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

ОТЧЕТ

по лабораторной работе №5

по дисциплине «Алгоритмы и структуры данных»

Тема: Случайное БДП с рандомизацией

Студент гр. 9304 _____ Афанасьев А.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург

2020

Цель работы.

Реализовать случайное бинарное дерево поиска с рандомизацией, а также соответствующие методы для него.

Постановка задачи.

Вариант *10.*

По заданной последовательности элементов *Elem* построить случайное бинарное дерево поиска с рандомизацией. Для построенной структуры данных проверить, входит ли в неё элемент *e* типа *Elem*, и если входит, то в скольких экземплярах. Добавить элемент *e* в структуру данных.

Выполнение работы.

Программа на вход ожидает строку для конструирования дерева, количество элементов и сами эти элементы для их обработки в соответствии с заданием. Пример вызова программы представлен на рисунке 1.

```
strx@strxpc:~/gitreps/ADS-9304/Afanasyev/lab5$ ./lab5 "324io43y5u34" 3 2 3 4
1 3 3
```

Рисунок 1 - Пример вызова программы

Класс BinaryTreeNode.

В нем хранится шаблонное поле *obj*, *weak_ptr* на родителя *parent* и *shared_ptr* на левого и правого потомков (*left* и *right*). Сам класс является шаблонным.

Класс Random*BinarySearchTree*.

Шаблонный класс случайного бинарного дерева поиска. Имеет поле *shared_ptr* на голову дерева *head*. Имеет конструктор от строки и метод *taskFindAndInsert(),* который выполняет логику в соответствии поставленной задаче. Также есть перегруженные операторы копирования, перемещения, конструкторы копирования, перемещения для любого подставляемого типа.

Реализованы методы префиксного, инфиксного и постфиксного обходов, метод проверки на пустоту *empty()*, метод очистки метод вставки элемента *insert()*, метод вычисления высоты *height(),* метод вычисления количества элементов *size()*, метод поиска количества вхождений определенного элемента *find(),* метод удаления элемента *remove()* и оператор вывода в поток. Также были реализованы вспомогательные приватные методы, о некоторых из которых позже.

leftTreeRotate() и *rightTreeRotate()* поворачивают дерево в определенном узле влево или вправо, соответственно. *insert()* работает как обычный алгоритм вставки в бинарного дерево поиска, но имеет вероятность вставить элемент в узел выше его обычного места, используя генератор случайных чисел и приватный метод *insertAsRoot()*. *insertAsRoot()* продолжает вставлять элемент в то место, куда его бы вставил обычный алгоритм вставки, но при этом он еще запоминает направления, по которым он двигался во время вставки, чтобы потом повернуть узлы сверху вниз в обратном направлении, таким образом вставляемый элемент переместится в нужный узел. Метод *remove()* находит удаляемый элемент и на его место ставит объединенное поддерево из левого и правого поддеревьев удаляемого элемента с помощью рекурсивного метода *merge()*, который использует генератор случайных чисел.

Тестирование.

Программу можно собрать командой *make*, после этого создается исполняемый файл *lab5.* Его можно запустить, передав в него следующие данные: строка, от которой построится случайное БДП, число элементов, с которыми нужно выполнить действия из задания, и эти элементы. Также можно запустить тестирующий скрипт *testScript.py,* конфигурационный файл которого лежит в папке с исполняемым файлом. В конфигурационном файле можно настроить многие параметры, включая количество тестов и директорию, в которой они находятся. В тестовом файле должна находиться только лишь одна строка – сам тест. Программа возвращает сообщение об синтаксической

ошибке ввода, если такая была, либо ответ. Тестирующий скрипт выводит на экран поданную строку, результат работы программы, правильный ответ и *success* или *fail* в зависимости от совпадения того, что вернула программа, и правильного ответа. Пример его работы можно посмотреть на рисунке 2. А в таблице 1 можно посмотреть примеры строк-тестов.

```
strx@strxpc:~/gitreps/ADS-9304/Afanasyev/lab5$ python testScript.py
Make sure that this script is in the same directory as the program execute file.

test0:
Input: "324io43y5u34" 1 3
CorrectAnswer: 3
Answer: 3
Result: success

test1:
Input: "324" 3 3 2 4
CorrectAnswer: 1 1 1
Answer: 1 1 1
Result: success

Total: Successes: 2. Fails: 0
```

Рисунок 2 - Пример вызова скрипта

Таблица 1. Примеры входных и выходных данных

№	Входные данные	Выходные данные
1	"324io43y5u34" 1 3	3
2	"324" 3 3 2 4	1 1 1
3	"8765456878" 8	incorrect input
4	"oifnfdnfd" 1 d	2
5	"324io43y5u34" 3 3 i o	3 1 1
6	"adsfdsakljf" 2 f a	2 2
7	"8945putr;jds" 2 2 p	0 1
8	"kjsadkfjhdsa" 5 ' ' d s t d	0 2 2 0 3
9	"iodf;fdk;fdk" 1 k	2
10	"nkfkjv kfdfdk" 2 j j	1 2

Выводы.

В ходе выполнения лабораторной работы было реализовано случайное БДП с рандомизацией. Выяснили, что алгоритмы вставки, поиска и удаления работают реже за o(n), чем в обычном БДП, но худший случай все еще имеет место быть. Для наилучшей скорости лучше использовать АВЛ-дерево, что предлагает o(logn).

ПРИЛОЖЕНИЕ А

main.cpp

```
#include "../libs/RandomBinarySearchTree.h"

#include <iostream>

int main(int argc, char const *argv[])
{
    size_t cntOfTests = std::stoull(std::string(argv[2]));
    if (argc >= 4 && (cntOfTests + 3 == (size_t)argc))
    {
        std::string str = argv[1];
        std::vector<char> arr;
        for (auto it = str.begin(); it != str.end(); ++it)
            arr.push_back(*it);
        RandomBinarySearchTree<char> tree(arr);
        std::stoull(std::string(argv[2]));
        for (size_t i = 0; i <
std::stoull(std::string(argv[2])); ++i)
        {
            if (i > 0)
                std::cout << ' ';
            std::cout << tree.taskFindAndInsert(argv[3 + i]
[0]);
        }
        std::cout << '\n';
    }
    else
        std::cerr << "incorrect input\n";

    return 0;
}
```

BinaryTreeNode.h

```
#ifndef __BINARYTREENODE__H__
#define __BINARYTREENODE__H__

#include <memory>

template <typename T>
class BinaryTreeNode
{
    T data;

    const size_t recursiveSize(const
std::shared_ptr<BinaryTreeNode<T>> &ptrNode) const;

public:
    std::shared_ptr<BinaryTreeNode<T>> left;
    std::shared_ptr<BinaryTreeNode<T>> right;
    std::weak_ptr<BinaryTreeNode<T>> parent;
```

```
    BinaryTreeNode(const T &val, const
std::shared_ptr<BinaryTreeNode<T>> &ptrParent = nullptr,
const std::shared_ptr<BinaryTreeNode<T>> &ptrLeft = nullptr,
const std::shared_ptr<BinaryTreeNode<T>> &ptrRight =
nullptr);
    void setData(const T &newData);
    const size_t size() const;
    const T getData() const;
    template <typename C>
    friend bool operator==(const BinaryTreeNode<C> &left,
const BinaryTreeNode<C> &right);

    BinaryTreeNode(BinaryTreeNode<T> &&toMove);
    BinaryTreeNode(const BinaryTreeNode<T> &toCopy);
    BinaryTreeNode<T> &operator=(BinaryTreeNode<T> &&toMove);
    BinaryTreeNode<T> &operator=(const BinaryTreeNode<T>
&toMove);

    template <typename C>
    friend std::ostream &operator<<(std::ostream &out, const
BinaryTreeNode<C> &right);
};

template <typename T>
BinaryTreeNode<T> &BinaryTreeNode<T>::operator=(const
BinaryTreeNode<T> &toMove)
{
    if (this != &toMove)
    {
        this->data = toMove->data;
        this->left = toMove->left;
        this->right = toMove->right;
        this->parent = toMove->parent;
    }
    return *this;
}

template <typename T>
BinaryTreeNode<T>
&BinaryTreeNode<T>::operator=(BinaryTreeNode<T> &&toMove)
{
    if (this != &toMove)
    {
        this->data = toMove->data;
        this->left = toMove->left;
        this->right = toMove->right;
        this->parent = toMove->parent;
        toMove->left = nullptr;
        toMove->right = nullptr;
        toMove->parent = nullptr;
    }
    return *this;
}
```

```
template <typename T>
BinaryTreeNode<T>::BinaryTreeNode(BinaryTreeNode<T> &&toMove)
{
    this->data = toMove->data;
    this->left = toMove->left;
    this->right = toMove->right;
    this->parent = toMove->parent;
    toMove->left = nullptr;
    toMove->right = nullptr;
    toMove->parent = nullptr;
}

template <typename T>
BinaryTreeNode<T>::BinaryTreeNode(const BinaryTreeNode<T>
&toCopy)
{
    this->data = toCopy->data;
    this->left = toCopy->left;
    this->right = toCopy->right;
    this->parent = toCopy->parent;
}

template <typename T>
const size_t BinaryTreeNode<T>::size() const
{
    return (1 + this->recursiveSize(this->left) + this-
>recursiveSize(this->right));
}

template <typename T>
const size_t BinaryTreeNode<T>::recursiveSize(const
std::shared_ptr<BinaryTreeNode<T>> &ptrNode) const
{
    return (ptrNode != nullptr) ? (1 + this-
>recursiveSize(ptrNode->left) + this->recursiveSize(ptrNode-
>right)) : 0;
}

template <typename C>
std::ostream &operator<<(std::ostream &out, const
BinaryTreeNode<C> &right)
{
    out << right.data;
    return out;
}

template <typename T>
void BinaryTreeNode<T>::setData(const T &newData)
{
    this->data = newData;
}

template <typename T>
const T BinaryTreeNode<T>::getData() const
{
```

```
    return this->data;
}

template <typename T>
BinaryTreeNode<T>::BinaryTreeNode(const T &val, const
std::shared_ptr<BinaryTreeNode<T>> &ptrParent, const
std::shared_ptr<BinaryTreeNode<T>> &ptrLeft, const
std::shared_ptr<BinaryTreeNode<T>> &ptrRight) : data(val),
left(ptrLeft), right(ptrRight), parent(ptrParent) {}

template <typename C>
bool operator==(const BinaryTreeNode<C> &left, const
BinaryTreeNode<C> &right)
{
    if (left.data != right.data)
        return 0;
    else
    {
        if ((left.left == nullptr || right.left == nullptr)
&& right.left != left.left)
            return 0;
        else
        {
            if ((left.right == nullptr || right.right ==
nullptr) && right.right != left.right)
                return 0;
            else
            {
                if (left.left != nullptr && left.right ==
nullptr)
                    return *(left.left) == *(right.left);
                else if (left.left == nullptr && left.right !
= nullptr)
                    return *(left.right) == *(right.right);
                else if (left.left != nullptr && left.right !
= nullptr)
                    return *(left.right) == *(right.right) &&
*(left.left) == *(right.left);
                else
                    return 1;
            }
        }
    }
}
```

RandomBinarySearchTree.h

```
#ifndef __RANDOMBINARYSEARCHTREE__H__
#define __RANDOMBINARYSEARCHTREE__H__

#include <iostream>
#include <ostream>
#include <vector>
#include <ctime>
#include <iomanip>
#include <cmath>
```

```
#include <sstream>
#include <algorithm>

#include "BinaryTreeNode.h"

template <typename T>
class RandomBinarySearchTree
{
    std::shared_ptr<BinaryTreeNode<T>> head;

    std::shared_ptr<BinaryTreeNode<T>>
merge(std::shared_ptr<BinaryTreeNode<T>> &ptrLeft,
std::shared_ptr<BinaryTreeNode<T>> &ptrRight);
    void recursivePrefixTraverse(std::vector<T>
&vectorOfNodes, const std::shared_ptr<BinaryTreeNode<T>>
&nodePtr) const;
    void recursivePostfixTraverse(std::vector<T>
&vectorOfNodes, const std::shared_ptr<BinaryTreeNode<T>>
&nodePtr) const;
    void recursiveInfixTraverse(std::vector<T>
&vectorOfNodes, const std::shared_ptr<BinaryTreeNode<T>>
&nodePtr) const;
    void recursiveInsert(const T &val,
std::shared_ptr<BinaryTreeNode<T>> &ptrNode, const
std::shared_ptr<BinaryTreeNode<T>> &ptrParent = nullptr);
    void recursiveRemove(const T &val,
std::shared_ptr<BinaryTreeNode<T>> &ptrNode);
    const size_t recursiveSize(const
std::shared_ptr<BinaryTreeNode<T>> &ptrNode) const;
    std::shared_ptr<BinaryTreeNode<T>> recursiveCopy(const
std::shared_ptr<BinaryTreeNode<T>> &ptrNodeToCopy, const
std::shared_ptr<BinaryTreeNode<T>> &ptrParent = nullptr);
    void insertAtRoot(const T &val,
std::shared_ptr<BinaryTreeNode<T>> &nodePtr, const
std::shared_ptr<BinaryTreeNode<T>> &ptrParent);
    void
recursiveGetLevel(std::vector<std::shared_ptr<BinaryTreeNode<
T>>> &vec, const std::shared_ptr<BinaryTreeNode<T>> &ptrNode,
const size_t &level, const size_t &curLevel) const;
    const size_t recursiveHeight(const
std::shared_ptr<BinaryTreeNode<T>> &ptrNode) const;
    void leftTreeRotate(const
std::shared_ptr<BinaryTreeNode<T>> &ptrNode);
    void rightTreeRotate(const
std::shared_ptr<BinaryTreeNode<T>> &ptrNode);
    const size_t recursiveFind(const T &val, const
std::shared_ptr<BinaryTreeNode<T>> &ptrNode) const;

public:
    ~RandomBinarySearchTree() = default;
    RandomBinarySearchTree();
    RandomBinarySearchTree(std::vector<T> elems);
    RandomBinarySearchTree(RandomBinarySearchTree<T> &&tree);
    RandomBinarySearchTree(const RandomBinarySearchTree<T>
&tree);
```

```
    RandomBinarySearchTree<T>
&operator=(RandomBinarySearchTree<T> &&tree);
    RandomBinarySearchTree<T> &operator=(const
RandomBinarySearchTree<T> &tree);
    const std::vector<T> prefixTraverse() const;
    const std::vector<T> postfixTraverse() const;
    const std::vector<T> infixTraverse() const;
    const std::vector<std::shared_ptr<BinaryTreeNode<T>>>
getLevel(const size_t &level) const;
    const size_t size() const;
    const size_t height() const;
    const size_t find(const T &val) const;
    bool empty() const;
    void erase();
    void insert(const T &val);
    void remove(const T &val);
    const size_t taskFindAndInsert(const T &val);

    template <typename C>
    friend std::ostream &operator<<(std::ostream &out, const
RandomBinarySearchTree<C> &bTree);
};

template <typename T>
void RandomBinarySearchTree<T>::erase()
{
    this->head = nullptr;
}

template <typename T>
const size_t
RandomBinarySearchTree<T>::taskFindAndInsert(const T &val)
{
    const size_t cnt = this->find(val);
    this->insert(val);
    return cnt;
}

template <typename T>
RandomBinarySearchTree<T>::RandomBinarySearchTree(std::vector
<T> elems)
{
    std::random_shuffle(elems.begin(), elems.end());
    for (auto it = elems.begin(); it != elems.end(); ++it)
        this->insert(*it);
}

template <typename T>
const size_t RandomBinarySearchTree<T>::find(const T &val)
const
{
    return this->recursiveFind(val, this->head);
}

template <typename T>
```

```
const size_t RandomBinarySearchTree<T>::recursiveFind(const T
&val, const std::shared_ptr<BinaryTreeNode<T>> &ptrNode)
const
{
    if (ptrNode != nullptr)
        return ((ptrNode->getData() == val) ? 1 : 0) + this-
>recursiveFind(val, ptrNode->left) + this->recursiveFind(val,
ptrNode->right);
    else
        return 0;
}

template <typename T>
const size_t RandomBinarySearchTree<T>::recursiveHeight(const
std::shared_ptr<BinaryTreeNode<T>> &ptrNode) const
{
    return (ptrNode != nullptr) ? (1 +
std::max<size_t>(recursiveHeight(ptrNode->left),
recursiveHeight(ptrNode->right))) : 0;
}

template <typename T>
const size_t RandomBinarySearchTree<T>::height() const
{
    return this->recursiveHeight(this->head);
}

template <typename T>
const std::vector<std::shared_ptr<BinaryTreeNode<T>>>
RandomBinarySearchTree<T>::getLevel(const size_t &level)
const
{
    std::vector<std::shared_ptr<BinaryTreeNode<T>>>
vecOfDatas;
    this->recursiveGetLevel(vecOfDatas, this->head, level,
0);
    return vecOfDatas;
}

template <typename T>
void
RandomBinarySearchTree<T>::recursiveGetLevel(std::vector<std:
:shared_ptr<BinaryTreeNode<T>>> &vec, const
std::shared_ptr<BinaryTreeNode<T>> &ptrNode, const size_t
&level, const size_t &curLevel) const
{
    std::shared_ptr<BinaryTreeNode<T>> left = nullptr;
    std::shared_ptr<BinaryTreeNode<T>> right = nullptr;
    std::shared_ptr<BinaryTreeNode<T>> root = nullptr;
    if (ptrNode != nullptr)
    {
        left = ptrNode->left;
        right = ptrNode->right;
        root = ptrNode;
    }
```

```
    if (curLevel == level)
        vec.push_back(root);
    else
    {
        this->recursiveGetLevel(vec, left, level, curLevel +
1);
        this->recursiveGetLevel(vec, right, level, curLevel +
1);
    }
}

template <typename T>
void RandomBinarySearchTree<T>::leftTreeRotate(const
std::shared_ptr<BinaryTreeNode<T>> &ptrNode)
{
    if (ptrNode != nullptr)
    {
        std::shared_ptr<BinaryTreeNode<T>> newRoot = ptrNode-
>right;
        ptrNode->right = newRoot->left;
        if (ptrNode->right != nullptr)
            ptrNode->right->parent = ptrNode;
        newRoot->parent = ptrNode->parent;
        newRoot->left = ptrNode;
        newRoot->left->parent = newRoot;

        if (newRoot->parent.lock() != nullptr)
        {
            if (newRoot->parent.lock()->left == ptrNode)
                newRoot->parent.lock()->left = newRoot;

            if (newRoot->parent.lock()->right == ptrNode)
                newRoot->parent.lock()->right = newRoot;
        }
        if (ptrNode == this->head)
            this->head = newRoot;
    }
}

template <typename T>
void RandomBinarySearchTree<T>::rightTreeRotate(const
std::shared_ptr<BinaryTreeNode<T>> &ptrNode)
{
    if (ptrNode != nullptr)
    {
        std::shared_ptr<BinaryTreeNode<T>> newRoot = ptrNode-
>left;
        ptrNode->left = newRoot->right;
        if (ptrNode->left != nullptr)
            ptrNode->left->parent = ptrNode;
        newRoot->parent = ptrNode->parent;
        newRoot->right = ptrNode;
        newRoot->right->parent = newRoot;

        if (newRoot->parent.lock() != nullptr)
```

```
        {
            if (newRoot->parent.lock()->left == ptrNode)
                newRoot->parent.lock()->left = newRoot;

            if (newRoot->parent.lock()->right == ptrNode)
                newRoot->parent.lock()->right = newRoot;
        }
        if (ptrNode == this->head)
            this->head = newRoot;
    }
}

template <typename T>
const size_t RandomBinarySearchTree<T>::size() const
{
    return this->recursiveSize(this->head);
}

template <typename T>
const size_t RandomBinarySearchTree<T>::recursiveSize(const
std::shared_ptr<BinaryTreeNode<T>> &ptrNode) const
{
    return (ptrNode != nullptr) ? (1 + this-
>recursiveSize(ptrNode->left) + this->recursiveSize(ptrNode-
>right)) : 0;
}

template <typename T>
RandomBinarySearchTree<T>::RandomBinarySearchTree(const
RandomBinarySearchTree<T> &tree) : head(this-
>recursiveCopy(tree.head, nullptr)) {}

template <typename T>
RandomBinarySearchTree<T>
&RandomBinarySearchTree<T>::operator=(const
RandomBinarySearchTree<T> &tree)
{
    if (this != &tree)
        this->head = this->recursiveCopy(tree.head, nullptr);
    return *this;
}

template <typename T>
std::shared_ptr<BinaryTreeNode<T>>
RandomBinarySearchTree<T>::recursiveCopy(const
std::shared_ptr<BinaryTreeNode<T>> &ptrNodeToCopy, const
std::shared_ptr<BinaryTreeNode<T>> &ptrParent)
{
    if (ptrNodeToCopy != nullptr)
    {
        std::shared_ptr<BinaryTreeNode<T>> newObj(new
BinaryTreeNode<T>(ptrNodeToCopy->getData(), ptrParent));
        newObj->left = recursiveCopy(ptrNodeToCopy->left,
newObj);
```

```
        newObj->right = recursiveCopy(ptrNodeToCopy->right,
newObj);
        return newObj;
    }
    else
        return nullptr;
}

template <typename T>
RandomBinarySearchTree<T>::RandomBinarySearchTree(RandomBinar
ySearchTree<T> &&tree) : head(std::move(tree.head)) {}

template <typename T>
RandomBinarySearchTree<T>
&RandomBinarySearchTree<T>::operator=(RandomBinarySearchTree<
T> &&tree)
{
    if (&tree != this)
        this->head = std::move(tree.head);
    return *this;
}

template <typename T>
void RandomBinarySearchTree<T>::remove(const T &val)
{
    this->recursiveRemove(val, this->head);
}

template <typename T>
void RandomBinarySearchTree<T>::recursiveRemove(const T &val,
std::shared_ptr<BinaryTreeNode<T>> &ptrNode)
{
    if (ptrNode != nullptr)
    {
        if (ptrNode->getData() < val)
            recursiveRemove(val, ptrNode->right);
        else if (ptrNode->getData() > val)
            recursiveRemove(val, ptrNode->left);
        else
        {
            std::shared_ptr<BinaryTreeNode<T>> ptrParent =
ptrNode->parent.lock();
            std::shared_ptr<BinaryTreeNode<T>> ptrTmp = this-
>merge(ptrNode->left, ptrNode->right);
            ptrNode->parent = ptrParent;
            if (ptrTmp == nullptr)
            {
                if (ptrParent != nullptr)
                {
                    if (ptrParent->right == ptrNode)
                        ptrParent->right = nullptr;
                    else
                        ptrParent->left = nullptr;
                }
                if (this->head == ptrNode)
```

```
                    this->head = nullptr;
                }
                ptrNode = ptrTmp;
            }
        }
    }

    template <typename T>
    std::shared_ptr<BinaryTreeNode<T>>
    RandomBinarySearchTree<T>::merge(std::shared_ptr<BinaryTreeNo
    de<T>> &ptrLeft, std::shared_ptr<BinaryTreeNode<T>>
    &ptrRight)
    {
        std::shared_ptr<BinaryTreeNode<T>> ptrNode = nullptr;
        const size_t leftSize = (ptrLeft != nullptr) ? ptrLeft-
    >size() : 0;
        const size_t rightSize = (ptrRight != nullptr) ?
    ptrRight->size() : 0;
        const size_t totalSize = leftSize + rightSize;
        if (totalSize != 0)
        {
            srand(time(0));
            const size_t randNum = 1 + ((size_t)rand() %
    totalSize);
            if (randNum <= leftSize)
            {
                ptrNode = ptrLeft;
                ptrNode->right = merge(ptrNode->right, ptrRight);
            }
            else
            {
                ptrNode = ptrRight;
                ptrNode->left = merge(ptrLeft, ptrNode->left);
            }
        }
        return ptrNode;
    }

    template <typename T>
    void RandomBinarySearchTree<T>::insert(const T &val)
    {
        this->recursiveInsert(val, this->head);
    }

    template <typename T>
    void RandomBinarySearchTree<T>::recursiveInsert(const T &val,
    std::shared_ptr<BinaryTreeNode<T>> &ptrNode, const
    std::shared_ptr<BinaryTreeNode<T>> &ptrParent)
    {
        if (ptrNode == nullptr)
            ptrNode = std::shared_ptr<BinaryTreeNode<T>>(new
    BinaryTreeNode<T>(val, ptrParent));
        else
        {
            const size_t treeSize = this->size();
```

```
        srand(time(0));
        const size_t randNum = 1 + (size_t)rand() % (treeSize
+ 1);

        if (randNum == treeSize + 1)
            this->insertAtRoot(val, ptrNode, ptrParent);
        else
        {
            if (ptrNode->getData() <= val)
                this->recursiveInsert(val, ptrNode->right,
ptrNode);
            else if (val < ptrNode->getData())
                this->recursiveInsert(val, ptrNode->left,
ptrNode);
        }
    }
}

template <typename T>
void RandomBinarySearchTree<T>::insertAtRoot(const T &val,
std::shared_ptr<BinaryTreeNode<T>> &ptrNode, const
std::shared_ptr<BinaryTreeNode<T>> &ptrParent)
{
    if (ptrNode == nullptr)
        ptrNode = std::shared_ptr<BinaryTreeNode<T>>(new
BinaryTreeNode<T>(val, ptrParent));
    else
    {
        if (ptrNode->getData() <= val)
        {
            this->insertAtRoot(val, ptrNode->right, ptrNode);
            this->leftTreeRotate(ptrNode);
        }
        if (val < ptrNode->getData())
        {
            this->insertAtRoot(val, ptrNode->left, ptrNode);
            this->rightTreeRotate(ptrNode);
        }
    }
}

template <typename T>
std::ostream &operator<<(std::ostream &out, const
RandomBinarySearchTree<T> &bTree)
{
    out << "Tree:\n";

std::vector<std::vector<std::shared_ptr<BinaryTreeNode<T>>>>
vecOfDatasInCurrentLevel;
    const size_t h = bTree.height();
    size_t maxLen = 0;

    for (size_t currentLevel = 0; currentLevel < h; +
+currentLevel)
    {
```

```
vecOfDatasInCurrentLevel.push_back(bTree.getLevel(currentLeve
l));
        for (auto it =
vecOfDatasInCurrentLevel[currentLevel].begin(); it !=
vecOfDatasInCurrentLevel[currentLevel].end(); ++it)
        {
            if ((*it) != nullptr)
            {
                std::ostringstream strToCheckLen;
                strToCheckLen << (*it)->getData();
                const size_t newLen =
strToCheckLen.str().length();
                if (newLen > maxLen)
                    maxLen = newLen;
            }
        }
    }
    for (size_t currentLevel = 0; currentLevel < h; +
+currentLevel)
    {
        if (currentLevel > 0)
            out << '\n';
        for (auto it =
vecOfDatasInCurrentLevel[currentLevel].begin(); it !=
vecOfDatasInCurrentLevel[currentLevel].end(); ++it)
        {
            if (it ==
vecOfDatasInCurrentLevel[currentLevel].begin())
                out << std::setw((1 << (h - 1 -
currentLevel)) * maxLen);
            else
                out << std::setw((1 << (h - currentLevel)) *
maxLen);
            if (*it == nullptr)
                out << '#';
            else
                out << (*it)->getData();
        }
    }
    return out;
}

template <typename T>
RandomBinarySearchTree<T>::RandomBinarySearchTree()
{
    this->head = std::unique_ptr<BinaryTreeNode<T>>(nullptr);
}

template <typename T>
const std::vector<T>
RandomBinarySearchTree<T>::prefixTraverse() const
{
    std::vector<T> vectorOfNodes;
    recursivePrefixTraverse(vectorOfNodes, this->head);
```

```
    return vectorOfNodes;
}

template <typename T>
const std::vector<T>
RandomBinarySearchTree<T>::postfixTraverse() const
{
    std::vector<T> vectorOfNodes;
    recursivePostfixTraverse(vectorOfNodes, this->head);
    return vectorOfNodes;
}

template <typename T>
const std::vector<T>
RandomBinarySearchTree<T>::infixTraverse() const
{
    std::vector<T> vectorOfNodes;
    recursiveInfixTraverse(vectorOfNodes, this->head);
    return vectorOfNodes;
}

template <typename T>
void
RandomBinarySearchTree<T>::recursivePrefixTraverse(std::vecto
r<T> &vectorOfNodes, const std::shared_ptr<BinaryTreeNode<T>>
&nodePtr) const
{
    if (nodePtr != nullptr)
    {
        vectorOfNodes.push_back(nodePtr->getData());
        this->recursivePrefixTraverse(vectorOfNodes, nodePtr-
>left);
        this->recursivePrefixTraverse(vectorOfNodes, nodePtr-
>right);
    }
}

template <typename T>
void
RandomBinarySearchTree<T>::recursivePostfixTraverse(std::vect
or<T> &vectorOfNodes, const
std::shared_ptr<BinaryTreeNode<T>> &nodePtr) const
{
    if (nodePtr != nullptr)
    {
        this->recursivePostfixTraverse(vectorOfNodes,
nodePtr->left);
        this->recursivePostfixTraverse(vectorOfNodes,
nodePtr->right);
        vectorOfNodes.push_back(nodePtr->getData());
    }
}

template <typename T>
```

```
void
RandomBinarySearchTree<T>::recursiveInfixTraverse(std::vector
<T> &vectorOfNodes, const std::shared_ptr<BinaryTreeNode<T>>
&nodePtr) const
{
    if (nodePtr != nullptr)
    {
        this->recursiveInfixTraverse(vectorOfNodes, nodePtr-
>left);
        vectorOfNodes.push_back(nodePtr->getData());
        this->recursiveInfixTraverse(vectorOfNodes, nodePtr-
>right);
    }
}

template <typename T>
bool RandomBinarySearchTree<T>::empty() const
{
    return (this->head == nullptr) ? 0 : 1;
}

#endif //!__RANDOMBINARYSEARCHTREE__H__

#endif //!__BINARYTREENODE__H__
```

Makefile

```
compiler = g++
flags = -c -std=c++17 -Wall
appname = lab5
lib_dir = Sources/libs/
src_dir = Sources/srcs/

src_files := $(wildcard $(src_dir)*)
obj_files := $(addsuffix .o, $(basename $(notdir $
(src_files))))

define compile
    $(compiler) $(flags) $<
endef

programbuild: $(obj_files)
    $(compiler) $^ -o $(appname)

%.o: $(src_dir)/%.cpp $(lib_dir)/*.h
    $(call compile)

clean:
    rm -f *.o $(appname)
```