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

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: Исследование случайных БДП с рандомизацией в среднем и в худшем случаях

Студентка гр. 9303 _____ Булыно Д.А.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург

2020

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студентка Булыно Д.А.

Группа 9303

Тема работы: исследование случайных БДП с рандомизацией в среднем и в худшем случаях.

Исходные данные: реализовать программу по созданию случайных БДП с рандомизацией, провести исследование случайных БДП с рандомизацией: сгенерировать входные данные, использовать их для измерения количественных характеристик структур данных, алгоритмов, действий, сравнить экспериментальные результаты с теоретическими.

Содержание пояснительной записки: аннотация, содержание, введение, описание алгоритма, описание структур данных и функций, тестирование, исследование случайного БДП с рандомизацией, заключение, список использованных источников, исходный код (в приложении).

Предполагаемый объем пояснительной записки:

Не менее 10 страниц.

Дата выдачи задания: 06.11.2020

Дата сдачи реферата: 25.12.2020

Дата защиты реферата: 25.12.2020

Студентка _____ Булыно Д.А.

Преподаватель _____ Филатов А.Ю.

АННОТАЦИЯ

В данной работе проводилось исследование случайных БДП с рандомизацией в среднем и в худшем случаях. Для этого были реализованы 2 функции для генерации наборов входных данных, помогающих установить сложность алгоритмов на основе подсчёта количества произведенных операций алгоритма, подсчёта времени основных операций, производимых данными деревьями. В работе приведены графики, основанные на подсчёте базовых операций, производится сопоставление с теоретическими данными.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы: реализация и экспериментальное машинное исследование алгоритмов случайного БДП с рандомизацией.

Основные задачи:

1. Провести реализацию случайного БДП с рандомизацией.
2. Реализовать программу для генерации представительного множества реализаций входных данных для среднего и для худшего случаев.
3. Выполнить исследуемые алгоритмы на сгенерированных наборах данных, при этом в ходе вычислительного процесса фиксируя как время работы программы, так и количество произведенных базовых операций алгоритма.
4. На основе полученных данных привести статистику.
5. Представить результаты испытаний и сопоставить с теоретическими оценками.

1. ОПИСАНИЕ СЛУЧАЙНОГО БДП С РАНДОМИЗАЦИЕЙ

1.1. Основные теоретические сведения

Бинарное дерево — это конечное множество узлов, которое либо пусто, либо состоит из корня и двух непересекающихся бинарных деревьев, называемых правым поддеревом и левым поддеревом.

Бинарное дерево называется бинарным деревом поиска (БДП), если для каждого узла `x` выполняется следующее условие: для каждого узла `l` в левом поддереве справедливо `key(l) < key(x)`, а для каждого узла `r` в правом поддереве – `key(r) > key(x)`.

Можно дать другое определение БДП: бинарное дерево является БДП, если ключи узлов дерева упорядочены при ЛКП-обходе.

В случайных БДП структура дерева определяется последовательностью вставок и удалений элементов. При этом высота дерева никак не регулируется, поэтому на некоторых последовательностях операций (например, вставка упорядоченной последовательности элементов), дерево будет вырождаться в линейный список с максимальной высотой, равной `n`.

Ключевая идея случайных БДП с рандомизацией состоит в чередовании обычной вставки в дерево поиска и вставки в корень. Чередование происходит случайным (рандомизированным) образом с использованием компьютерного генератора псевдослучайных чисел. Цель такого чередования – сохранить хорошие свойства случайного БДП в среднем и исключить (сделать маловероятным) появление «худшего случая» (поддеревьев большой высоты).

1.2. Описание алгоритма

Основное свойство дерева поиска — любой ключ в левом поддереве меньше корневого ключа, а в правом поддереве — больше корневого. Это свойство позволяет организовать поиск заданного ключа, перемещаясь от корня вправо или влево в зависимости от значения корневого ключа.

Необходимой составляющей рандомизации является применение специальной вставки нового ключа, в результате которой новый ключ

оказывается в корне дерева. Для реализации вставки в корень используется функция поворота, которая производит локальное преобразование дерева.

Вставка в корень. Сначала рекурсивно вставляется новый ключ в корень левого или правого поддеревьев (в зависимости от результата сравнения с корневым ключом), а после выполняется правый (левый) поворот, который поднимает нужный узел в корень дерева.

Рандомизированная вставка значения в дерево. Пусть в дереве имеется `n` узлов. После добавления еще одного узла любой узел с равной вероятностью может быть корнем дерева. Из-за этого с вероятностью `1/(n + 1)` осуществляется вставка в корень, иначе рекурсивно используется рандомизированная вставка в левое или правое поддерево в зависимости от значения ключа.

Удаление значения из дерева. Способ 1. Если удаляемый узел — лист, то просто производится его удаление. Если у узла один ребенок, то узел удаляется, вместо него размещается узел-ребенок. Если у узла два ребенка, то ищется узел, который непосредственно следует за исходным узлом при ЛКП обходе, т. е. узел с минимальным значением в правом поддереве. Значения узлов обмениваются и узел, имеющий не более одного ребенка, удаляется.

Удаление значения из дерева. Способ 2 (используется в коде программы). Пусть даны два дерева поиска с корнями `p` и `q`, причем любой ключ первого дерева меньше любого ключа во втором дереве. Требуется объединить эти два дерева в одно. В качестве корня нового дерева можно взять любой из двух корней, пусть это будет `p`. Тогда левое поддерево `p` можно оставить как есть, а справа к `p` подвесить объединение двух деревьев — правого поддерева `p` и всего дерева `q` (они удовлетворяют всем условиям задачи).

С другой стороны, с тем же успехом можно сделать корнем нового дерева узел `q`. В рандомизированной реализации выбор между этими альтернативами делается случайным образом. Пусть размер левого дерева равен `n`, правого — `m`. Тогда `p` выбирается новым корнем с вероятностью `n/(n + m)`, а `q` — с вероятностью `m/(n + m)`.

Удаление происходит по ключу — ищется узел с заданным ключом (все ключи различны) и этот узел удаляется из дерева. Стадия поиска такая же, как и при поиске в БДП. После объединяются левое и правое поддеревья найденного узла, удаляется узел, возвращается корень объединенного дерева.

1.3. Описание структур данных и функций

Для реализации и исследования случайных БДП с рандомизацией была создана структура для представления узлов дерева `struct Node`. Также были реализованы следующие функции:

`int getsize (Node* btree)` – функция-обёртка для поля `size`;

`int fixsize(Node* btree)` – функция для установления корректного размера дерева;

`int height(Node* btree)` – функция для подсчёта максимальной высоты дерева, возвращающая, соответственно, максимальную высоту дерева;

`Node* find(Node* btree, int key)` – функция поиска элемента по ключу, написанная по алгоритму поиска в БДП, возвращающая поддерево, корнем которой является искомый элемент;

`Node* usual_insert(Node* btree, int k)` – функция классической вставки нового элемента по ключу, используется для создания случайного БДП в качестве сравнения со случайным БДП с рандомизацией;

`Node* rotateright(Node* btree)` – функция поворота вправо, которая используется в функции вставки элемента в корень;

`Node* rotateleft(Node* btree)` – функция поворота влево, которая используется в функции вставки элемента в корень;

`Node* insertroot(Node* btree, int k)` – функция вставки элемента в корень дерева по ключу;

`void Print(Node* btree, int level)` – функция для вывода дерева, повернутого на 90 градусов;

`Node* insert(Node* btree, int k)` – функция рандомизированной вставки элемента по ключу, написанная по алгоритму;

`Node* join(Node* btree1, Node* btree2)` – функция для объединения двух деревьев, используется в функции удаления первого найденного узла по ключу;

`Node* remove(Node* btree, int k)` – функция удаления первого найденного узла по ключу;

`vector<int> generate_average(int n)` – функция генерации элементов для среднего случая, подробнее в п. 2.2.;

`vector<int> generate_worst(int n)` – функция генерации элементов для худшего случая, подробнее в п. 2.2;

`void function_for_bst(string generation, Node* tree, Node* newtree)` – функция для работы со сгенерированными деревьями: функция осуществляет поиск, вставку и удаление элементов в сгенерированных деревьях;

`void func_for_researhing_average(int size, int new_iterations, string generation, Node* tree, Node* btree, Node* newtree, vector<int> average_case)` – функция создания дерева для среднего случая из сгенерированных значений;

`void func_for_researching_worst(int size, int new_iterations, string generation, Node* tree, Node* btree, Node* newtree, vector<int> worst_case)` – функция создания дерева для худшего случая из сгенерированных значений;

`int main()` – главная функция, в которой производится выбор между генерацией и созданием исследуемых деревьев.

2. ИССЛЕДОВАНИЕ СЛУЧАЙНОГО БДП С РАНДОМИЗАЦИЕЙ

2.1. Теоретическая оценка сложности алгоритмов

Цель рандомизации – сохранить хорошие свойства случайного БДП в среднем и исключить (сделать маловероятным) появление «худшего случая» (поддеревьев большой высоты). Поэтому в среднем у такого дерева сложность вставки и удаления будет таким же, как и у случайного БДП, а в худшем случае будет вариативность $\log_2 n$ или n, как у случайного БДП в худшем случае.

2.2. Генерация предварительного множества реализаций входных данных для среднего и худшего случаев

Для генерации входных данных были написаны функции `vector<int> generate_average(int n)` и `vector<int> generate_worst(int n)`, которые генерируют числовые последовательности размера `n`.

Функция `vector<int> generate_average(int n)` предусматривает невозрастающую и неубывающую последовательность из `n` случайных чисел, в то время как функция `vector<int> generate_worst(int n)` генерирует случайным образом либо возрастающую, либо убывающую последовательность для определения свойств исследуемого дерева.

2.3. Выполнение исследуемых алгоритмов

Для исследования сгенерированы случайные БДП с рандомизацией в 10, 100, 1000, 10000 элементов для среднего и худшего случаев.

Вставка элемента.

В таблице 1 представлены результаты тестирования алгоритма вставки для среднего и худшего случаев.

Таблица 1. Усредненные результаты тестирования алгоритма вставки.

Кол-во элементов	**Средний случай**		**Худший случай**	
	Кол-во итераций	**Время работы,s**	**Кол-во итераций**	**Время работы,s**
10	5.2	3.12e-06	5.4	3.74e-06
100	13.6	3.84e-06	10.8	4.16e-06
1000	15.8	5e-06	13.6	4.56e-06
10000	22.8	6.7e-6	21.8	5.82e-06

Графики зависимостей количества итераций от количества элементов для алгоритма вставки для среднего и худшего случаев приведены на рис. 1 и 2.

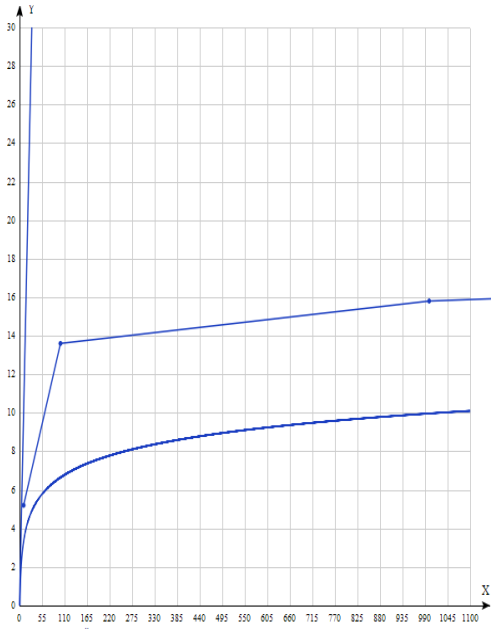

Рисунок 1. График алгоритма вставки для среднего случая

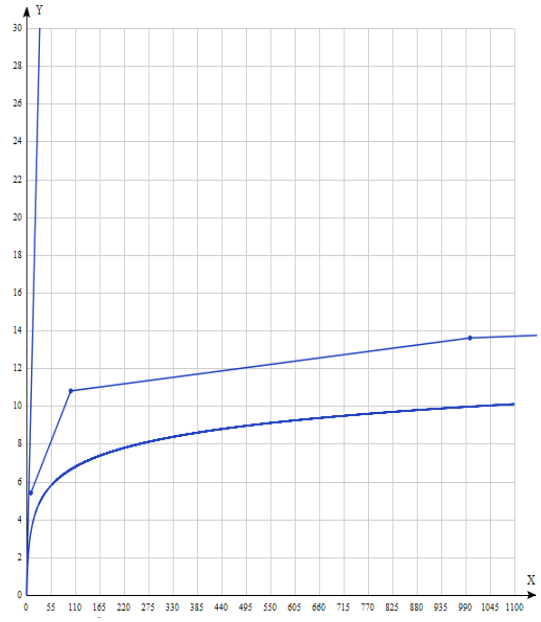

Рисунок 2. График алгоритма вставки для худшего случая

Построенные по точкам графики – графики по полученным результатам. Функция, находящаяся рядом с осью Oy, – $y = x$, под полученным графиком располагается функция $y = \log_2 x$. Исходя из графиков, изображенных на рис. 1 и 2, можно сделать вывод, что сложность алгоритма вставки для среднего и худшего случаев при данных значениях будет равна $O(\log_2 x)$, однако может быть и $O(x)$ в худшем, если будет происходить постоянно случайная вставка элемента в корень.

Удаление элемента.

В таблице 2 представлены результаты тестирования алгоритма удаления для среднего и худшего случаев.

Таблица 2. Усредненные результаты тестирования алгоритма удаления.

Кол-во элементов	**Средний случай**		**Худший случай**	
	Кол-во итераций	**Время работы,s**	**Кол-во итераций**	**Время работы,s**
10	4.4	2.66e-06	3.8	2.86e-06
100	8.6	2.6e-06	9.8	2.7e-06
1000	14.4	2.9e-06	14.4	2.44e-06
10000	17.4	2.44e-06	15.8	3.14e-06

Графики зависимостей количества итераций от количества элементов для алгоритма вставки для среднего и худшего случаев приведены на рис. 3 и 4.

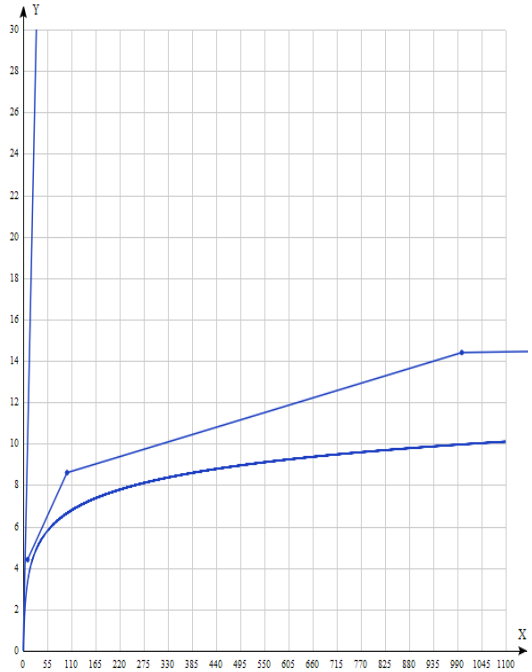

Рисунок 3. График алгоритма удаления для среднего случая

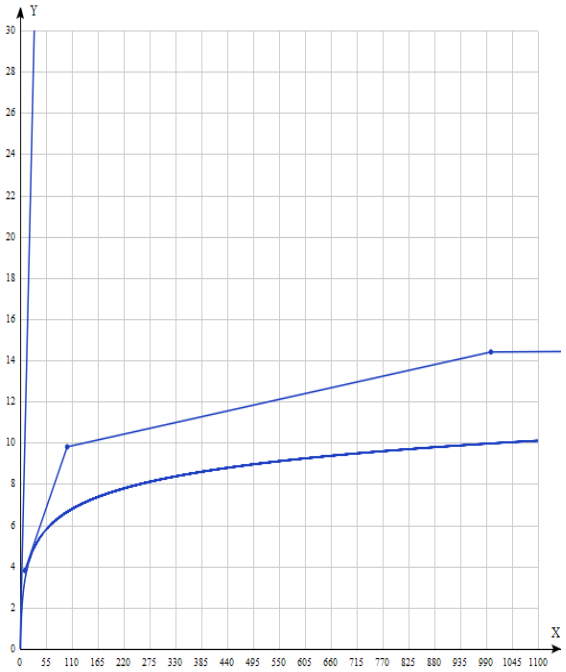

Рисунок 4. График алгоритма удаления для худшего случая

Построенные по точкам графики – графики по полученным результатам. Функция, находящаяся рядом с осью Oy, – $y = x$, под полученным графиком располагается функция $y = \log_2 x$. Исходя из графиков, изображенных на рисунках 3 и 4, можно сделать вывод, что сложность алгоритма удаления для среднего и худшего случаев будет равна $O(\log_2 x)$, однако может быть и $O(x)$ в худшем, если будет происходить постоянно случайная вставка элемента в корень.

2.4. Тестирование

На рис. 5 представлена генерация элементов для среднего случая, состоящего из 100 элементов, реализация дерева, сравнение данных со случайным БДП.

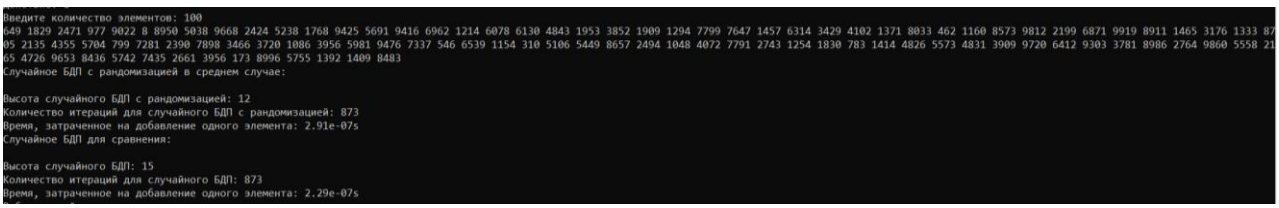

Рисунок 5. Генерация элементов для среднего случая

На рис. 6, 7, 8, 9 продемонстрированы поиск элемента, вставка случайного элемента, вставка элемента и удаление элемента соответственно в сгенерированном дереве для среднего случая.

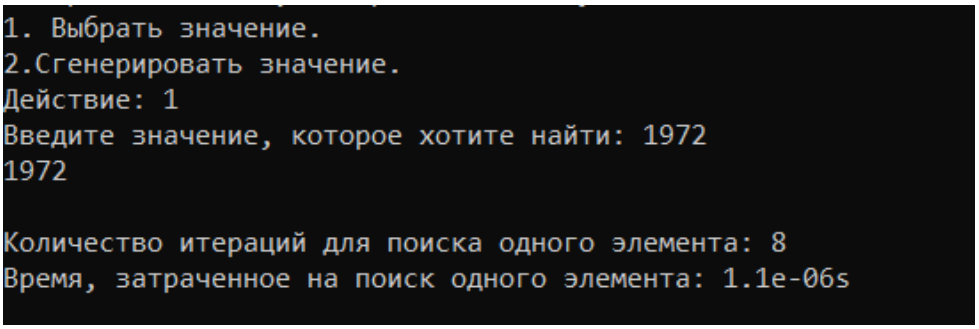

Рисунок 6. Поиск элемента в сгенерированном дереве для среднего случая

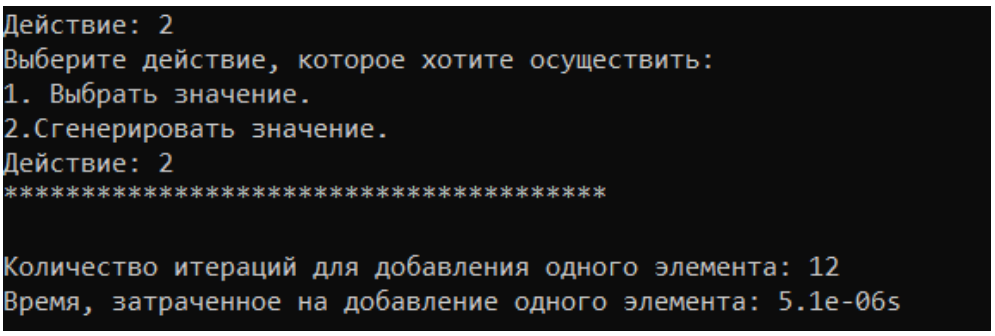

Рисунок 7. Вставка случайного элемента в сгенерированном дереве для среднего случая

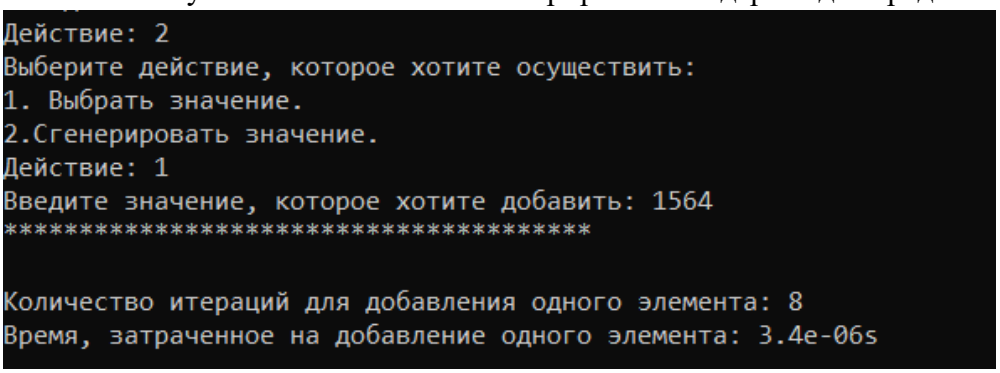

Рисунок 8. Вставка элемента в сгенерированном дереве для среднего случая

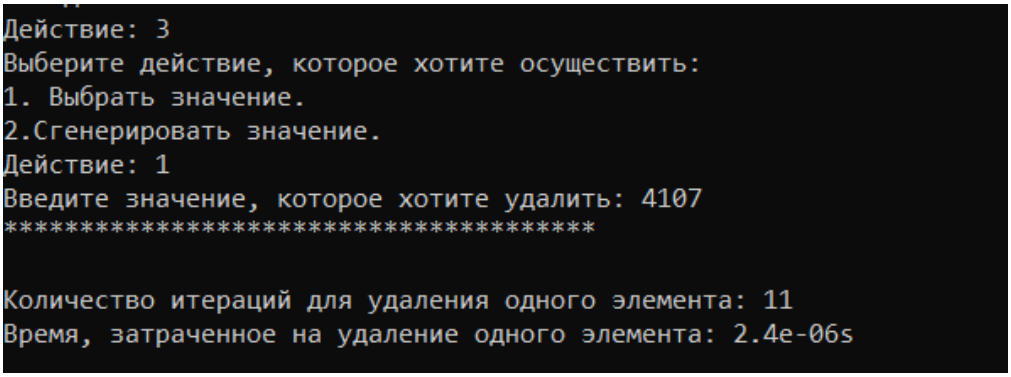

Рисунок 9. Удаление элемента в сгенерированном дереве для среднего случая

На рис. 10 представлена генерация элементов для худшего случая, состоящего из 10 элементов, реализация дерева, сравнение данных со случайным БДП.

```
Действие:
2
Введите количество элементов: 10
25 1136 1931 2294 3866 3953 5798 7626 7709 8035

Случайное БДП с рандомизацией в худшем случае:
            8035
        7709
    7626
                5798
            3953
        3866
2294
    1931
        1136
            25

Высота случайного БДП с рандомизацией: 5
Количество итераций для случайного БДП с рандомизацией: 34
Время, затраченное на добавление одного элемента: 6.2e-07s
Случайное БДП для сравнения:
                                    8035
                                7709
                            7626
                        5798
                    3953
                3866
            2294
        1931
    1136
25

Высота случайного БДП: 10
Количество итераций для случайного БДП: 55
Время, затраченное на добавление одного элемента: 1.1e-07s
```

Рисунок 10. Генерация элементов для худшего случая

Поиск, вставка и удаление элемента в сгенерированном дереве для худшего случая делаются аналогично алгоритмам сгенерированного дерева для среднего случая.

На рис. 11 представлено создание исследуемого дерева.

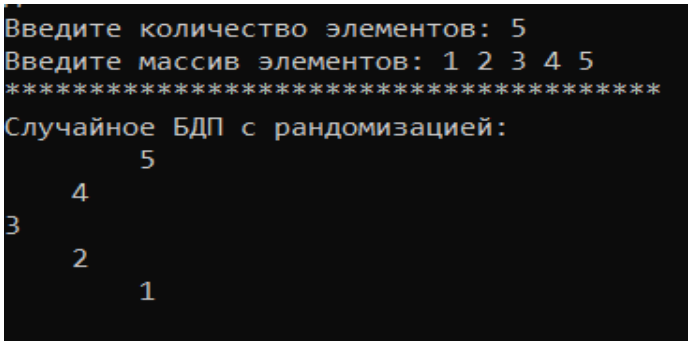

Рисунок 11. Создание исследуемого дерева

На рис. 12 и 13 представлены вставка элемента в корень и обычная вставка для сгенерированного дерева.

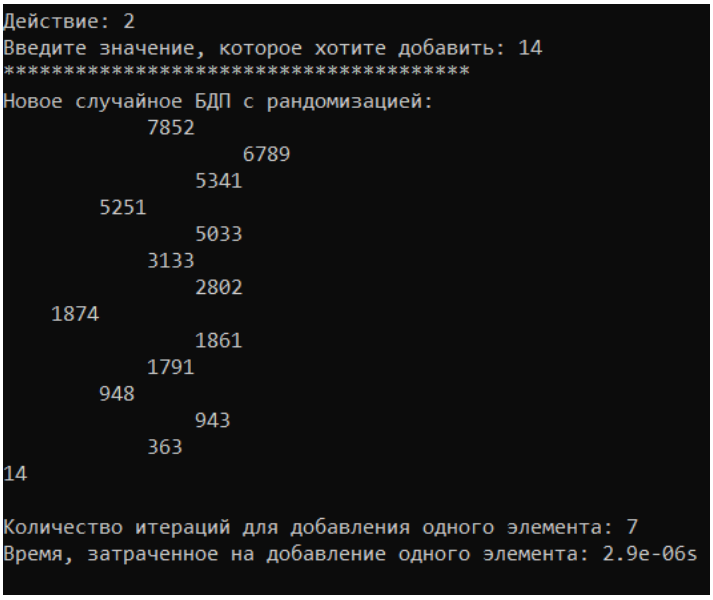

Рисунок 12. Вставка элемента в корень

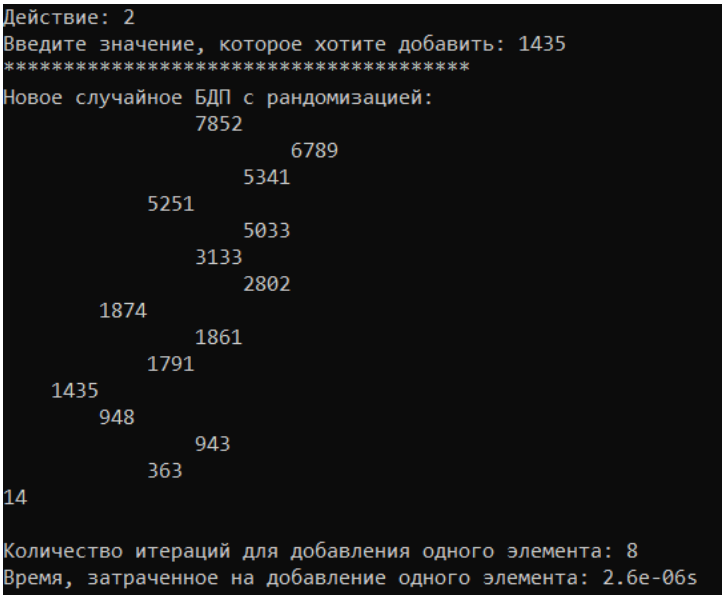

Рисунок 13. Обычная вставка для рандомизированного дерева

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы была написана программа для реализации и исследования случайного БДП с рандомизацией с помощью генерируемых значений для среднего и худшего случаев.

В результате проведения исследований были оценены и доказаны сложности алгоритмов в случайном БДП с рандомизацией на основе полученных значений. Установлено, что для вставки и удаления в среднем случае сложность алгоритмов составляет $O(\log_2 n)$, в худшем – $O(n)$, как у случайных БДП, либо $O(\log_2 n)$.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Методические указания к лабораторным работам, практическим занятиям и курсовой работе по дисциплине «Алгоритмы и структуры данных»»: учеб.-метод. пособие / сост.: С.А Ивановский , Т.Г. Фомичева , О.М. Шолохова.. СПб. 2017. 88 с.

2. Сайт по работе с языком C++. URL: https://www.cplusplus.com/

3. Хабр. URL: https://habr.com/

4. Ravesli. URL: https://ravesli.com/

ПРИЛОЖЕНИЕ А
КОД ПРОГРАММЫ

main.cpp

```
#include <iostream>
#include <fstream>
#include <ctime>
#include <string>
#include <vector>
#include <chrono>
#include <algorithm>

using namespace std;

//счётчик для количества итераций
int iterations = 0;

//структура дерева
struct Node {
	int key;
	int size;
	Node* left;
	Node* right;
	Node(int k) {
		key = k;
		left = right = nullptr;
		size = 1;
	}
};

int getsize(Node* btree) {
	if (!btree) return 0;
	return btree->size;
}

int fixsize(Node* btree) {
	return btree->size = getsize(btree->left) + getsize(btree->right) +
1;
}

//функция для подсчёта максимальной высоты дерева
int height(Node* btree){
	Node *temp = btree;
	int h1 = 0, h2 = 0;
```

```
      if (btree == NULL)return(0);
      if (btree->left) { h1 = height(btree->left); }
      if (btree->right) { h2 = height(btree->right); }
      return (max(h1, h2) + 1);
}

//функция для нахождения элемента дерева
Node* find(Node* btree, int key){
      if (!btree) {
            iterations++;
            return 0;
      }
      if (key == btree->key) {
            iterations++;
            return btree;
      }
      if (key < btree->key) {
            iterations++;
            return find(btree->left, key);
      }
      else {
            iterations++;
            return find(btree->right, key);
      }
}

//функция для классической вставки элемента в БДП
Node* usual_insert(Node* btree, int k){
      if (!btree) {
            iterations++;
            return new Node(k);
      }
      if (btree->key > k) {
            iterations++;
            btree->left = usual_insert(btree->left, k);
      }
      else {
            iterations++;
            btree->right = usual_insert(btree->right, k);
      }
      fixsize(btree);
      return btree;
}
```

```
//функция для поворота вправо поддерева
Node* rotateright(Node* btree) {
	iterations++;
	//cout << "rotate R " << iterations << endl;
	Node* newbtree = btree->left;
	if (!newbtree) return btree;
	btree->left = newbtree->right;
	newbtree->right = btree;
	newbtree->size = btree->size;
	fixsize(btree);
	return newbtree;
}

//функция для поворота влево поддерева
Node* rotateleft(Node* btree) {
	iterations++;
	//cout << "rotate L " << iterations << endl;
	Node* newbtree = btree->right;
	if (!newbtree) return btree;
	btree->right = newbtree->left;
	newbtree->left = btree;
	newbtree->size = btree->size;
	fixsize(btree);
	return newbtree;
}

//функция для вставки в корень элемента дерева
Node* insertroot(Node* btree, int k) {
	if (!btree) {
		iterations++;
		//cout << "root new" << iterations << endl;
		return new Node(k);
	}
	if (k < btree->key) {
		iterations++;
		//cout << "root left " << iterations << endl;
		btree->left = insertroot(btree->left, k);
		return rotateright(btree);
	}
	else {
		iterations++;
		//cout << "root right " << iterations << endl;
		btree->right = insertroot(btree->right, k);
		return rotateleft(btree);
```

```
	}
}

//функция для вывода дерева, повёрнутого на 90 градусов против часовой стрелки
void Print(Node* btree, int level) {
	if (btree->right != nullptr) {
		Print(btree->right, level + 1);
	}
	for (int i = 0; i < level; i++) {
		cout << "    ";
	}
	cout << btree->key << "\n";
	if (btree->left != nullptr) {
		Print(btree->left, level + 1);
	}
}

//функция вставки элемента в рандомизированное дерево
Node* insert(Node* btree, int k) {
	if (!btree) {
		iterations++;
		//cout << "new " << iterations << endl;
		return new Node(k);
	}
	if (rand() % (btree->size + 1) == 0) {
		//cout << "Вставка в корень значения " << k << endl;
		btree = insertroot(btree, k);
		//Print(btree, 0);
		//cout << endl;
		return btree;
	}
	if (btree->key > k){
		iterations++;
		//cout << "left " << iterations << endl;
		btree->left = insert(btree->left, k);
	}
	else{
		iterations++;
		//cout << "right " << iterations << endl;
		btree->right = insert(btree->right, k);
	}
	fixsize(btree);
	return btree;
```

```
}

//функция для объеденения двух поддеревьев
Node* join(Node* btree1, Node* btree2) {
	if (!btree1) {
		iterations++;
		//cout << "no left " << iterations << endl;
		return btree2;
	}
	if (!btree2) {
		iterations++;
		//cout << "no right " << iterations << endl;
		return btree1;
	}
	if (rand() % (btree1->size + btree2->size) < btree1->size) {
		iterations++;
		//cout << "right join " << iterations << endl;
		btree1->right = join(btree1->right, btree2);
		fixsize(btree1);
		return btree1;
	}
	else {
		iterations++;
		//cout << "left join " << iterations << endl;
		btree2->left = join(btree1, btree2->left);
		fixsize(btree2);
		return btree2;
	}
}

//функция для удаления элемента из дерева
Node* remove(Node* btree, int k) {
	if (!btree) {
		iterations++;
		//cout << "this " << iterations << endl;
		return btree;
	}
	if (btree->key == k) {
		Node* newbtree = join(btree->left, btree->right);
		delete btree;
		return newbtree;
	}
	else if (k < btree->key) {
		iterations++;
```

```
            //cout << "left " << iterations << endl;
            btree->left = remove(btree->left, k);
      }
      else {
            iterations++;
            //cout << "right " << iterations << endl;
            btree->right = remove(btree->right, k);
      }
      return btree;
}

//функция генерации элементов для среднего случая
vector<int> generate_average(int n) {
      vector<int> values;
      for (int i = 0; i < n; i++) {
            int value = rand () %10000;
            values.push_back(value);
      }
      if(n > 2){
            int count1 = 0;
            int count2 = 0;
            for (int i = 0; i < n; i++) {
                  if (values[i] < values[i + 1] || values[i] == values[i +
1]) count1++;
                  if (values[i] > values[i + 1] || values[i] == values[i +
1]) count2++;
            }
            if (count2 == n  || count1 == n) generate_average(n);
//ликвидация создания худшего случая
            else return values;
      }
      return values;
}

//функция генерации элементов для худшего случая
vector<int> generate_worst(int n) {
      vector<int> values;
      for (int i = 0; i < n; i++) {
            int value = rand() % 10000;
            values.push_back(value);
      }
      int chance = rand() % 2; //устанавливается вероятность генерации по
возрастанию и убыванию
      if (chance == 0) sort(values.begin(), values.end());
```

```
	if (chance == 1) sort(values.begin(), values.end(),
greater<int>());
	return values;
}

//функция для работы со сгенерированными деревьями
void function_for_bst(string generation, Node* tree, Node* newtree) {
	srand(time(0));
	chrono::time_point<std::chrono::system_clock> start1, end1, start2,
end2, start3, end3; //используются для подсчёта времени выполнения
операций
	int value; //используется для ввода нового значения
	string operation; //используется для выбора операции
	string menu_for_value;
	while (generation == "y" || generation == "yes") {
		iterations = 0;
		cout << "Выберите действие, которое хотите осуществить:\n0.
Создание нового дерева.\n1. Поиск элемента.\n2. Вставка элемента.\n3.
Удаление элемента.\n";
		cout << "Действие: ";
		cin >> operation;
		if (operation == "0") break;
		else if (operation == "1") { //поиск элемента и анализ
			cout << "Выберите действие, которое хотите
осуществить:\n1. Выбрать значение.\n2.Сгенерировать значение.\n";
			cout << "Действие: ";
			cin >> menu_for_value;
			if (menu_for_value == "1") {
				cout << "Введите значение, которое хотите найти: ";
				cin >> value;
			}
			else value = rand() % 10000;
			start1 = chrono::system_clock::now(); //начало подсчёта
времени
			newtree = find(tree, value);
			end1 = chrono::system_clock::now(); //конец подсчёта
времени
			chrono::duration<double> duration1 = end1 - start1;
//промежуток времени работы функции
			if (!newtree) cout << "Данного элемента нет в
дереве.\n";
			else {
				if(fixsize(newtree) <= 20) Print(newtree, 0);
```

```
                    cout << endl << "Количество итераций для поиска
одного элемента: " << iterations << endl;
                    cout << "Время, затраченное на поиск одного
элемента: " << duration1.count() << "s\n";
               }
          }
          else if (operation == "2") { //добавление элемента и анализ
               cout << "Выберите действие, которое хотите
осуществить:\n1. Выбрать значение.\n2.Сгенерировать значение.\n";
               cout << "Действие: ";
               cin >> menu_for_value;
               if (menu_for_value == "1") {
                    cout << "Введите значение, которое хотите добавить:
";
                    cin >> value;
               }
               else value = rand() % 10000;
               start2 = chrono::system_clock::now(); //начало подсчёта
времени
               tree = insert(tree, value);
               end2 = chrono::system_clock::now(); //конец подсчёта
времени
               chrono::duration<double> duration2 = end2 - start2;
//промежуток времени работы функции
               cout << "****************************************\n";
               if (1 + fixsize(tree) <= 20) {
                    cout << "Новое случайное БДП с рандомизацией:" <<
endl;
                    Print(tree, 0);
               }
               cout << endl << "Количество итераций для добавления
одного элемента: " << iterations << endl;
               cout << "Время, затраченное на добавление одного
элемента: " << duration2.count() << "s\n";
          }
          else if (operation == "3") { //удаление элемента и анализ
               cout << "Выберите действие, которое хотите
осуществить:\n1. Выбрать значение.\n2.Сгенерировать значение.\n";
               cout << "Действие: ";
               cin >> menu_for_value;
               if (menu_for_value == "1") {
                    cout << "Введите значение, которое хотите удалить:
";
                    cin >> value;
```

```
                }
                else value = rand() % 10000;
                if (find(tree, value)) { //проверка на наличие элемента
в дереве
                     if (fixsize(tree) > 1) { //проверка на возможность
удаления
                          iterations = 0;
                          start3 = chrono::system_clock::now(); //начало
подсчёта времени
                          tree = remove(tree, value);
                          end3 = chrono::system_clock::now(); //конец
подсчёта времени
                          chrono::duration<double> elapsed_seconds3 =
end3 - start3; //промежуток времени работы функции
                          cout <<
"****************************************\n";
                          if (1 + fixsize(tree) <= 20) {
                               cout << "Новое случайное БДП с
рандомизацией:" << endl;
                               Print(tree, 0);
                          }
                          cout << endl << "Количество итераций для
удаления одного элемента: " << iterations << endl;
                          cout << "Время, затраченное на удаление одного
элемента: " << elapsed_seconds3.count() << "s\n";
                     }
                     else cout << endl << "Дерево состоит из одного
элемента." << endl;
                }
                else cout << "Данного элемента нет в дереве." << endl;
           }
           else {
                cout << "Действие выбрано неверно.";
           }
           cout << endl;
           cout << "Введите 'yes/y', если хотите продолжить работать с
данным БДП: ";
           cin >> generation;
           if (generation == "y" || generation == "yes") continue;
           else {
                tree = nullptr;
                iterations = 0;
           }
     }
```

```
}

//функция создания дерева для среднего случая
void func_for_researhing_average(int size, int new_iterations, string
generation, Node* tree, Node* btree, Node* newtree, vector<int>
average_case) {
	srand(time(0));
	chrono::time_point<std::chrono::system_clock> start1, end1, start2,
end2; //используются для подсчёта времени генерации деревьев
	cout << "Введите количество элементов: ";
	cin >> size;
	while (size <= 0) {
		cout << "Неправильно указано количество элементов.\nВведите
количество элементов: ";
		cin >> size;
	}
	tree = nullptr;
	average_case = generate_average(size); //формируется вектор из
случайных чисел для среднего случая
	if (size <= 100) {
		for (int i = 0; i < size; i++) {
			cout << average_case[i] << " ";
		}
	}
	cout << endl;
	start1 = chrono::system_clock::now(); //начало подсчёта времени для
сл. БДП с рандомизацией
	for (int i = 0; i < size; i++) {
		tree = insert(tree, average_case[i]);
	}
	end1 = chrono::system_clock::now(); //конец подсчёта времени для
сл. БДП с рандомизацией
	chrono::duration<double> duration1 = end1 - start1; //промежуток
времени работы функции для сл. БДП с рандомизацией
	cout << "Случайное БДП с рандомизацией в среднем случае:" << endl;
	if (size <= 20) Print(tree, 0);
	new_iterations = iterations;
	cout << endl << "Высота случайного БДП с рандомизацией: " <<
height(tree) << endl;
	cout << "Количество итераций для случайного БДП с рандомизацией: "
<< iterations << endl;
	cout << "Время, затраченное на добавление одного элемента: " <<
duration1.count() / size << "s";
	cout << endl;
```

```
	start2 = chrono::system_clock::now(); //начало подсчёта времени для сл. БДП
	for (int i = 0; i < size; i++) {
		btree = usual_insert(btree, average_case[i]);
	}
	end2 = chrono::system_clock::now(); //конец подсчёта времени для сл. БДП, которое приводится для сравнения
	chrono::duration<double> duration2 = end2 - start2; //промежуток времени работы функции для сл. БДП, которое приводится для сравнения
	cout << "Случайное БДП для сравнения:" << endl;
	if (size <= 10) Print(btree, 0);
	cout << endl << "Высота случайного БДП: " << height(btree) << endl;
	cout << "Количество итераций для случайного БДП: " << iterations - new_iterations << endl;
	cout << "Время, затраченное на добавление одного элемента: " << duration2.count() / size << "s\n";
	generation = "y";
	function_for_bst(generation, tree, newtree);
}

//функция создания дерева для худшего случая
void func_for_researching_worst(int size, int new_iterations, string generation, Node* tree, Node* btree, Node* newtree, vector<int> worst_case) {
	srand(time(0));
	chrono::time_point<std::chrono::system_clock> start1, end1, start2, end2; //используются для подсчёта времени генерации деревьев
	cout << "Введите количество элементов: ";
	cin >> size;
	while (size <= 0) {
		cout << "Неправильно указано количество элементов.\nВведите количество элементов: ";
		cin >> size;
	}
	tree = nullptr;
	worst_case = generate_worst(size); //формируется вектор из случайных чисел для худшего случая
	if (size <= 100) {
		for (int i = 0; i < size; i++) {
			cout << worst_case[i] << " ";
		}
	}
	cout << endl;
```

```
	start1 = chrono::system_clock::now(); //начало подсчёта времени для сл. БДП с рандомизацией
	for (int i = 0; i < size; i++) {
		tree = insert(tree, worst_case[i]);
	}
	end1 = chrono::system_clock::now(); //конец подсчёта времени для сл. БДП с рандомизацией
	chrono::duration<double> duration1 = end1 - start1; //промежуток времени работы функции для сл. БДП с рандомизацией
	cout << endl << "Случайное БДП с рандомизацией в худшем случае:" << endl;
	if (size <= 20) Print(tree, 0);
	new_iterations = iterations;
	cout << endl << "Высота случайного БДП с рандомизацией: " << height(tree) << endl;
	cout << "Количество итераций для случайного БДП с рандомизацией: " << iterations << endl;
	cout << "Время, затраченное на добавление одного элемента: " << duration1.count() / size << "s";
	cout << endl;
	start2 = chrono::system_clock::now(); //начало подсчёта времени для сл. БДП, которое приводится для сравнения
	for (int i = 0; i < size; i++) {
		btree = usual_insert(btree, worst_case[i]);
	}
	end2 = chrono::system_clock::now();  //конец подсчёта времени для сл. БДП, которое приводится для сравнения
	chrono::duration<double> duration2 = end2 - start2; //промежуток времени работы функции для сл. БДП, которое приводится для сравнения
	cout << "Случайное БДП для сравнения:" << endl;
	if (size <= 10) Print(btree, 0);
	cout << endl << "Высота случайного БДП: " << height(btree) << endl;
	cout << "Количество итераций для случайного БДП: " << iterations - new_iterations << endl;
	cout << "Время, затраченное на добавление одного элемента: " << duration2.count() / size << "s\n";
	cout << endl;
	generation = "y";
	function_for_bst(generation, tree, newtree);
}

int main() {
	setlocale(LC_ALL, "Russian");
	int new_iterations = 0;
```

```
	string start_end; //используется для работы программы в качестве входа/выхода
	string menu_for_generation; //используется в качестве меню по генерации деревьев
	string menu_for_creation; //используется в качестве меню по обработки созданного дерева
	string generation; //используется для работы со сгенерированным деревом
	cout << "Введите 'yes/y', если хотите начать работать с БДП: ";
	cin >> start_end;
	while (start_end == "y" || start_end =="yes"){
		int size;
		cout << "Выберите действие, которое хотите осуществить:\n";
		cout << "0. Завершить работу.\n1. Сгенерировать случайное БДП с рандомизацией и провести его анализ.\n2. Построить случайное БДП с рандомизацией.\n";
		string menu; //используется в качестве меню программы
		cout << "Действие: ";
		cin >> menu;
		vector<int> average_case;
		vector<int> worst_case;
		Node* tree = nullptr;
		Node* btree = nullptr;
		Node* newtree = nullptr;
		if(menu == "0") break;
		else if (menu == "1") { //генерация случайного БДП с рандомизацией и анализ
			menu_for_generation = "y";
			while (menu_for_generation == "y" || menu_for_generation == "yes") { //меню выбора генерации дерева
				cout << "Выберите действие, которое хотите осуществить:\n1. Генерация элементов в среднем случае.\n2. Генерация элементов в худшем случае.\n";
				cout << "Действие: ";
				string type_generation; //используется для определения типа генерации
				cin >> type_generation;
				if (type_generation == "1") func_for_researhing_average(size, new_iterations, generation, tree, btree, newtree, average_case);
				else if (type_generation == "2") func_for_researching_worst(size, new_iterations, generation, tree, btree, newtree, average_case);
				else cout << "Действие выбрано неверно.";
```

```
                    cout << endl;
                    cout << "Введите 'yes/y', если хотите продолжить
работать с генерацией БДП: ";
                    cin >> menu_for_generation;
                    if (menu_for_generation == "y" ||
menu_for_generation == "yes") {
                         btree = nullptr;
                         iterations = 0;
                    }
               }
          }
          else if (menu == "2") { //создание случайного БДП с
рандомизацией
               srand(time(0));
               cout << "Введите количество элементов: ";
               cin >> size;
               while (size <= 0) {
                    cout << "Неправильно указано количество
элементов.\nВведите количество элементов: ";
                    cin >> size;
               }
               cout << "Введите массив элементов: ";
               for (int i = 0; i < size; i++) {
                    int in;
                    cin >> in;
                    tree = insert(tree, in);
               }
               cout << "**************************************\n";
               cout << "Случайное БДП с рандомизацией:" << endl;
               Print(tree, 0);
               cout << endl;
               menu_for_creation = "y";
               while (menu_for_creation == "y" || menu_for_creation ==
"yes") { //меню выбора метода работы с деревом
                    cout << "Выберите действие, которое хотите
осуществить:\n0. Создание нового дерева.\n1. Поиск элемента.\n2. Вставка
элемента.\n3. Удаление элемента.\n";
                    cout << "Действие: ";
                    string method;
                    int value;
                    cin >> method;
                    if (method == "0") break;
                    else if (method == "1") { //поиск элемента
                         srand(time(0));
```

```
                        cout << "Введите значение, которое хотите
найти: ";
                        cin >> value;
                        cout << endl;
                        newtree = find(tree, value);
                        if (!newtree) cout << "Данного элемента нет в
дереве.\n";
                        else Print(newtree, 0);
                  }
                  else if (method == "2") { //добавление элемента
                        srand(time(0));
                        cout << "Введите значение, которое хотите
добавить: ";
                        cin >> value;
                        cout << endl;
                        tree = insert(tree, value);
                        cout <<
"****************************************\n";
                        cout << "Новое случайное БДП с рандомизацией:"
<< endl;
                        Print(tree, 0);
                  }
                  else if (method == "3") { //удаление элемента
                        srand(time(0));
                        cout << "Введите значение, которое хотите
удалить: ";
                        cin >> value;
                        if (find(tree, value)) {
                              if (fixsize(tree) >= 1) {
                                    tree = remove(tree, value);
                                    cout <<
"****************************************\n";
                                    cout << "Новое случайное БДП с
рандомизацией:" << endl;
                                    Print(tree, 0);
                              }
                        }
                        else cout << "Данного элемента нет в
дереве.\n";
                  }
                  else{
                        cout << "Действие выбрано неверно.";
                  }
                  cout << endl;
```

```
                    cout << "Введите 'yes/y', если хотите продолжить
работать с данным БДП: ";
                    cin >> menu_for_creation;
                    if (menu_for_creation == "y" || menu_for_creation
== "yes") continue;
                    else iterations = 0;
               }
          }
          else{
               cout << "Действие выбрано неверно.";
          }
          cout << endl;
          cout << "Введите 'yes/y', если хотите продолжить работать с
БДП: ";
          cin >> start_end;
          if (start_end == "y" || start_end == "yes") iterations = 0;
     }
     cout << "Работа завершена.";
     return 0;
}
```