

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	5
1.	Описание программы	6
1.1.	Используемые структуры	6
1.2.	Используемые функции	6
2.	Генерация деревьев	9
3.	Вставка элемента	10
4.	Удаление элемента	11
5.	Тестирование	13
6.	Исследование	14
7.	Заключение	18
8.	Список использованных источников	19
9.	Приложение А. исходный код программы	20

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 реализации случайного бинарного дерева поиска была создана структура RandomBinarySearchTree, а также несколько функций, 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 поддереву случайного БДП

right - правое поддереву случайного БДП

parent - родитель узла

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 нового узла.

Int data - элемент дерева

RandomBinarySearchTree* rbsr - случайное БДП

randomBinarySearchTree* parent - родитель нового узла

Возвращает созданный узел

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

RandomBinarySearchTree* root - случайное БДП

Возвращает найденный узел

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

Возвращает случайное БДП с удаленным узлом

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 узла

RandomBinarySearchTree* rbst - случайное БДП

Возвращает высоту данного узла

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

Int data - элемент, который необходимо вставить

RandomBinarySearchTree* rbsr - само случайное БДП

RandomBinarySearchTree* parent - родитель

Возвращает высоту вставляемого узла

int maxDepth(RandomBinarySearchTree* randomBinarySearchTree) -

функция поиска глубины дерева.

RandomBinarySearchTree* randomBinarySearchTree - случайное БДП

Возвращает глубину случайного БДП

void generateInsertElements(RandomBinarySearchTree**

randomBinarySearchTree, int countOfTrees, Results* results) - функция

генерации вставляемых элементов.

RandomBinarySearchTree** randomBinarySearchTree - массив случайных БДП

Int countOfTrees - количество деревьев

Results* results - структура результатов

void generateRemoveElements(RandomBinarySearchTree**

randomBinarySearchTree, int countOfTrees, Results* results) - функция

генерации удаляемых элементов.

RandomBinarySearchTree** randomBinarySearchTree - массив случайных БДП

Int countOfTrees - количество деревьев

Results* results - структура результатов

void printResults(Results* results) - функция вывода результатов

Results* results - 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

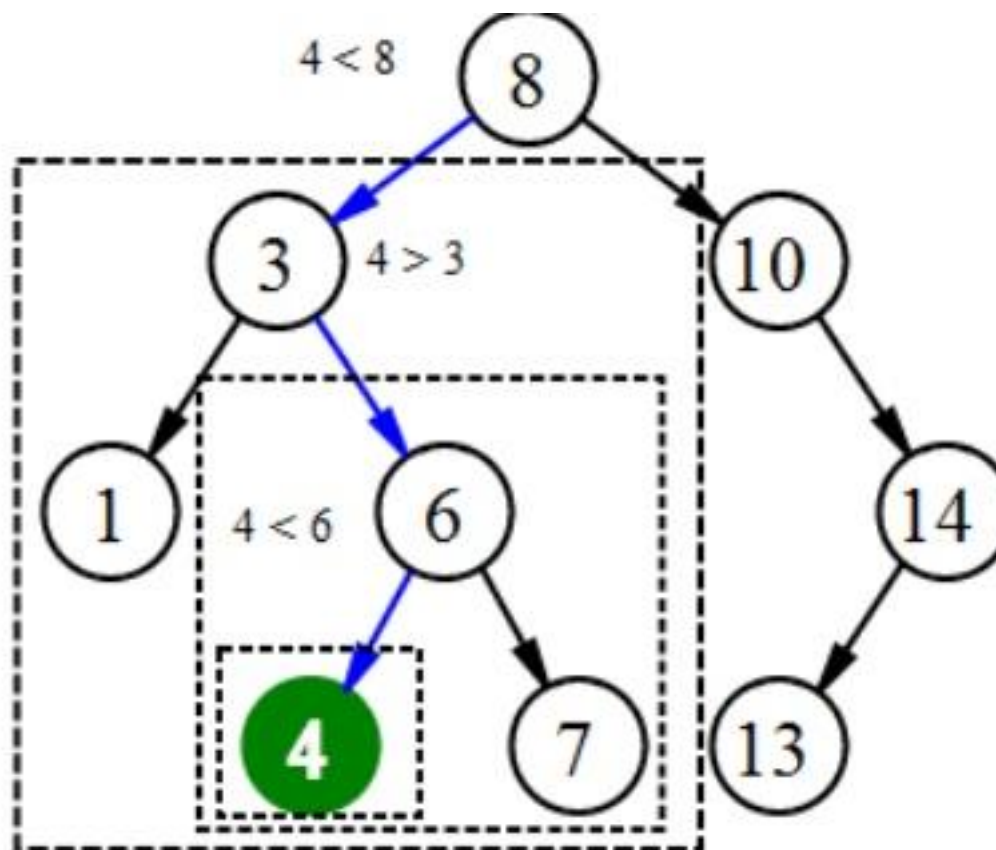
2. ГЕНЕРАЦИЯ ДЕРЕВЬЕВ

Программа генерирует два случайных чисел, которые обозначают количество случайных БДП для среднего и худшего случая. После этого она генерирует массив элементов каждого случайного БДП. (Длина массива также определяется случайно для каждого дерева). Затем, на основе этих данных, алгоритм создает случайные БДП (для среднего и худшего случая). Сгенерированные массивы деревьев отправляются в следующую функцию для вставки элементов.

3. ВСТАВКА ЭЛЕМЕНТА

Сначала программа просит ввести количество вставляемых элементов. После этого она генерирует набор случайных чисел, которые нужно будет вставить в дерево. Поиск и вставку выполняет рекурсивная функция `searchAndInsert`. При первом вызове ей подается корень дерева. Проверяется этот корень. Если вставляемый элемент меньше данного корня, 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.

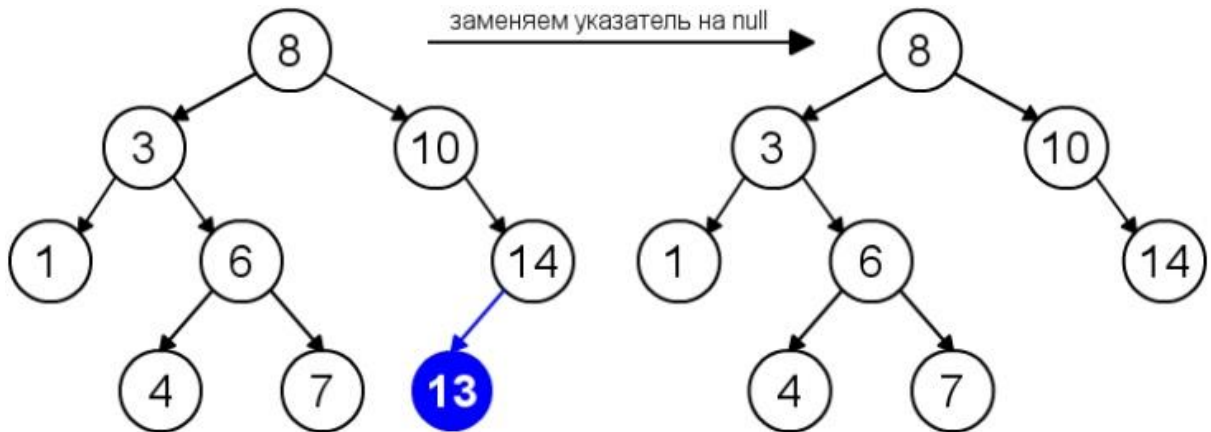
Пример вставки элемента 4 в дерево:



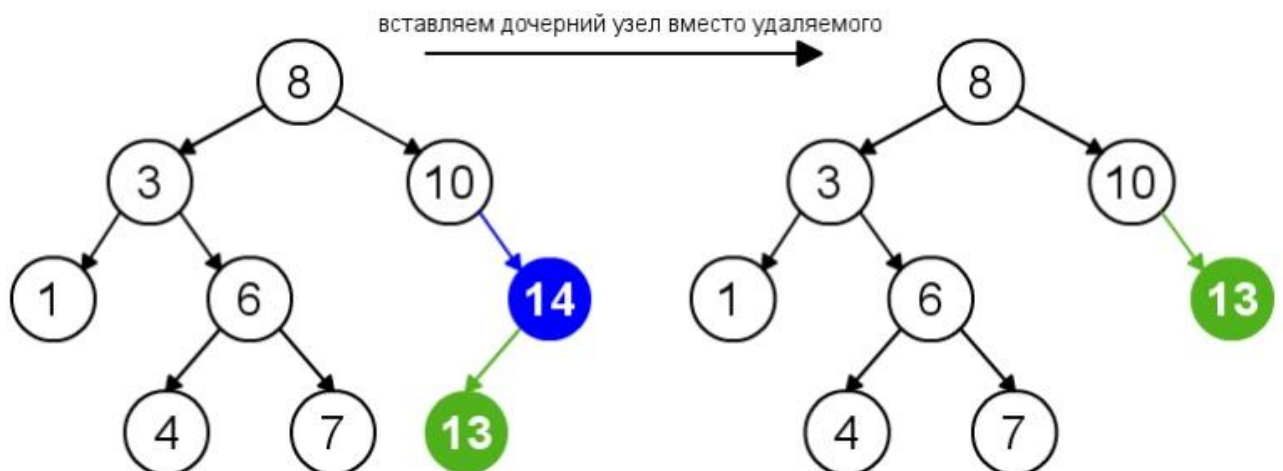
4. УДАЛЕНИЕ ЭЛЕМЕНТА

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;

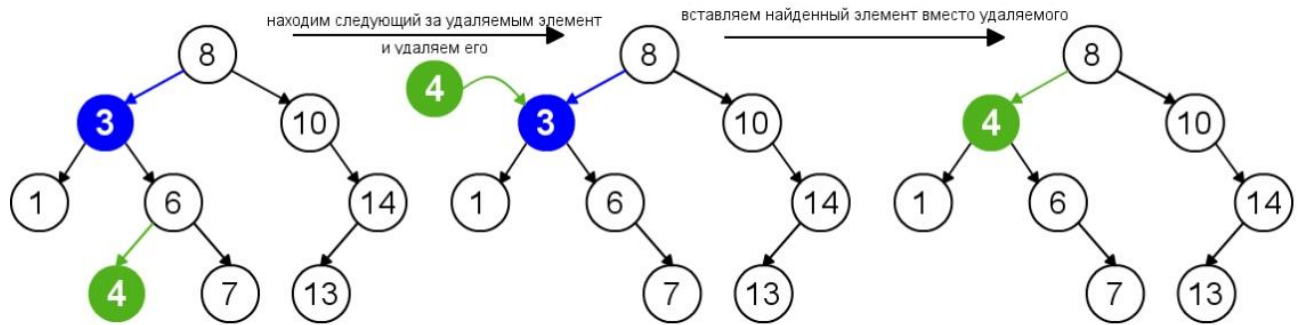


- удаляемый элемент имеет одно поддерево - если удаляемый элемент имеет только левое или только правое поддерево, то на место этого элемента ставится его поддерево



- 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
n
Built 11 trees in average case
Built 16 trees in worst case

Enter count of element which will be insert
1000
Enter count of element which will be insert
300

Enter count of element which will be delete
1900

Enter count of element which will be delete
1000

Average Case:

Count of trees: 11.

Average iteration to insert of all trees: 11.2728
Average iteration to deletion of all trees: 10.9538
Average n: 549.909
Worst Case:

Count of trees: 16.

Average iteration to insert of all trees: 21.6129
Average iteration to deletion of all trees: 11.3176
Average n: 21.8125
```

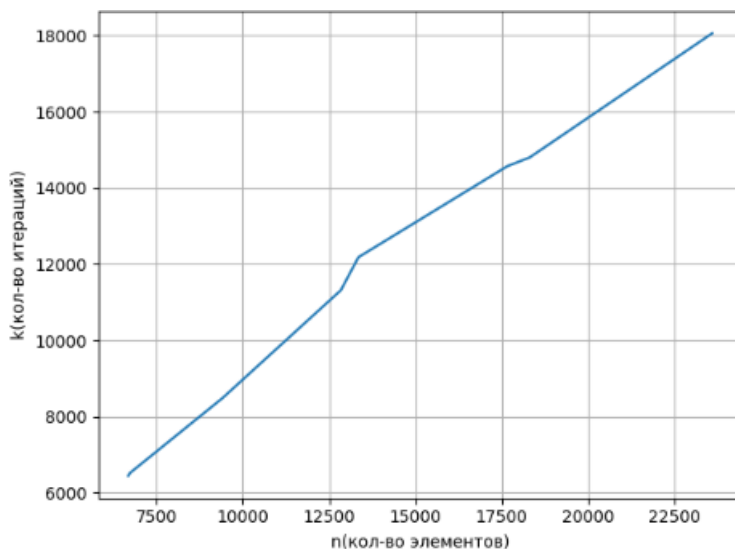
В результате работы программа выдает среднее количество итераций для вставки и удаления(в среднем и худшем случае), а также среднее количество элементов в дереве.

6. ИССЛЕДОВАНИЕ

6.1.1 Тестирование среднего количества итераций для вставки элемента в бинарное дерево(худший случай).

Среднее количество элементов в случайном БДП	Среднее количество итераций для вставки элементов
6729.4	6524.08
6674.94	6444.36
12839.4	11317.9
9407.57	8486.46
13348.2	12183.2
17661.6	14571.5
18315.4	14803.6
23606.3	18061.5

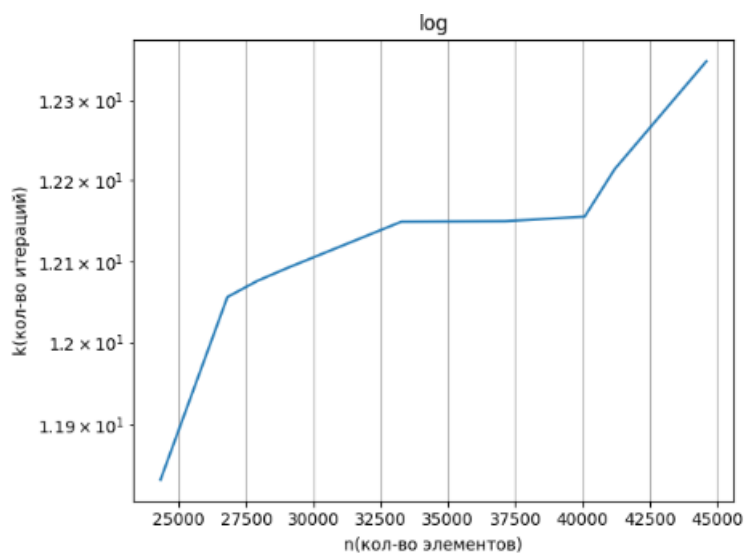
График зависимости итераций для вставки от количества элементов в случайном БДП(худший случай)



6.1.2 Тестирование среднего количества итераций для вставки элемента в бинарное дерево(средний случай).

Среднее количество элементов в случайном БДП	Среднее количество итераций для вставки элементов
26802.4	12.0559
24314.1	11.8331
29040.4	12.9018
27919.2	12.0758
33270.5	12.1489
40083.9	12.1552
37125.2	12.1495
41198.4	12.2141
44611.7	12.3494

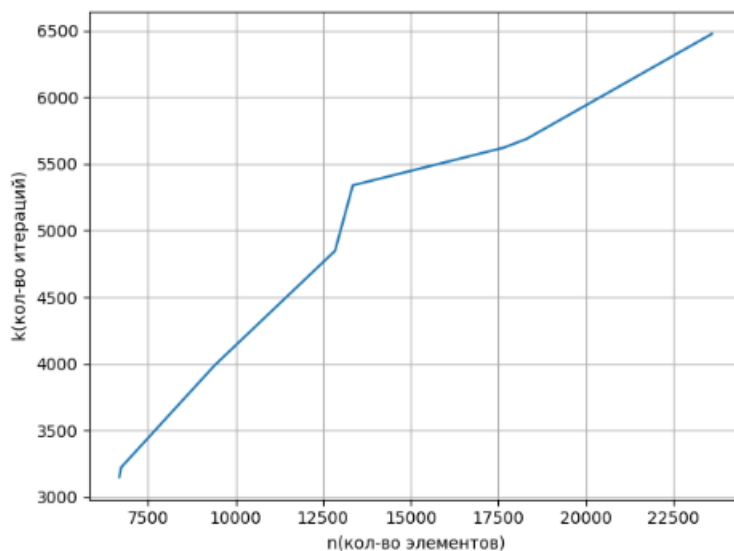
График зависимости итераций для вставки от количества элементов в случайном БДП(средний случай)



6.2.1 Тестирование среднего количества итераций для удаления элемента из бинарное дерево(худший случай).

Среднее количество элементов в случайном БДП	Среднее количество итераций для удаления элементов
6729.4	3222.69
6674.94	3148.76
12839.4	4847.26
9407.57	3990.19
13348.2	5338.8
17661.6	5621.56
18315.4	5687.52
23606.3	6475.02

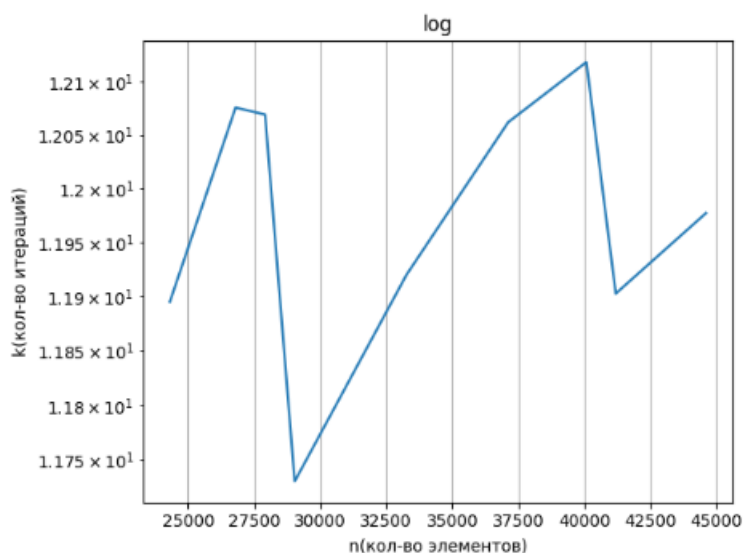
График зависимости итераций для удаления от количества элементов в случайном БДП(худший случай)



6.2.2 Тестирование среднего количества итераций для удаления элемента из бинарное дерево(средний случай).

Среднее количество элементов в случайном БДП	Среднее количество итераций для удаления элементов
26802.4	12.0756
24314.1	11.895
29040.4	11.7304
27919.2	12.0691
33270.5	11.9198
40083.9	12.1183
37125.2	12.0619
41198.4	11.9026
44611.7	11.9771

График зависимости итераций для удаления от количества элементов в случайном БДП(средний случай)



На графиках видно, что средняя количество итераций по вставке и удалению в случайном БДП в среднем случае $O(\log(n))$ 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. Экспериментальные данные, полученные в ходе работы, совпали с теоретическими. В среднем операция вставки и удаления выполняются за $O(\log(n))$, а в худшем случае 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\).](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(NULL));
    cout<<"Press 'q' to show intermediate results, otherwise press any
button\n";
    cin>>modes;
    mode = modes == 'q';
    makeTrees();
    return 0;
}

```

}