);
else removeElement(randomBinarySearchTree, 0);
}
int main() {
    enterTree();
    return 0;
}

```