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

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: Случайные БДП - вставка и исключение

Студент гр. 9382 _____ Дерюгин Д.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент Дерюгин Д.А.

Группа 9382

Тема работы: Случайные БДП - вставка и исключение. Исследование (в среднем, в худшем случае)

Исходные данные:
Программа генерирует количество деревьев и их содержание. Пользователь вводит количество удаляемых и вставляемых элементов.

Содержание пояснительной записки:
«Содержание», «Описание программы», «Тестирование», «Исследование», «Заключение», «Список использованных источников»

Предполагаемый объем пояснительной записки:
Не менее 15 страниц.

Дата выдачи задания: 03.11.2020

Дата сдачи реферата: 23.12.2020

Дата защиты реферата: 23.12.2020

Студент _____ Дерюгин Д.А.

Преподаватель _____ Фирсов М.А.

АННОТАЦИЯ

В ходе выполнения курсовой работы была разработана программа, которая исследует алгоритм вставки и удаления элементов в случайном БДП. Программа создает деревья и заполняет их, а также подсчитывает число итераций, требуемых для вставки и удаления элементов. В результате исследования было выявлено, что теоретические сведения совпадают с практическими.

SUMMARY

In the course of the course work there was a program that explores the algorithm for inserting and deleting elements in a random BDP. The program creates trees and populates them, and also counts the number of iterations required to insert and remove elements. As a result of the study, it was revealed that theoretical information coincides with practical information.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Задание

Случайные БДП - вставка и исключение. Исследование (в среднем, в худшем случае). Вариант 8

Основные задачи

Генерация входных данных, использование их для измерения количественных характеристик структур данных, алгоритмов, действий, сравнение экспериментальных результатов с теоретическими.

Методы решения

Разработка программы велась в операционной системе Windows 10 в среде разработки Clion. Для реализвации случайного бинарного дерева поиска была создана стукрута RandomBinarySearchTree, а также несколько функий, которые создают и обрабатывают дерево.

Теоретические положения

Случайное бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева, в структуре случайного БДП есть поле, которое отвечает за количество одинаковых элементов. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде.

1. ОПИСАНИЕ ПРОГРАММЫ

1.1 ИСПОЛЬЗУЕМЫЕ СТРУКТУРЫ

struct RandomBinarySearchTree

left - левое поддерево случайного БДП

right - правое поддерево случайного БДП

parent - родитель узда

count - количество вхождений данного элемента

data - сам элемент

height - высота данного элемента

struct Results

countOfTrees - количество сгенерированных деревьев

countOfNumbersInTree -количество элементов в деревьях

depth - глубина дерева

avarageIterationsOfInsert - среднее число итераций для вставки элемента

avarageIterationsOfDeletion - среднее число итераций для удаления элемента

1.2 ИСПОЛЬЗУЕМЫЕ ФУНКЦИИ

RandomBinarySearchTree* createTree(int data, RandomBinarySearchTree* randomBinarySearchTree, RandomBinarySearchTree* parent) - функция создания дерева.

Int data - элемент дерева

RandomBinarySearchTree* rbsr - случайное БДП

randomBinarySearchTree* parent - родитель нового узла

RandomBinarySearchTree* findMax(RandomBinarySearchTree* root) - функция поиска максимального элемента левого поддерева(см. Описание алгаритма удаления)

RandomBinarySearchTree* root - случайное БДП

RandomBinarySearchTree* deleteNode(RandomBinarySearchTree* root, int data, bool replaced, int& sumOfIteration, bool& empty) - функция удаления элемента

RandomBinarySearchTree* root - корневой элемент дерева

Int data - самое значение, которое надо удалить

Bool replaced - требуется при удалении(см. Описание алгоритма удаления)

Int sumOfIteration - ссылка на переменную подсчета итераций

Bool& empty - переменная для проверки пустоты дерева

void enterTree(int* rawTree, int numbersInLine, RandomBinarySearchTree* randomBinarySearchTree) - функция инициализации начальных значений случайного БДП

rawTree - массив элементов случайного БДП

numbersInLine - количество элементов в случайном БДП

RandomBinarySearchTree* rbst - само бинарное дерево

int height(RandomBinarySearchTree* randomBinarySearchTree) - функция поиска высоты данного элемента(см. описание создания деревьев)

RandomBinarySearchTree* rbst - случайное БДП

bool searchIdeal(RandomBinarySearchTree* randomBinarySearchTree) - фунция поиска идеального БДП

int searchAndInsert(int data, RandomBinarySearchTree* randomBinarySearchTree, RandomBinarySearchTree* parent) - фукция поиска и вставки элемента

Int data - элемент, который необходимо вставить

RandomBinarySearchTree* rbsr - само случайное БДП

RandomBinarySearchTree* parent - родитель

int maxDepth(RandomBinarySearchTree* randomBinarySearchTree) - функция поиска глубины дерева.

RandomBinarySearchTree* randomBinarySearchTree - случайное БДП

void generateInsertElements(RandomBinarySearchTree randomBinarySearchTree, int countOfTrees, Results* results)** - функция генерации вставляемых элементов.

RandomBinarySearchTree** randomBinarySearchTree - массив случайных БДП

Int coutnOfTrees - количество деревьев

Results* results - структура результатов

void generateRemoveElements(RandomBinarySearchTree randomBinarySearchTree, int countOfTrees, Results* results)** - функция генерации удаляемых элементов.

RandomBinarySearchTree** randomBinarySearchTree - массив случайных БДП

Int coutnOfTrees - количество деревьев

Results* results - структура результатов

void printResults(Results* results) - функция вывода результатов

Results* resutls - структура результатов

void removeIdeal(RandomBinarySearchTree randomBinarySearchTree, int countOfTrees, Results* results, const int* countInLine)** - функция удаления идеальных БДП

RandomBinarySearchTree** randomBinarySearchTree - массив деревьев

Int countOfTrees - количество деревьев

Results* result - структура результатов

Const int* countInLine - массив элементов деревьев.

void makeTrees() - функция генерации изначальных данных

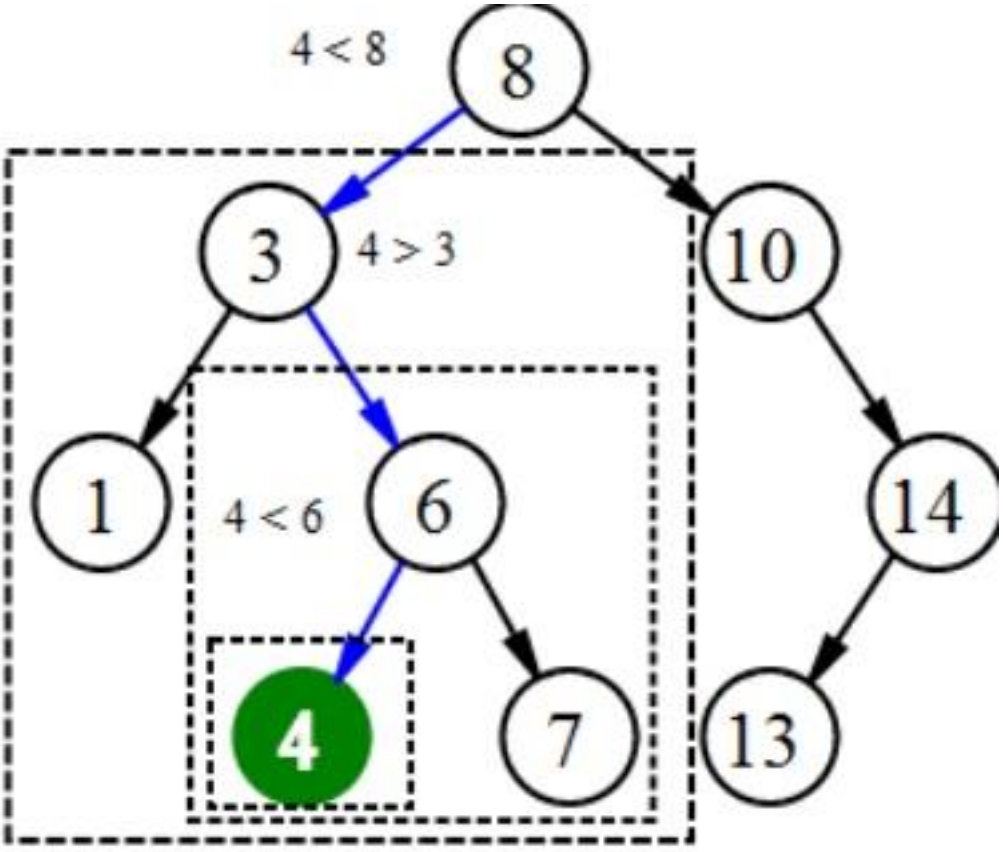
2. ГЕНЕРАЦИЯ ДЕРЕВЬЕВ

Программа генерирует случайное число, которое обозначаент количество случайных БДП. После этого она генерирует массив элементов каждого случайного БДП.(Длина массива также определяется случайно для каждого дерева). Затем, на основе этих данных, алгоритм создает случайые БДП. Так как требуется исследовать только средний и худший случай вставки и удаления, нужно исколючить все деревья, которые оказались идеальными. (дерево называется идеальным, если разница между высотами левого и правого поддерева <= 1). Сгенерированный массив деревьев(уже без идеальных) отправляется в следующую функция для вставки элементов).

3. ВСТАВКА ЭЛЕМЕНТА

Сначала программа просит ввести количество вставляемых элементов. После этого она генерирует набор случайных чисел, которые нужно будет вставить с деревья. Поиск и вставку выполняет рекурсивная функия searchAndInsert. При первом вызове ей подается корень дерева. Проверяется этот корень. Если вставляемы элемент меньше данного корнья, то вызывается эта же функция но с левым поддеревом. Если вставляемый элемент является корнем, то увеличиваем на единицу количество вхождений этого элемента. Если вставляемый элемент больше корня, то вызывается эта функция с правым поддеревом. Если же корень пустой, то создается новый узел с данным элементом.

Пример вставки элемента 4 в дерево:



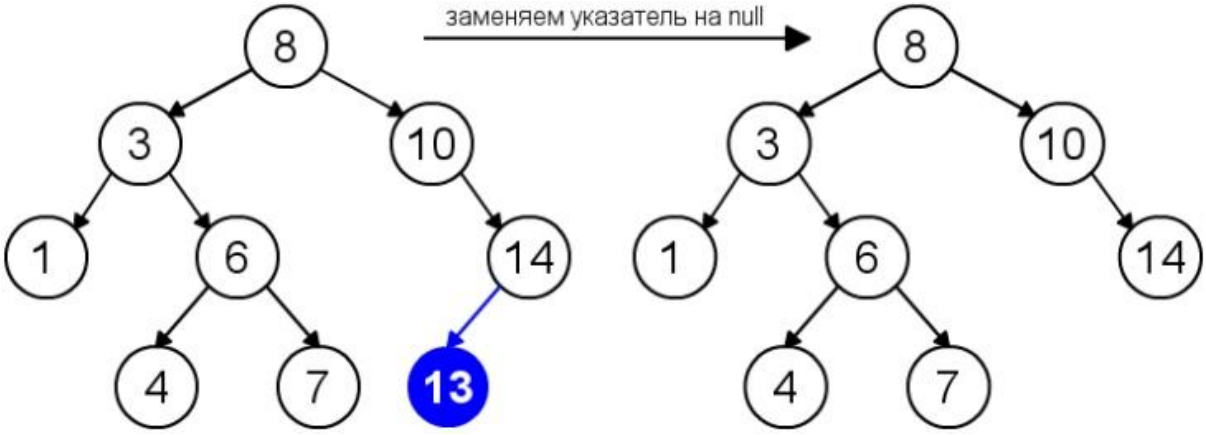
4. УДАЛЕНИЕ ЭЛЕМЕНТА

Удаление делится на несколько вариантов:

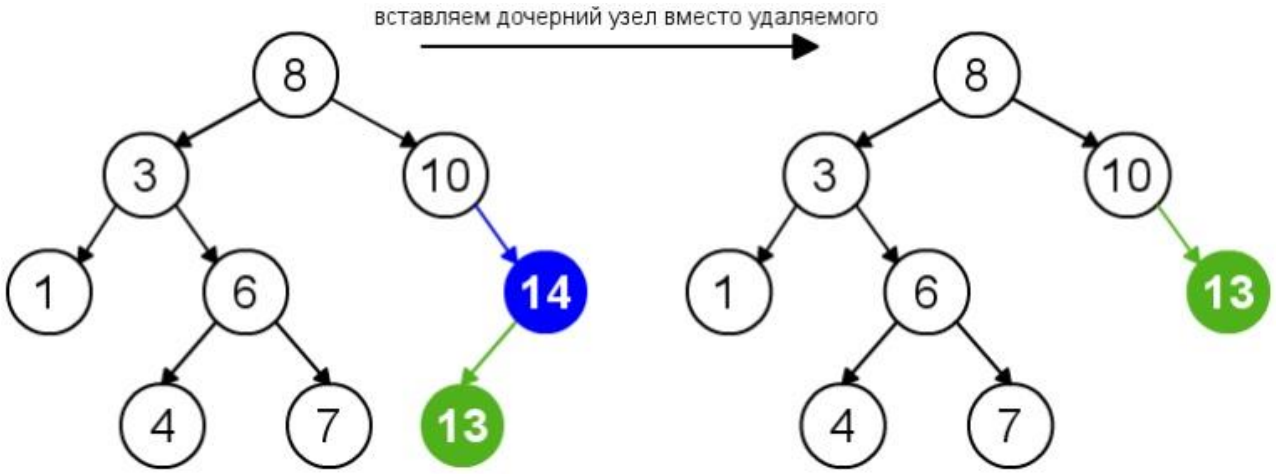
- элемента в дереве нет - если удаляемый элемент не найден, то функция удаления ничего не делает;

- удаляемый элемент имеет несколько копий - если удаляемый элемент имеет несколько вхождений, то уменьшается на 1 поле, которое отвечает за количество элемента;

- удаляемый элемент не имеет поддеревьев(лист) - если удаляемый элемент не имеет поддеревьев, то он удаляется, а ссылка на него в родительском дереве обнуляется;

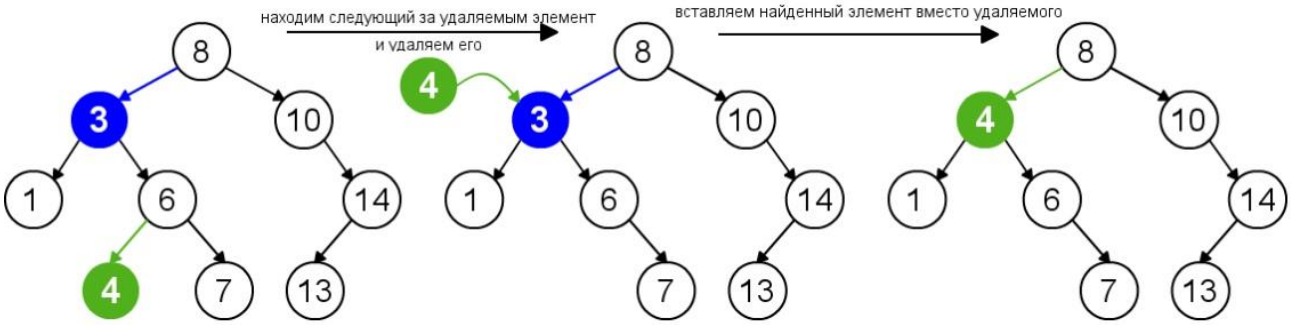


- удаляемый элемент имеет одно поддерево - если удаляемый элемент имеет только левое или только правое поддерево, то на место этого элемента ставится его поддерево



- удаляемый элемент имеет два поддерева - если удаляемый элемент имеет два поддерева, то ищется максимальный элемент левого поддерева(либо

минимальный элемент правого поддерева) и переставляется на место удаляемого элемента



5. ТЕСТИРОВАНИЕ

```
Press 'q' to show intermediate results, otherwise press any button
z
Built 66 trees

Enter count of element which will be insert
10000

Enter count of element which will be delete
10000




Count of trees: 66.




Average iteration to insert of all trees: 10.7647
Average iteration to deletion of all trees: 9.60599
Average n: 498
```

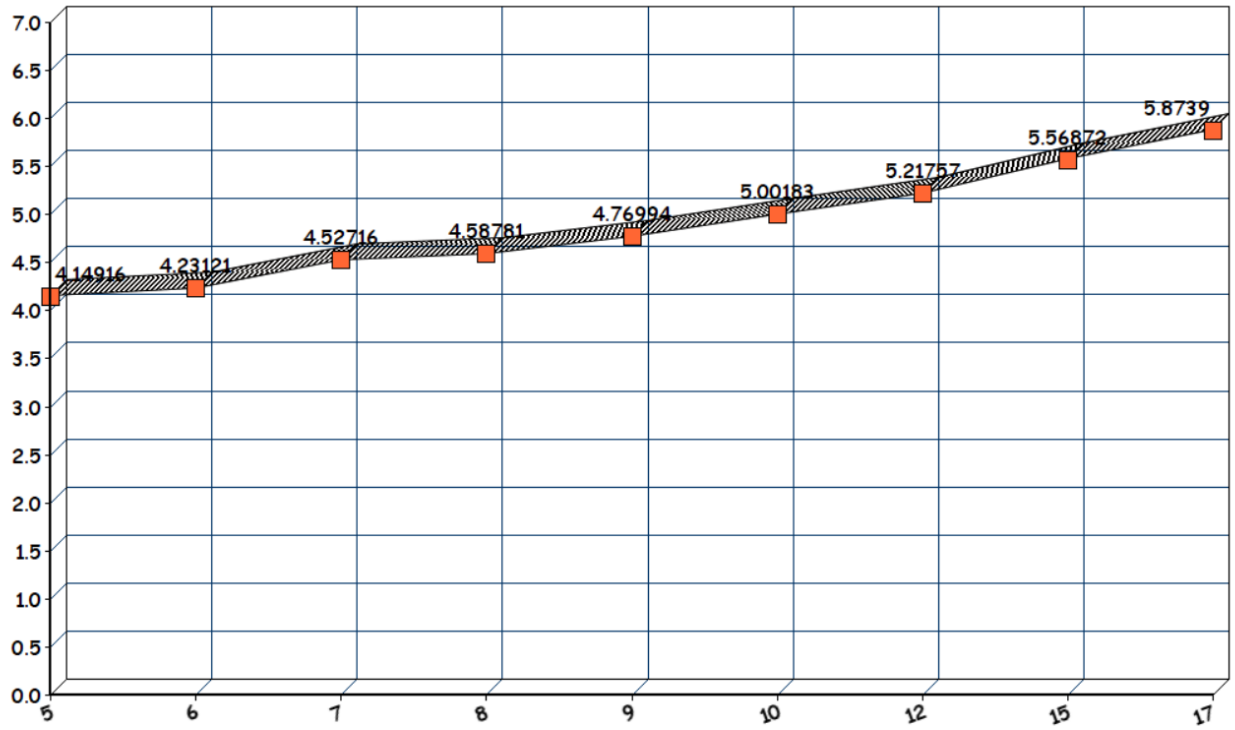
В результате работы программа выдает среднее количество итераций для вставки и удаления, а также среднее количество элементов в дереве.

6. ИССЛЕДОВАНИЕ

6.1 Тестирование среднего количества итераций для вставки элемента в бинарное дерево.

Количество элементов в случайном БДП	**Количество итераций для вставки элементов**
5	4.14916
6	4.23121
7	4.52716
8	4.58781
9	4.76994
10	5.00183
12	5.21757
15	5.56872
17	5.8739

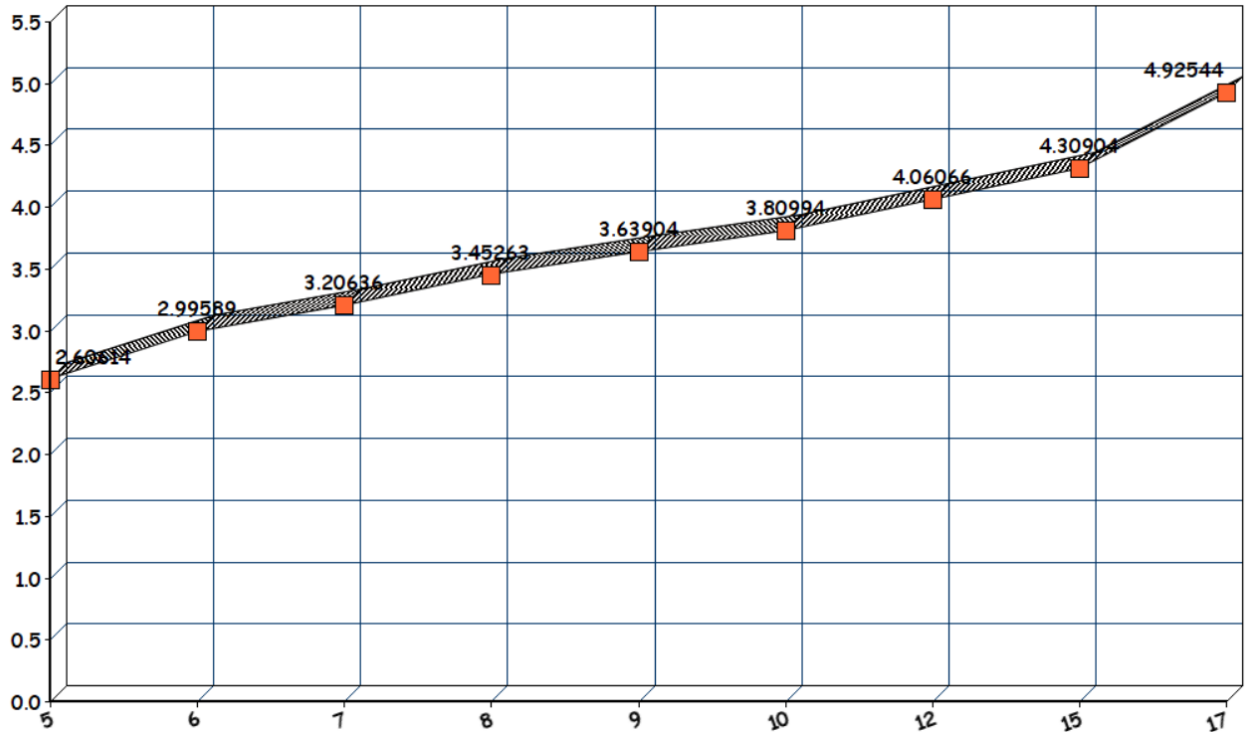
График зависимости итераций для вставки от количества элементов в случайном БДП



6.2 Тестирование среднего количества итераций для удаления элемента из бинарное дерево.

Количество элементов в случайном БДП	Количество итераций для удаления элементов
5	2.60614
6	2.99589
7	3.20636
8	3.45263
9	3.63904
10	3.80994
12	4.06066
15	4.30904
17	4.92544

График зависимости итераций для удаления от количества элементов в случайном БДП



На графиках видно, что средняя количество итераций по вставке и удалению в случайном БДП в среднем случае O(Lnn) и в худшем случае O(n).

7. ЗАКЛЮЧЕНИЕ

В ходе данной курсовой работы была написана программа для генерации случайных БДП и исследования зависимости количества итераций для удаления и вставки от количества элементов в случайном БДП. Была исследована сложность алгоритмов вставки и удаления элементов. Экспрериментальные данные, полученные входе работы, совпали с теоретическими. В среднем операция вставки и удаления выполняются за O(logn), а в худшем случае за за O(n).

8. СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. https://en.wikipedia.org/wiki/Binary_search_tree

2. https://www.geeksforgeeks.org/binary-search-tree-set-2-delete/

3. https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/

4. https://stackoverflow.com/questions/526718/search-times-for-binary-search-tree#:~:text=In%20general%2C%20a%20balanced%20binary,more%20values%20than%20their%20parents).

9. ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <fstream>
#include <time.h>
#include <cmath>

#define MAX_NUMBER 1000
#define MAX_COUNT_OF_TREES 100
static bool mode;

using namespace std;

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

struct Results {
    int countOfTrees;// count of trees
    int* countOfNumbersInTree;// array with counts of numbers in trees
    int* depth;// depth of trees
    double* averageIterationsOfInsert;// average count of iteration
    double* averageIterationsOfDeletion;// average count of deletion
};


RandomBinarySearchTree*     createTree(int     data,     RandomBinarySearchTree*
randomBinarySearchTree, RandomBinarySearchTree* parent) {
    //create new element of tree
    if (randomBinarySearchTree == nullptr) {
        randomBinarySearchTree = new RandomBinarySearchTree;
        randomBinarySearchTree->data = data;
        randomBinarySearchTree->count = 1;
        randomBinarySearchTree->left = nullptr;
        randomBinarySearchTree->right = nullptr;
        randomBinarySearchTree->parent = parent;
```

```
        randomBinarySearchTree->height = parent->height + 1;
    }
    else if (data < randomBinarySearchTree->data) {
        randomBinarySearchTree->left = createTree(data, randomBinarySearchTree->left, randomBinarySearchTree);
    }
    else if (data == randomBinarySearchTree->data) {
        randomBinarySearchTree->count++;
    }
    else {
        randomBinarySearchTree->right = createTree(data, randomBinarySearchTree->right, randomBinarySearchTree);
    }
    return randomBinarySearchTree;
}

RandomBinarySearchTree* findMax(RandomBinarySearchTree* root) {

    if(root == nullptr) return nullptr;

    while(root->right) {
        root = root->right;
    }
    return root;
}

RandomBinarySearchTree* deleteNode(RandomBinarySearchTree* root, int data, bool replaced, int& sumOfIteration, bool& empty) {
    //cannot find tree
    if(root == nullptr) {
        if (mode) cout<<"There no element: "<<data<<endl;
        empty = true;
        return root;
    }
    else if(data < root->data)
        root->left = deleteNode(root->left, data, replaced, sumOfIteration, empty);
    else if (data > root->data)
        root->right = deleteNode(root->right, data, replaced, sumOfIteration, empty);
    else {
        //there no subtree
```

```
if (root->count > 1 && replaced) {
    root->count--;
    if (mode) {
        cout<<"Element: "<<data<<" will be deleted. ";
        cout<<"Count of iterations: "<<root->height<<endl;
    }
    sumOfIteration+= root->height;
}
else if (!root->parent && !root->left && !root->right) {
    if (mode) {
        cout<<"empty tree";
    }
    return nullptr;
}
else if(root->right == nullptr && root->left == nullptr){
    if (mode) {
        cout<<"Element: "<<data<<" will be deleted. ";
        cout<<"Count of iterations: "<<root->height<<endl;
    }
    sumOfIteration+= root->height;
    delete root;
    root = nullptr;
}
    //has right subtree
else if(root->right == nullptr) {
    if (mode) {
        cout<<"Element: "<<data<<" will be deleted. ";
        cout<<"Count of iterations: "<<root->height<<endl;
    }
    sumOfIteration+= root->height;
    root = root->left;
}
// has left subtree
else if(root->left == nullptr) {
    if (mode) {
        cout<<"Element: "<<data<<" will be deleted. ";
        cout<<"Count of iterations: "<<root->height<<endl;
    }

    sumOfIteration+= root->height;
    root= root->right;
}
```

```
            // has two subtrees
        else {
            RandomBinarySearchTree* temp = findMax(root->left);
            if (mode) {
                cout<<"Element: "<<data<<" will be deleted. ";
                cout<<"Count of iterations: "<<root->height<<endl;
            }
            sumOfIteration+= root->height;
            root->data = temp->data;
            root->count = temp->count;
            root->left     =     deleteNode(root->left,     temp->data,     false,
sumOfIteration, empty);
        }
    }
    return root;
}


void   enterTree(int*   rawTree,   int   numbersInLine,   RandomBinarySearchTree*
randomBinarySearchTree) {

    //create main node
    randomBinarySearchTree->data = rawTree[0];
    randomBinarySearchTree->count = 1;
    randomBinarySearchTree->left = nullptr;
    randomBinarySearchTree->right = nullptr;
    randomBinarySearchTree->parent = nullptr;
    randomBinarySearchTree->height = 1;
    // loop for adding element in tree
    for (int i = 1; i < numbersInLine; i++) {
       randomBinarySearchTree  =  createTree(rawTree[i],  randomBinarySearchTree,
randomBinarySearchTree);
    }
}

int height(RandomBinarySearchTree* randomBinarySearchTree) {
    if (!randomBinarySearchTree) return 0;
    if  (height(randomBinarySearchTree->left)  >  height(randomBinarySearchTree-
>right)) return height(randomBinarySearchTree->left) + 1;
    else return height(randomBinarySearchTree->right) + 1;
}
```

```
bool searchIdeal(RandomBinarySearchTree* randomBinarySearchTree) {
    if (randomBinarySearchTree == nullptr) return true;
    return                  (searchIdeal(randomBinarySearchTree->left)         &&
searchIdeal(randomBinarySearchTree->right)
    && abs(height(randomBinarySearchTree->left) - height(randomBinarySearchTree-
>right)) <= 1);
}

int searchAndInsert(int data, RandomBinarySearchTree* randomBinarySearchTree,
RandomBinarySearchTree* parent) {
    if (!randomBinarySearchTree) {
        randomBinarySearchTree = new RandomBinarySearchTree;
        randomBinarySearchTree->height = parent->height + 1;
        randomBinarySearchTree->left = nullptr;
        randomBinarySearchTree->right = nullptr;
        randomBinarySearchTree->parent = parent;
        randomBinarySearchTree->data = data;
        randomBinarySearchTree->count = 1;
        return randomBinarySearchTree->height;
    }
    else if (data < randomBinarySearchTree->data) return searchAndInsert(data,
randomBinarySearchTree->left, randomBinarySearchTree);
    else if (data > randomBinarySearchTree->data) return searchAndInsert(data,
randomBinarySearchTree->right, randomBinarySearchTree);
    else {
        randomBinarySearchTree->count++;
        return randomBinarySearchTree->height;
    }
}

int maxDepth(RandomBinarySearchTree* randomBinarySearchTree) {
    if (!randomBinarySearchTree) return 0;
    if                 (maxDepth(randomBinarySearchTree->left)                <=
maxDepth(randomBinarySearchTree->right)) return maxDepth(randomBinarySearchTree-
>right) + 1;
    else return maxDepth(randomBinarySearchTree->left) + 1;
}

void generateInsertElements(RandomBinarySearchTree** randomBinarySearchTree, int
countOfTrees, Results* results) {
    int countOfInsert, iteration, sumOfIteration = 0, depth;
    cout<<"Enter count of element which will be insert\n";
```

```
    do cin>>countOfInsert; while(countOfInsert <= 0);

    int randomElements[countOfInsert];

    if (mode) cout<<"Random elements: ";
    for (int i = 0; i < countOfInsert; i++) {
        randomElements[i] = rand() % MAX_NUMBER;
        if (mode) cout<<randomElements[i]<<" ";
    }
   if (mode) cout<<endl;

    for (int i = 0; i < countOfTrees; i++) {
        if (mode) {
            if (i == 0) cout<<"1st Tree:\n";
            else if (i == 1) cout<<"2nd Tree:\n";
            else if (i == 2) cout<<"3rd Tree:\n";
            else cout<<i + 1<<"th Tree:\n";
            cout<<"Maximum         height         of         this         tree:
"<<maxDepth(randomBinarySearchTree[i])<<endl;
        }

        depth = maxDepth(randomBinarySearchTree[i]);
        results->depth[i] = depth;
        for (int j = 0; j < countOfInsert; j++) {
            iteration            =            searchAndInsert(randomElements[j],
randomBinarySearchTree[i], randomBinarySearchTree[i]);
            sumOfIteration+= iteration;
            if (mode) {
                cout<<"Element: "<<randomElements[j]<<" will be insert. ";
                cout<<"Count of iterations(height): "<<iteration<<".\n";
            }
        }
        results->averageIterationsOfInsert[i]                                    =
static_cast<double>(sumOfIteration)/countOfInsert;
        sumOfIteration = 0;
        if (mode) cout<<endl;
    }

}

void generateRemoveElements(RandomBinarySearchTree** randomBinarySearchTree, int
countOfTrees, Results* results) {
```

```
    int countOfDeletion = 0, sumOfIterations = 0, tmp = 0;
    bool empty = false;
    cout<<"\nEnter count of element which will be delete\n";
    do cin>>countOfDeletion; while(countOfDeletion <= 0);

    int randomElements[countOfDeletion];

    if (mode) cout<<"Random elements: ";
    for (int i = 0; i < countOfDeletion; i++) {
        randomElements[i] = rand() % MAX_NUMBER;
        if (mode) cout<<randomElements[i]<<" ";
    }
    cout<<endl;

    for (int i = 0; i < countOfTrees; i++) {
        if (mode) {
            if (i == 0) cout<<"1st Tree:\n";
            else if (i == 1) cout<<"2nd Tree:\n";
            else if (i == 2) cout<<"3rd Tree:\n";
            else cout<<i + 1<<"th Tree:\n";
            cout<<"Maximum         height         of         this         tree:
"<<maxDepth(randomBinarySearchTree[i])<<endl;
        }
        for (int j = 0; j < countOfDeletion; j++) {
            if (mode) cout<<j<<") ";
            randomBinarySearchTree[i]   =   deleteNode(randomBinarySearchTree[i],
randomElements[j], true, sumOfIterations, empty);
            if (empty) tmp++;
            empty = false;
            if (!randomBinarySearchTree[i]) {
                break;
            }
        }
        results->averageIterationsOfDeletion[i]    =   countOfDeletion   ==   tmp?
static_cast<double>(sumOfIterations)/countOfDeletion:
static_cast<double>(sumOfIterations)/(countOfDeletion - tmp);
        sumOfIterations = 0;
        tmp = 0;
        if (mode) cout<<endl;
    }
}
```

```
void printResults(Results* results) {
    double averageInsert = 0, averageDeletion = 0, averageDepth = 0,
averageCountOfNumbers = 0;
    cout<<endl<<endl<<endl<<endl;
    cout<<"Count of trees: "<< results->countOfTrees<<".\n";
    for (int i = 0; i < results->countOfTrees; i++) {
        if (mode) {
            cout<<i<<" tree has "<<results->countOfNumbersInTree[i]<<" count of
numbers in start.\n";
            cout<<"Average count to insert is: "<<results-
>averageIterationsOfInsert[i]<<". "<<
            "Average count to deletion is: "<<results-
>averageIterationsOfDeletion[i]<<". Max height of tree: "<<
            results->depth[i]<<endl;
        }
        averageInsert+= results->averageIterationsOfInsert[i];
        averageDeletion+= results->averageIterationsOfDeletion[i];
        averageDepth+= results->depth[i];
        averageCountOfNumbers+= results->countOfNumbersInTree[i];
    }

    cout<<endl<<endl<<endl<<endl;

    cout<<"Average iteration to insert of all trees: "<<averageInsert/results-
>countOfTrees<<endl;
    cout<<"Average iteration to deletion of all trees:
"<<averageDeletion/results->countOfTrees<<endl;
    cout<<"Average n: "<<averageCountOfNumbers/results->countOfTrees<<endl;
}

void removeIdeal(RandomBinarySearchTree** randomBinarySearchTree, int
countOfTrees, Results* results, const int* countInLine) {

    int idealTrees = 0, j = 0;
    for (int i = 0; i < countOfTrees; i++) {
        if (searchIdeal(randomBinarySearchTree[i])) {
            idealTrees++;
            randomBinarySearchTree[i] = nullptr;
        }
    }
```

```
    if (idealTrees == 1) cout<<"It's "<< idealTrees<<" ideal tree. It will be
delete to remove complexity of best case\n";
    else if (idealTrees > 1) cout<<"There is "<< idealTrees<<" ideal trees. They
will be delete to remove complexity of best case\n";
    // array without ideal trees
    auto**  treesWithoutIdeal  =  new  RandomBinarySearchTree*[countOfTrees  -
idealTrees];

    results->countOfNumbersInTree = new int[countOfTrees - idealTrees];
    results->averageIterationsOfInsert = new double [countOfTrees - idealTrees];
    results->averageIterationsOfDeletion   =   new   double[countOfTrees   -
idealTrees];
    results->depth = new int[countOfTrees - idealTrees];

    for (int i = 0 ; i < countOfTrees; i++) {
        if (randomBinarySearchTree[i]) {
            treesWithoutIdeal[j] = new RandomBinarySearchTree;
            treesWithoutIdeal[j] = randomBinarySearchTree[i];
            results->countOfNumbersInTree[j] = countInLine[i];
            j++;
        }
    }

    results->countOfTrees = countOfTrees - idealTrees;

    //insert in trees
    generateInsertElements(treesWithoutIdeal,countOfTrees     -     idealTrees,
results);

    //remove from trees
    generateRemoveElements(treesWithoutIdeal,   countOfTrees   -   idealTrees,
results);



    printResults(results);
}

void makeTrees() {
    //results
    auto* results = new Results;
    // count of trees
    int countOfTrees= rand() % MAX_COUNT_OF_TREES + 10;
```

```
    //array of count of integer in lines
    int countInLine[countOfTrees];
    cout<<"Built "<<countOfTrees<<" trees\n";

    //write count of numbers in trees
    for (int i = 0; i < countOfTrees; i++) {
            countInLine[i] = rand() % MAX_NUMBER + 10;
    }
    // array of array of lines
    int **arrayOfLines = new int*[countOfTrees];
    for (int i = 0; i < countOfTrees; i++) arrayOfLines[i] = new
int[countInLine[i]];
    // create and print lines
    for (int i = 0; i < countOfTrees; i++) {
        for (int j = 0; j < countInLine[i]; j++) {
            arrayOfLines[i][j] = (rand() % MAX_NUMBER);
        }
    }
    cout<<endl;
    //create trees
    auto** arrayOfTrees = new RandomBinarySearchTree*[countOfTrees];
    for (int i = 0; i < countOfTrees; i++) arrayOfTrees[i] = new
RandomBinarySearchTree;

    // fill trees
    for (int i = 0; i < countOfTrees; i++) {
        enterTree(arrayOfLines[i], countInLine[i], arrayOfTrees[i]);
    }

    removeIdeal(arrayOfTrees, countOfTrees, results, countInLine);
}

int main() {
    char modes;
    srand(time(nullptr));
    cout<<"Press 'q' to show intermediate results, otherwise press any
button\n";
    cin>>modes;
    mode = modes == 'q';
    makeTrees();
    return 0;
```

```
}
```