**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: Исследование случайного бинарного дерева поиска**

Студент гр. 9303 _____ Лойконен М.Р.

Преподаватель _____ Филатов А.Ю.

Санкт-Петербург
2020

**ЗАДАНИЕ**
**НА КУРСОВУЮ РАБОТУ**

Студент Лойконен М.Р.

Группа 9303

Тема работы: исследование случайного бинарного дерева поиска

Исходные данные:
Случайные БДП – вставка и исключение. Исследование (в среднем, в худшем случае). Последовательность символов для построения БДП генерируется случайно.

Содержание пояснительной записки:
«Содержание», «Введение», «Постановка задачи», «Основные теоретические сведения», «Описание структур данных», «Тестирование», «Исследование», «Заключение», «Список использованных источников», «Программный код»

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

Студент _____ Лойконен М.Р.

Преподаватель _____ Филатов А.Ю.

# АННОТАЦИЯ

В данной работе проводилось исследование алгоритмов вставки и удаления в случайном бинарном дереве поиска. Была написана программа на языке программирования C++, в которой реализован класс случайного бинарного дерева. Для генерации входных данных были реализованы две функции — одна генерирует последовательность чисел в «среднем» случае, вторая в «худшем». Также предусмотрена возможность считать последовательность с файла. Результатами исследования являются полученные числовые сведения и графики, сопоставленные с теоретическими оценками.

# СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

Целью данной курсовой работы является реализация случайного бинарного дерева и исследование вставки и удаления элементов (в «среднем» и «худшем случаях»).

Для достижения поставленной цели требуется решить следующие задачи:

1. Изучение теоретических сведений о случайных бинарных деревьях поиска.
2. Генерация различных числовых последовательность для «средних» и «худших» случаев.
3. Накопление статистики и фиксация результатов тестирования.
4. Сопоставление полученных данных с теоретическими.

# 1. ПОСТАНОВКА ЗАДАЧИ

Реализовать структуру данных — случайное бинарное дерево поиска, и провести исследование алгоритмов вставки и удаления элементов (в «среднем» и «худшем» случаях).

# 2. ОСНОВНЫЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

## 2.1. Бинарные деревья поиска

Двоичные деревья представляют собой иерархическую структуру, в которой каждый узел имеет не более двух потомков.

Двоичное дерево упорядоченно, если для любой его вершины x справедливы такие свойства: все элементы в левом поддереве меньше элемента, хранимого в x; все элементы в правом поддереве больше элемента, хранимого в x; все элементы дерева различны.

Случайные деревья поиска представляют собой упорядоченные бинарные деревья поиска, при создании которых элементы (их ключи) вставляются в случайном порядке. Структура такого дерева полностью зависит от порядка входной последовательности.

## 2.2. Описание алгоритмов

Вставка элемента происходит следующим образом: элемент сравнивается с корнем дерева, если вставляемый элемент больше, то алгоритм рекурсивно вызывается для правого поддерева, если меньше, то алгоритм рекурсивно вызывается для левого поддерева. В конечном итоге, дойдя до пустого поддерева, элемент вставляется как лист дерева.

Удалить элемент из случайного БДП проще всего, когда он находится в листе дерева. Тогда удаляется непосредственно этот лист. Если же удаляемый лист находится во внутреннем узле b, то в ситуации, когда у него существует правое поддерево, следует найти минимальный элемент правого поддерева, рекурсивно удалить его и заменить им содержимое узла b. В ситуации, когда у

узла b нет правого поддерева, требуется найти максимальный элемент правого поддерева, рекурсивно удалить его и заменить им содержимое узла b.

## 3. ОПИСАНИЕ СТРУКТУР ДАННЫХ

В работе был реализован класс `BinSearchTree` и методы для работы с ним. Внутри класса была создана структура узла дерева `Node`, имеющая поля `data` — данные, хранящиеся в узле дерева, `shared_ptr<Node> right` и `shared_ptr<Node> left` — указатели на правое и левое поддерево соответственно. Класс имеет приватное поле `shared_ptr<Node> head` — указатель на корень дерева.

Также были реализованы следующие методы для работы с деревом:

`void add(shared_ptr<Node> &node, int el)` — приватный метод, добавляющий новый элемент со значением `el` в дерево, если элемент с таким значением уже есть в дереве, то он просто не добавляется;

`void findMin(shared_ptr<Node> node)` — приватный метод, возвращающий значение минимального элемента дерева;

`void findMax(shared_ptr<Node> node)` — приватный метод, возвращающий значение максимального элемента дерева;

`void deleteEl(shared_ptr<Node> &node, int el)` — приватный метод, удаляющий элемент со значением `el` из дерева, если такого элемента в дереве нет, то выводится соответствующее сообщение;

`void print(shared_ptr<Node> node, int color, int level)` — приватный метод, печатающий дерево в наглядном виде в консоли;

`void addElem(int el)` — публичный метод, который содержит в себе вызов метода `add` с аргументом `el`;

`void deleteElem(int el)` — публичный метод, который содержит в себе вызов метода `deleteEl` с аргументом `el`;

`bool isEmpty()` — публичный метод, который проверяет дерево на пустоту, если элементов в дереве нет, то возвращается `true`, в противном случае возвращается `false`;

`void printTree()` - публичный метод, который содержит в себе вызов метода `print`.

## 4. ТЕСТИРОВАНИЕ

На рис. 1, 2, 3 и 4 представлены примеры работы программы при вставке и удалении элементов.

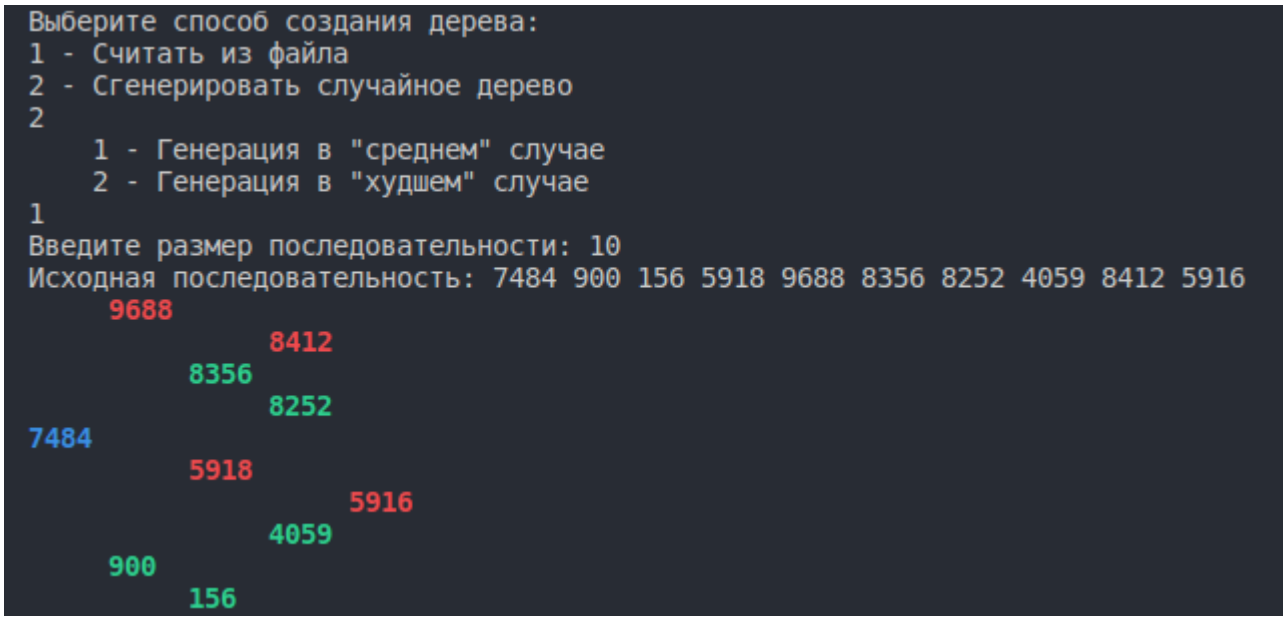


Рисунок 1 - Генерация дерева в среднем случае

```
      9688
                8412
           8356
                8252
7484
           5918
                     5916
                4059
      900
           156
1 - Добавить случайный элемент
2 - Удалить случайный элемент
1
Добавление элемента 7346
      9688
                8412
           8356
                8252
7484
                7346
           5918
                     5916
                4059
      900
           156
Время, затраченное на добавление - 0.000014 s
Количество итераций - 4
```

Рисунок 2 - Вставка элемента в среднем случае

```
Выберите способ создания дерева:
1 - Считать из файла
2 - Сгенерировать случайное дерево
2
    1 - Генерация в "среднем" случае
    2 - Генерация в "худшем" случае
1
Введите размер последовательности: 10
Исходная последовательность: 830 5311 4911 2045 4209 4251 5564 98 6732 3772
           6732
      5564
 5311
           4911
                     4251
                4209
                     3772
           2045
830
     98
1 - Добавить случайный элемент
2 - Удалить случайный элемент
2
Удаление элемента 5311
      6732
 5564
           4911
                     4251
                4209
                     3772
           2045
830
     98
Время, затраченное на удаление - 0.000006 s
Количество итераций - 4
```

Рисунок 3 - Удаление элемента в среднем случае

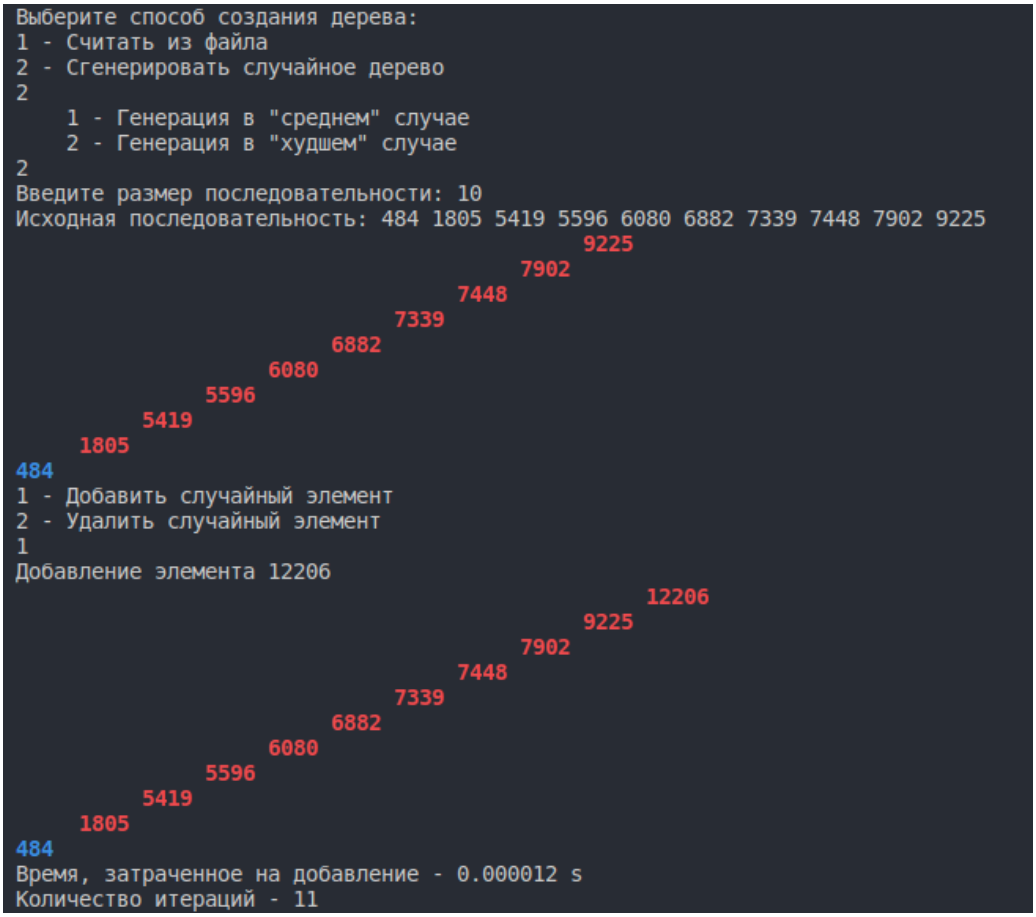


Рисунок 4 - Создание и вставка в худшем случае

# 5. ИССЛЕДОВАНИЕ

## 5.1. Генерация входных данных

Для генерации входных данных были написаны функции `generate(int n)` и `generateWorst(int n)`, которые генерируют числовые последовательности размера `n`. Притом `generateWorst(int n)` генерирует упорядоченную возрастающую последовательность. Также предусмотрена возможность считать последовательность из файла.

## 5.2. Теоретическая оценка сложности алгоритмов

В среднем случае вставка элемента в случайное БДП имеет сложность $O(\log_2 n)$ . Удаление будет иметь такую же сложность $O(\log_2 n)$ . В теории для вставки встречается более точная оценка с точностью до константы, а именно $1.386 \log_2 n$ .

Худший случай получается, когда дерево строится по упорядоченной последовательности элементов, случайное БДП будет представлять собой линейный список (визуально будет выглядеть как дерево, у которого есть только правые поддеревья в случае возрастающей последовательности). Алгоритмы вставки и удаления при этом будут иметь сложность $O(n)$ .

## 5.3. Вставка и удаление в среднем случае.

В табл. 1 представлены результаты тестирования для среднего случая.

Таблица 1 — Результаты тестирования для среднего случая

| Количество элементов n | Среднее количество итераций при добавлении | Среднее время работы программы при добавлении, мс | Среднее количество итераций при удалении | Среднее время работы программы при удалении, мс |
|---|---|---|---|---|
| 10 | 4.1 | 11 | 4.9 | 11 |
| 50 | 8.4 | 12 | 7.8 | 10 |
| 100 | 9.3 | 10 | 9 | 10 |
| 200 | 10.3 | 10 | 10.15 | 13 |
| 500 | 11.7 | 11 | 11.8 | 13 |
| 1000 | 14 | 14 | 13.2 | 13 |

Построим графики по полученным данным и сравним их с графиком $\log_2 n$ . На рис. 5 изображен полученный график для алгоритма вставки элемента.

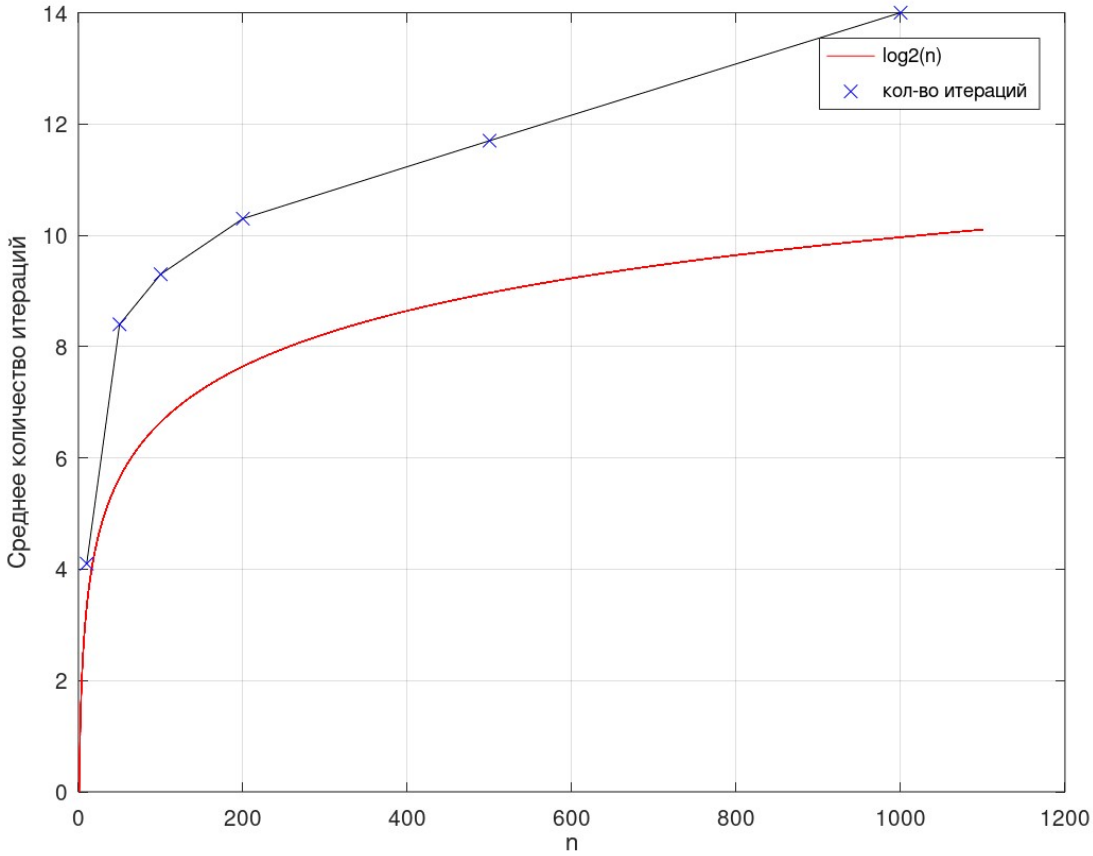


Рисунок 5 - Вставка элемента в среднем случае

Аппроксимируем график вставки функцией $1.386 \log_2 n$ . Результат представлен на рис. 6.

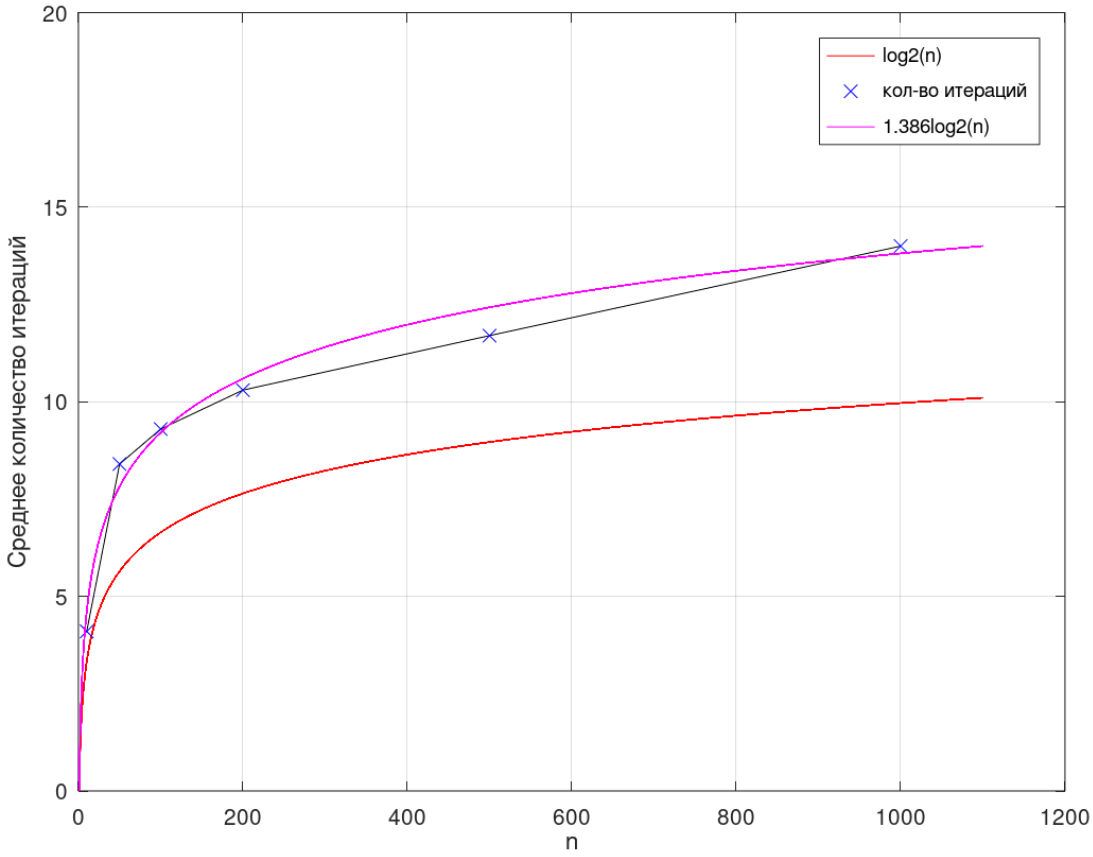


Рисунок 6 - Аппроксимация графика вставки элемента в среднем

Аналогично построим график для алгоритма удаления элемента и попытаемся его аппроксимировать функцией $1.32\log_2 n$. Графики представлены на рис. 7 и 8 соответственно.

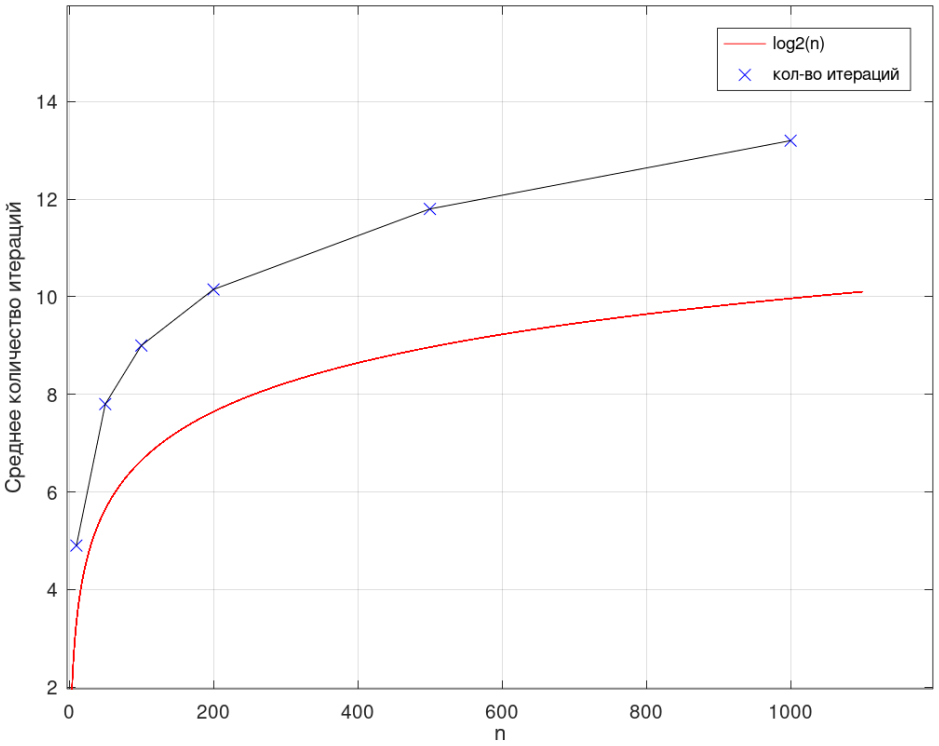


Рисунок 7 - Удаление элемента в среднем случае

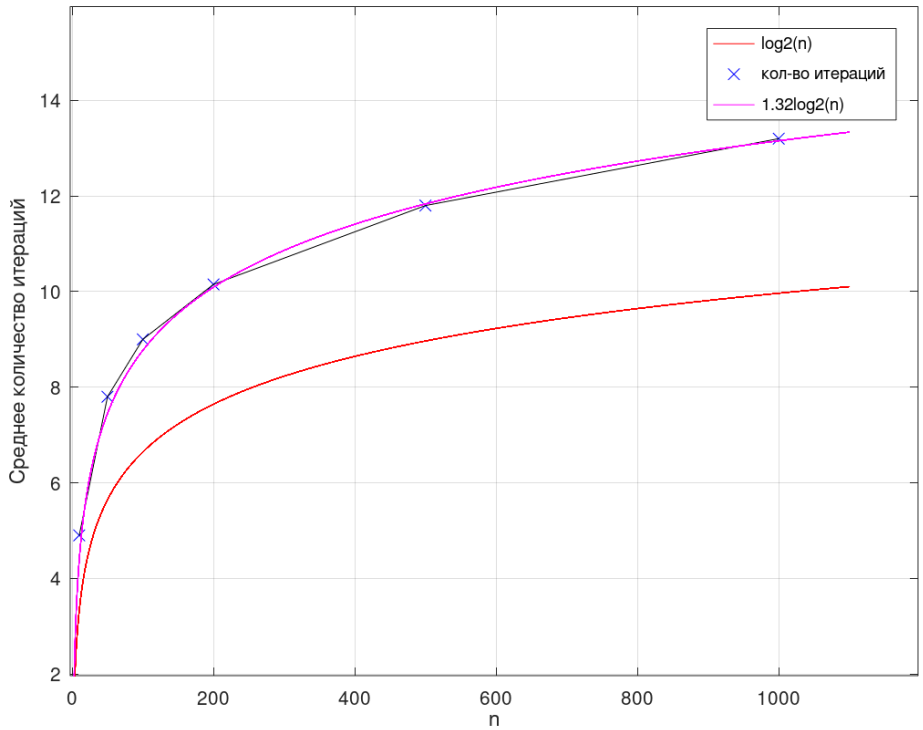


Рисунок 8 - Аппроксимация графика удаления элемента в среднем

Из построенных графиков можно заметить, что алгоритмы вставки и удаления действительно ведут себя примерно как $\log_2 n$ . Для вставки довольно точной оказалась оценка с константой $1.386\log_2 n$ . Константа перед логарифмом возникает из-за случайности исходной последовательности, но её не очень большая величина свидетельствует о том, что в среднем строятся довольно сбалансированные деревья по высоте.

**5.4. Вставка и удаление в худшем случае.**

В табл. 2 представлены результаты тестирования для худшего случая.

Таблица 2 — Результаты тестирования для худшего случая

| Количество элементов n | Среднее количество итераций при добавлении | Среднее время работы программы при добавлении, мс | Среднее количество итераций при удалении | Среднее время работы программы при удалении, мс |
|---|---|---|---|---|
| 10 | 6.6 | 7.9 | 10 | 12 |
| 50 | 29.6 | 13.1 | 50 | 28 |
| 100 | 53.2 | 17 | 100 | 47 |
| 200 | 112.5 | 22 | 200 | 74,9 |
| 500 | 296.7 | 42 | 500 | 184,3 |
| 1000 | 545.4 | 71 | 1000 | 270.2 |

Построим графики по полученным данным. На рис. 9 представлен график для вставки элемента, а на рис. 10 его аппроксимация линейной функцией. Так как значения для количества итераций брались как среднее арифметическое по проведенным тестам, может возникать погрешность, но тем не менее должна сохраниться определенная зависимость (далее будет видно, что она является линейной).

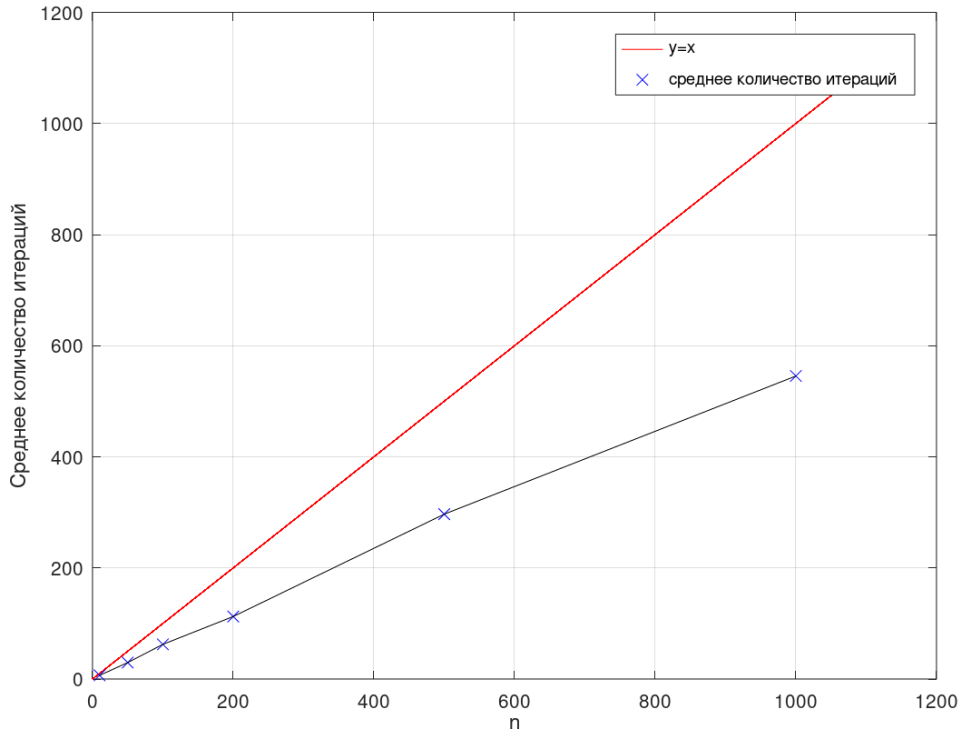


Рисунок 9 - Вставка элемента в худшем случае

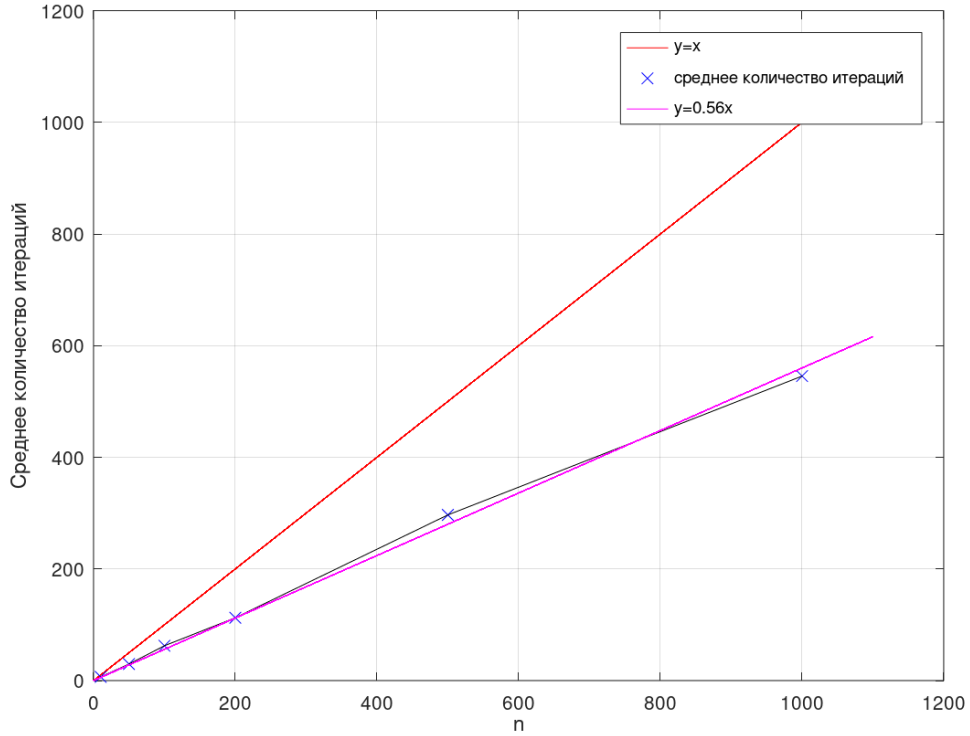


Рисунок 10 - Аппроксимация вставки элемента в худшем случае

Из полученных графиков видно, что среднее количество итераций растет линейно. В данном же случае получилось, что количество итераций приблизительно в 2 раза меньше чем количество элементов. Связано это с тем, что удаляется случайный элемент из дерева, из-за чего возникает довольно большой разброс между количеством итераций для разных тестов. Но при этом все равно сохраняется линейность.

Для удаления количество итераций будет всегда равно количеству элементов в дереве, так как после удаления элемента, мы должны рекурсивно удалить тот элемент, которым его заменили, и так далее. В результате этого процесса будут просмотрены все элементы дерева, из-за чего и возникает данная оценка. График для удаления (рис. 11) будет представлять собой прямую.

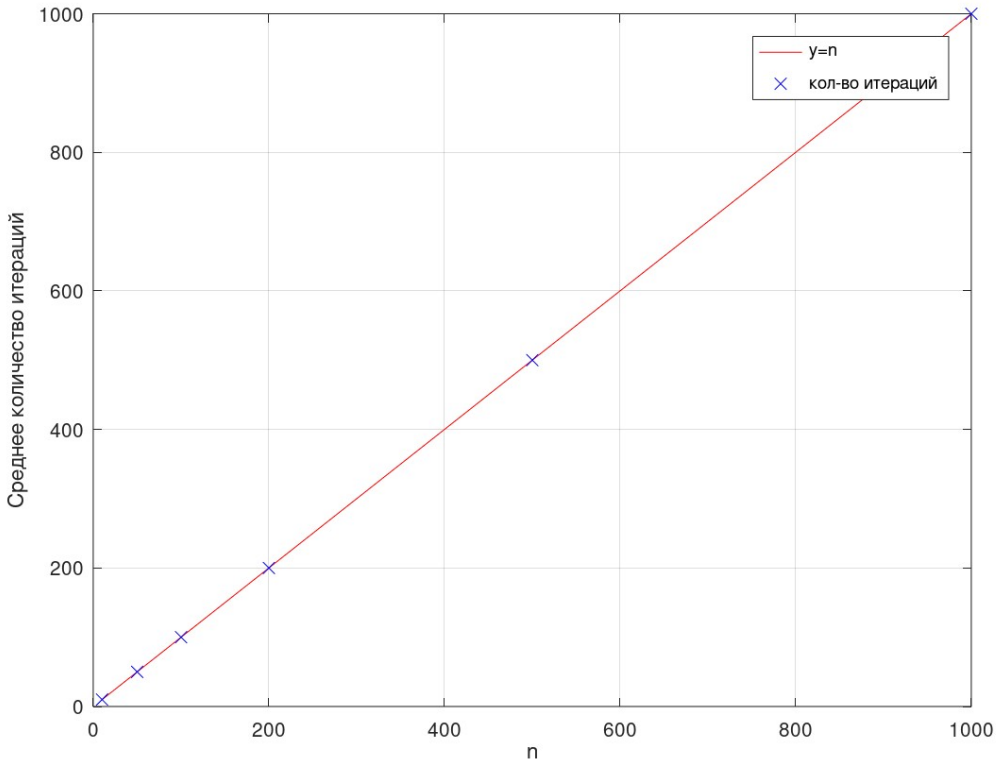


Рисунок 11 - Удаление в худшем случае

# ЗАКЛЮЧЕНИЕ

В ходе выполнения работы была написана программа, генерирующая случайные числовые последовательности и строящая по ним случайное бинарное дерево поиска.

В результате исследования были графически доказаны теоретические оценки сложности алгоритмов вставки и удаления в случайном бинарном дереве поиска в среднем и худшем случае, а именно $O(\log_2 n)$ для вставки и удаления в среднем случае и $O(n)$ для вставки и удаления в худшем случае.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Основы программирования на языках C и C++. URL: http://cppstudio.com/

2. URL: https://habr.com/ru/post/267855/

3. URL: https://tproger.ru/translations/binary-search-tree-for-beginners/

## ПРИЛОЖЕНИЕ А

## ИСХОДНЫЙ КОД ПРОГРАММЫ

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

using namespace std;

int count_iter = 0;

class BinSearchTree{
private:
     struct Node{
          int data;
          shared_ptr<Node> left;
          shared_ptr<Node> right;
          Node(int el): data(el), left(nullptr), right(nullptr) {}
     };
     void add(shared_ptr<Node> &node, int el){
          if (!node){
               count_iter++;
               node = make_shared<Node>(el);
               return;
          }
          if (el < node->data){
               count_iter++;
               add(node->left, el);
          }
          else if (el > node->data){
               count_iter++;
               add(node->right, el);
          }
     }

     int findMin(shared_ptr<Node> node){
          auto cur = node;
          while (cur->left){
               cur = cur->left;
          }
          return cur->data;
     }
```

```
int findMax(shared_ptr<Node> node){
     auto cur = node;
     while (cur->right){
          cur = cur->right;
     }
     return cur->data;
}

void deleteEl(shared_ptr<Node>& node, int el){
     if (!node){
          cout << "Данного элемента нет в дереве\n";
          return;
     }
     if (el > node->data){
          count_iter++;
          deleteEl(node->right, el);
     }
     else if (el < node->data){
          count_iter++;
          deleteEl(node->left, el);
     }
     else{
          count_iter++;
          if (node->right){
               int tmp = findMin(node->right);
               deleteEl(node->right, tmp);
               node->data = tmp;
          }
          else if (node->left){
               int tmp = findMax(node->left);
               deleteEl(node->left, tmp);
               node->data = tmp;
          }
          else{
               node = nullptr;
          }
     }
}

void print(shared_ptr<Node> node, int color, int level){
     if (node){
          print(node->right, 2, level+5);
          for (int i = 0; i<level; i++){
               cout << "\033[0;0m ";
          }
```

```
                if (color == 1)
                     cout  <<  "\033[1;34m"  <<  node->data  <<  "\
033[0;0m" << '\n';
                else if (color == 2)
                     cout  <<  "\033[1;31m"  <<  node->data  <<  "\
033[0;0m" << '\n';
                else
                     cout  <<  "\033[1;32m"  <<  node->data  <<  "\
033[0;0m" << '\n';
                print(node->left, 3, level+5);
           }
     }

     shared_ptr<Node> head;
public:
     BinSearchTree(): head(nullptr) {};
     ~BinSearchTree() = default;

     void addElem(int el){
          add(head, el);
     }

     void deleteElem(int el){
          deleteEl(head, el);
     }

     bool isEmpty(){
          if (head)
               return false;
          else
               return true;
     }

     void printTree(){
          if (head)
               print(head, 1, 0);
          else
               cout << "В дереве нет элементов";
     }

};

vector<int> generate(int n){
     int val;
     vector<int> vec;
     for (int i = 0; i<n; i++){
```

```
            val = rand()%11000;
            vec.push_back(val);
        }
        return vec;
}

vector<int> generateWorst(int n){
        int val;
        vector<int> vec;
        for (int i = 0; i<n; i++){
            val = rand()%11000;
            vec.push_back(val);
        }
        sort(vec.begin(), vec.end());
        return vec;
}

int main(){
        srand(time(0));
        string name;
        int mode;
        cout << "Выберите способ создания дерева:\n1 - Считать из
файла\n2 - Сгенерировать случайное дерево\n";
        cin >> mode;
        switch(mode){
            case 1:{
                cout << "Введите название файла: ";
                cin >> name;
                ifstream fin(name);
                BinSearchTree* bst = new BinSearchTree();
                vector<int> vec;
                int elem, m;
                int start, end;
                char opt = 'y';
                cout << "Исходная последовательность элементов: ";
                while(!fin.eof()){
                    fin >> elem;
                    cout << elem << " ";
                    bst->addElem(elem);
                    vec.push_back(elem);
                }
                cout << '\n';
                bst->printTree();

                count_iter = 0;
```

```
                cout << "1 - Добавить случайный элемент\n2 - Удалить
случайный элемент\n";
                cin >> m;
                if (m == 1){
                     int tmp = rand()%12500;
                     cout << "Добавление элемента " << tmp << '\n';
                     start = clock();
                     bst->addElem(tmp);
                     end = clock();
                     bst->printTree();
                     cout << "Время, затраченное на добавление - "
<< fixed << (double)(end-start)/CLOCKS_PER_SEC << " s\n";
                     cout << "Количество итераций - " << count_iter
<< '\n';
                }
                else{
                     int tmp = vec[rand()%vec.size()];
                     cout << "Удаление элемента " << tmp << '\n';
                     start = clock();
                     bst->deleteElem(tmp);
                     end = clock();
                     bst->printTree();
                     cout << "Время, затраченное на удаление - " <<
fixed << (double)(end-start)/CLOCKS_PER_SEC << " s\n";
                     cout << "Количество итераций - " << count_iter
<< '\n';
                }
                delete bst;
                break;
           }
           case 2:{
                int n;
                int m;
                int start, end;
                vector<int> vec;
                BinSearchTree* bst = new BinSearchTree();
                cout << " 1 - Генерация в \"среднем\" случае\n 2 -
Генерация в \"худшем\" случае\n";
                cin >> m;
                cout << "Введите размер последовательности: ";
                cin >> n;

                if (m == 1)
                     vec = generate(n);
                else
                     vec = generateWorst(n);
```

```
                cout << "Исходная последовательность: ";
                for (int i = 0; i<n; i++){
                     bst->addElem(vec[i]);
                     cout << vec[i] << " ";
                }
                cout << '\n';

                bst->printTree();
                count_iter = 0;
                cout << "1 - Добавить случайный элемент\n2 - Удалить
случайный элемент\n";
                cin >> m;
                if (m == 1){
                     int tmp = rand()%11500;
                     cout << "Добавление элемента " << tmp << '\n';
                     start = clock();
                     bst->addElem(tmp);
                     end = clock();
                     bst->printTree();
                     cout << "Время, затраченное на добавление - "
<< fixed << (double)(end-start)/CLOCKS_PER_SEC << " s\n";
                     cout << "Количество итераций - " << count_iter
<< '\n';
                }
                else{
                     int tmp = vec[rand()%n];
                     cout << "Удаление элемента " << tmp << '\n';
                     start = clock();
                     bst->deleteElem(tmp);
                     end = clock();
                     bst->printTree();
                     cout << "Время, затраченное на удаление - " <<
fixed << (double)(end-start)/CLOCKS_PER_SEC << " s\n";
                     cout << "Количество итераций - " << count_iter
<< '\n';
                }

                delete bst;
                break;
           }
     default:
          break;
     }
     return 0;
}
```