

**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: Работа со случайными бинарными деревьями поиска.**

Студент гр. 9382

\_\_\_\_\_

Герасев Г.

Преподаватель

\_\_\_\_\_

Фирсов М.А.

Санкт-Петербург

2020

**ЗАДАНИЕ**  
**НА КУРСОВУЮ РАБОТУ (КУРСОВОЙ ПРОЕКТ)**

Студент Герасев Г

Группа 9382

Тема работы: Случайные БДП – вставка и исключение. Вставка в корень БДП. Текущий контроль.

Исходные данные: количество элементов в дереве для дальнейшего создания случайного дерева поиска студентами, и удаление из получившегося дерева случайно выбранного узла.

Содержание пояснительной записки: «Содержание», «Введение», «Основные теоретические сведения», «Описание алгоритмов», «Описание структур данных и используемых функций», «Текущий контроль», «Тестирование», «Заключение», «Исходный код программы»

Предполагаемый объем пояснительной записки:

Не менее 20 страниц.

Дата выдачи задания: 31.10.2020

Дата сдачи реферата: 26.12.2020

Дата защиты реферата: 26.12.2020

Студент

\_\_\_\_\_

Герасев Г.

Преподаватель

\_\_\_\_\_

Фирсов М.А.



## **АННОТАЦИЯ**

В данной курсовой работе была реализована программа, которая выполняет создание случайного БДП, удаление элемента из него и вывод дерева в удобном для прочтения виде. Текущий контроль включает в себя вывод заданий по созданию дерева и удалению элемента из него. Реализованную программу можно использовать в обучении для проверки знаний студентов.

## **SUMMARY**

In this course work, a program was implemented that creates a random BDP, deletes an element from it and displays a tree in an easy-to-read form. Current control includes the output of tasks for creating a tree and deleting an element from it. The implemented program can be used in teaching to test the knowledge of students.

## СОДЕРЖАНИЕ

ВВЕДЕНИЕ.....	6
Основные теоретические положения.....	7
Описание алгоритмов.....	8
Описание структур данных и используемых функций.....	10
Описание интерфейса пользователя.....	12
ТЕКУЩИЙ КОНТРОЛЬ.....	13
Тестирование.....	13
Заключение.....	13
ПРИЛОЖЕНИЕ А.....	15
ИСХОДНЫЙ КОД ПРОГРАММЫ.....	15

## ВВЕДЕНИЕ

В данной курсовой работе были реализованы создание случайного БДП и удаления элемента из него и реализация текущего контроля по данным темам.

Цель:

Целью данной работы является реализация работы создания случайного БДП и удаления элемента из него и реализация текущего контроля по данным темам.

Для реализации данной цели нужно решить следующие задачи:

- Собрать теоретические сведения по изучаемым алгоритмам, структуре данных и анализируемым функциям
- На основе теоретических данных написать программу на языке C++, реализующую заданную структуру данных и необходимые алгоритмы
- Реализовать текущий контроль студентов, для определения их понимания данной темы.

## Основные теоретические положения

Бинарное дерево поиска, это бинарное дерево, значения в котором лежат так, что для любого узла верно, что все значения в левом поддереве будут меньше корневого узла, а все значения в правом – больше.

Такое дерево будет называться сбалансированным, если у всех его листьев одинаковая высота. Чем ближе разница высот к нулю, тем сбалансированнее дерево. Чем сбалансированнее дерево, тем быстрее будет проходить по нему поиск.

Случайное бинарное дерево поиска, это такое дерево поиска, что при его создания во время последовательного добавления в него элементов, элемент имеет шанс стать корневым или «листовым» (т.е. лежащем в поддереве) обратно пропорционален количеству элементов в дереве. Таким образом можно создавать сбалансированные деревья поиска, с поиском  $\sim \ln(2n)$ , что крайне хороший результат, учитывая то, что алгоритм не требует обработки вводимых значений (на пример сортировки).

Вставка элемента в дерево – добавление элемента в дерево таким образом, что его свойства не нарушаются. На пример в БДП вставка элемента не меняет свойство дерева быть деревом поиска.

Вставка в корень – такая вставка в дерево, что значение в корне меняется на переданное, и все дерево изменяется так, чтобы остаться деревом поиска и добавить замененное значение в само дерево. Благодаря алгоритму, описанному в соответствующем разделе, это делается достаточно эффективно.

## Описание алгоритмов

Для реализации случайной вставки элемента в дерево нам потребуются также:

Поворот дерева – локальное преобразование дерева, поднимающее один из корней выше, и опускающее другое. На пример вот один из поворотов дерева –  $(A, (B, C)) \rightarrow ((A, B), C)$  – A, где A, B и C – поддеревья, которые остаются неизменными. Как можно заметить данное преобразование не изменяет свойства дерева поиска.

Алгоритм вставки элемента в корень дерева –

Для этого выбирается поддерево, в котором должен оказаться данный элемент, путем сравнения его с корневым, после чего алгоритм вызывается рекурсивно для этого поддерева. Далее производится поворот дерева, левый или правый в зависимости от поддерева, в котором оказалось значение, чтобы поднять данное значение на 1 уровень выше.

Если поддерево оказалось пустым, то туда записывается значение. Таким образом вставка в корень достигается  $\log(n)$  поворотами дерева, что является локальным преобразованием, и в результате данный алгоритм работает эффективно.

Алгоритм случайной вставки элемента в бинарное дерево –

Рассматривается количество элементов в дереве (обычно это значение, которое хранит узел). После чего с шансом  $1/(n+1)$ , где n – количество элементов в дереве, производится вставка в узел, иначе алгоритм рекурсивно вызывается у соответствующего поддерева. Если поддерево пустое, то просто создается узел с данным элементом. Таким образом каждый элемент из ввода имеет равный шанс оказаться корневым, что приводит к созданию довольно сбалансированных деревьев.



Алгоритм объединения поддеревьев бинарного дерева –

Данный алгоритм потребует для удаления элемента из бинарного дерева. Его работа крайне проста – из левого и правого поддерева пропорционально количеству элементов в них выбирается узел который станет корневым, а второе дерево рекурсивно передается в качестве одного из аргументов с соответствующим поддеревом выбранного дерева. Таким образом два дерева объединяются в одно дерево, которое тоже будет сбалансировано также, как и остальное дерево методом случайной вставки.

Алгоритм удаления значения из дерева –

Дерево не гарантирует отсутствия одинаковых элементов, по этому удаляется первый найденный. Производится простой поиск по бинарному дереву, и если найден узел с требуемым значением, то его поддерева объединяются и полученный корень объявляется новым корневым узлом, а старый корневой узел удаляется.

## Описание структур данных и используемых функций

Для реализации случайных БДП был создан класс `BinarySearchTree`. В нем определены следующие поля и методы:

- `int data` // Данные, хранящиеся в узле
- `Pointers pointers` // Структура с указателями на левый и правый узел
- `unsigned int quantityOfNodes` // Количество узлов в дереве
- `BinarySearchTree(int data = 0);` // Конструктор
- `BinarySearchTree(const BinarySearchTree & binarySearchTree);` // Оператор копирования
- `~BinarySearchTree();` // Деструктор
- `void draw(string buffer = "", bool isLast = true);` // Метод рисующий дерево в стандартный поток вывода
- `void drawInString(string* res, string buffer = "", bool isLast = true);` // Метод рисующий дерево в строку
- `int getQuantityOfNodes();` // Возвращает количество элементов в дереве
- `void updateQuantityOfNodes();` // Обновляет значения поля количества элементов в дереве, путем рекурсивного прохода
- `BinarySearchTree* rotateLeft();`
- `BinarySearchTree* rotateRight();` // Левый и правый поворот узла
- `BinarySearchTree* insertInRoot(int data);` // Вставка в узел дерева
- `BinarySearchTree* insert(int data);` // Случайная вставка в узел дерева
- `BinarySearchTree* deleteFirst(int data);` // Удаление первого элемента с переданным значением

Далее перечислены функции

- `BinarySearchTree* join(BinarySearchTree* smallerTree, BinarySearchTree* biggerTree)` // Объединение поддеревьев
- `int* giveRandMasWithLength(unsigned int n, unsigned int upperBoundary)` // Создание массива случайных чисел с данной длиной и верхней границе
- `BinarySearchTree* giveTreeWithLength(int* mas, unsigned int n)` // Создание дерева по полученному массиву

- `void makeTest(unsigned int n, unsigned int upBound) // Создание теста по переданным длине и верхней границе (в файл)`

## **Описание интерфейса пользователя**

В начале работы программа узнает у пользователя, количество элементов в дереве, а также верхнюю границу. После чего создает тест, состоящий из двух вопросов – создание дерева по данным значениям (случайно сгенерированных), а также удаление из получившегося дерева требуемого элемента из дерева.

Тест записывается в файл RandomTreeTest.txt с примерами ответов, т. к. очевидно на данные вопросы можно дать много верных ответов.

## ТЕКУЩИЙ КОНТРОЛЬ

В текущем контроле задаются заранее подготовленные вопросы, которые относятся к переданным программе ограничениям на деревья. В программе проверяется корректность ограничений, и при неправильно переданных значениях программа не выдает тест.

### Тестирование

Примеры работы программы:

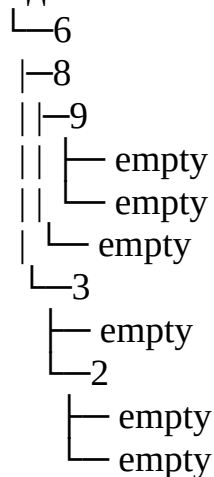
Переданные значения 5 и 10. К сожалению в данном отчете возникают сложности с прочтением вывода в виду форматирования, но при реальном выводе программы таких проблем нет.

Задание №1:

Создайте случайное дерево поиска из следующих чисел:

6 3 8 2 9

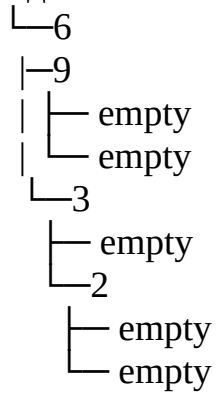
Один из возможных ответов на задание №1:



Задание №2:

Удалите из получившегося дерева элемент 8

Один из возможных ответов на задание №2:



### **Заключение**

Были изучены алгоритмы случайной вставки элемента в дерево и удаление элемента из него. Была разработана программа для построения бинарных деревьев на языке C++. Был представлен текущий контроль.



## ПРИЛОЖЕНИЕ А

### ИСХОДНЫЙ КОД ПРОГРАММЫ

#### **TestGenerator.h:**

```
#include <iostream>
#include <fstream>
#include <string.h>
#include <cstdlib>

using namespace std;

class BinarySearchTree
{
    struct Pointers
    {
        BinarySearchTree* left = nullptr;
        BinarySearchTree* right = nullptr;
    };

public:
    int data; // in our case the node data == node key, but it's easy to change
    Pointers pointers;
    unsigned int quantityOfNodes; // For random bin search tree only

    BinarySearchTree(int data = 0);
    BinarySearchTree(const BinarySearchTree & binarySearchTree); // Copy operator
    ~BinarySearchTree();

    void draw(string buffer = "", bool isLast = true);
    void drawInString(string* res, string buffer = "", bool isLast = true);

    int getQuantityOfNodes();
    void updateQuantityOfNodes();

    BinarySearchTree* rotateLeft();
    BinarySearchTree* rotateRight();

    BinarySearchTree* insertInRoot(int data); // Return root pointer
    BinarySearchTree* insert(int data);

    BinarySearchTree* deleteFirst(int data);
```

#### **TestGenerator.cpp:**

```
#include "TestGenerator.h"

BinarySearchTree::BinarySearchTree(int inputData)
{
    data = inputData;
    quantityOfNodes = 1;
    pointers.left = nullptr;
    pointers.right = nullptr;
}

BinarySearchTree::BinarySearchTree(const BinarySearchTree & binarySearchTree) //
Copy operator
{
    data = binarySearchTree.data;

    if (binarySearchTree.pointers.left != nullptr)
        pointers.left = new BinarySearchTree(*binarySearchTree.pointers.left);
```



```

        if (binarySearchTree.pointers.right != nullptr)
            pointers.right = new BinarySearchTree(*binarySearchTree.pointers.right);
    }

BinarySearchTree::~BinarySearchTree()
{
    if (pointers.left != nullptr)
        delete pointers.left;
    if (pointers.right != nullptr)
        delete pointers.right;
}

int BinarySearchTree::getQuantityOfNodes()
{
    return quantityOfNodes;
}

void BinarySearchTree::updateQuantityOfNodes()
{
    unsigned int quantityOfNodesLeft = 0;
    unsigned int quantityOfNodesRight = 0;
    if (pointers.left != nullptr)
        quantityOfNodesLeft = pointers.left->getQuantityOfNodes();

    if (pointers.right != nullptr)
        quantityOfNodesRight = pointers.right->getQuantityOfNodes();

    quantityOfNodes = 1 + quantityOfNodesLeft + quantityOfNodesRight;
}

BinarySearchTree* BinarySearchTree::rotateLeft() // (A, (B, C)) -> ((A, B), C))
{
    BinarySearchTree* right = pointers.right;
    if (right == nullptr)
        return this;
    pointers.right = right->pointers.left;
    right->pointers.left = this;

    right->quantityOfNodes = quantityOfNodes;
    updateQuantityOfNodes();
    return right;
}

BinarySearchTree* BinarySearchTree::rotateRight() // ((A, B), C)) -> (A, (B, C))
{
    BinarySearchTree* left = pointers.left;
    if (left == nullptr)
        return this;
    pointers.left = left->pointers.right;
    left->pointers.right = this;

    left->quantityOfNodes = quantityOfNodes;
    updateQuantityOfNodes();
    return left;
}

BinarySearchTree* BinarySearchTree::insertInRoot(int inputData)
{
    if (inputData < data)
    {
        if (pointers.left == nullptr)
        {
            pointers.left = new BinarySearchTree(inputData);
            return this;
        }
    }
}

```

```

        }
        else
        {
            pointers.left = pointers.left->insertInRoot(inputData);
            return rotateRight();
        }
    }

    if (pointers.right == nullptr)
    {
        pointers.right = new BinarySearchTree(inputData);
        return this;
    }
    else
    {
        pointers.right = pointers.right->insertInRoot(inputData);
        return rotateLeft();
    }
}

BinarySearchTree* BinarySearchTree::insert(int inputData)
{
    int randNumber = rand();
    srand(randNumber);

    bool stopHere = false;
    if (randNumber%(quantityOfNodes + 1) == 0)
        stopHere = true;

    if (stopHere)
    {
        BinarySearchTree* res = insertInRoot(inputData);
        updateQuantityOfNodes();
        return res;
    }

    if (inputData < data)
    {
        if (pointers.left == nullptr)
            pointers.left = new BinarySearchTree(inputData);
        else
            pointers.left = pointers.left->insert(inputData);
    }
    else
    {
        if (pointers.right == nullptr)
            pointers.right = new BinarySearchTree(inputData);
        else
            pointers.right = pointers.right->insert(inputData);
    }

    updateQuantityOfNodes();
    return this;
}

BinarySearchTree* join(BinarySearchTree* smallerTree, BinarySearchTree* biggerTree)
{
    if (smallerTree == nullptr)
        return biggerTree;
    if (biggerTree == nullptr)
        return smallerTree;

    int randNumber = rand();
    srand(randNumber);
    bool goSmaller = false;

```

```

        if (randNumber%(smallerTree->getQuantityOfNodes() + biggerTree-
>getQuantityOfNodes()) < smallerTree->getQuantityOfNodes())
            goSmaller = true;

        if (goSmaller) {
            smallerTree->pointers.right = join(smallerTree->pointers.right, biggerTree);
            smallerTree->updateQuantityOfNodes();
            return smallerTree;
        } else {
            biggerTree->pointers.left = join(smallerTree, biggerTree->pointers.left);
            biggerTree->updateQuantityOfNodes();
            return biggerTree;
        }
    }
}

```

```

BinarySearchTree* BinarySearchTree::deleteFirst(int inputData)
{
    if (inputData == data)
    {
        BinarySearchTree* res = join(pointers.left, pointers.right);
        return res;
    }

    if (inputData < data)
    {
        if (pointers.left != nullptr)
            pointers.left = pointers.left->deleteFirst(inputData);
    }

    else
    {
        if (pointers.right != nullptr)
            pointers.right = pointers.right->deleteFirst(inputData);
    }

    return this;
}

```

```

void BinarySearchTree::draw(string buffer, bool isLast)
{
    string branch = "|";
    string pipe = "|";
    string end = "L";
    string dash = "-";

    if (isLast)
    {
        cout << buffer << end << dash << data << '\n';
        buffer += " ";
    }

    else
    {
        cout << buffer << pipe << dash << data << '\n';
        buffer += pipe + " ";
    }

    if (pointers.right != nullptr)
        pointers.right->draw(buffer, false);
    else
        cout << buffer << branch << dash << " empty\n";

    if (pointers.left != nullptr)
        pointers.left->draw(buffer, true);
}

```

```

        else
            cout << buffer << end << dash << " empty\n";
    }

void BinarySearchTree::drawInString(string* res, string buffer, bool isLast)
{
    string branch = "|";
    string pipe = "|";
    string end = "L";
    string dash = "-";

    if (isLast)
    {
        *res += buffer + end + dash + to_string(data) + '\n';
        buffer += " ";
    }

    else
    {
        *res += buffer + pipe + dash + to_string(data) + '\n';
        buffer += pipe + " ";
    }

    if (pointers.right != nullptr)
        pointers.right->drawInString(res, buffer, false);
    else
        *res += buffer + branch + dash + " empty\n";

    if (pointers.left != nullptr)
        pointers.left->drawInString(res, buffer, true);
    else
        *res += buffer + end + dash + " empty\n";
}

bool isInMasLength(unsigned int n, int x, int* mas)
{
    for (int i=0; i<n; i++)
    {
        if (mas[i] == x)
        {
            return true;
        }
    }
    return false;
}

int* giveRandMasWithLength(unsigned int n, unsigned int upperBoundary)
{
    auto res = new int[n];
    int r;
    for (int i=0; i<n; i++)
    {
        do
        {
            r = rand();
            srand(r);
        } while(isInMasLength(i+1, r%upperBoundary, res));
        res[i] = r%upperBoundary;
    }
    return res;
}

BinarySearchTree* giveTreeWithLength(int* mas, unsigned int n)
{

```

```

        auto result = new BinarySearchTree(mas[0]);
        if (n == 0 || n == 1)
            return result;

        for (int i=1; i<n; i++)
            result->insert(mas[i]);

        return result;
    }

    void makeTest(unsigned int n, unsigned int upBound)
    {
        string path = "./RandomTreeTest.txt";
        ofstream fout;
        fout.open(path);
        int* theMas = giveRandMasWithLength(n, upBound);
        fout << "\nЗадание №1:\nСоздайте случайное дерево поиска из следующих чисел:\n";
        for (int i=0; i<n; i++)
            fout << theMas[i] << ' ';

        fout << "\nОдин из возможных ответов на задание №1:\n";

        auto res = giveTreeWithLength(theMas, n);
        string treeString;
        res->drawInString(&treeString);
        fout << treeString;
        treeString = "";

        int r = rand()%n;
        fout << "\n\n\nЗадание №2:\nУдалите из получившегося дерева элемент " <<
theMas[r] << "\n\n\n";
        fout << "\nОдин из возможных ответов на задание №2:\n";
        res->deleteFirst(theMas[r]);
        res->drawInString(&treeString);
        fout << treeString;
    }

    int main(int argc, char *argv[])
    {
        srand(time(0));
        cout << "Please input the length of tree and upper boundary for test ";
        unsigned int len;
        unsigned int upBound;
        cin >> len;
        cin >> upBound;
        cout << "\nThe length is " << len << '\n';
        cout << "\nThe upper boundary " << upBound << '\n';

        if (0 < len <= upBound)
        {
            makeTest(len, upBound);
            cout << "\nThe test has been made\n";
        }
        else
        {
            cout << "Wrong input\n";
        }
        return 0;
    }
}

```