МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: Случайные бинарные деревья поиска. Исследование.

Студент гр. 7383 _____ Кирсанов А. Я.

Преподаватель _____ Размочаева Н. В.

Санкт-Петербург

2018

ЗАДАНИЕ
НА КУРСОВУЮ РАБОТУ

Студент Кирсанов А. Я.

Группа 7383

Тема работы: Случайные бинарные деревья поиска. Исследование.

Содержание пояснительной записки:

- Содержание
- Введение
- Описание функций работы с бинарными деревьями
- Примеры работы программы
- Исследование
- Заключение
- Список использованных источников

Предполагаемый объем пояснительной записки:
Не менее 15 страниц.

Дата выдачи задания:

Дата сдачи реферата:

Дата защиты реферата:

Студент _____ Кирсанов А. Я.

Преподаватель _____ Размочаева Н. В.

АННОТАЦИЯ

В работе была реализована программа на языке программирования C++ с использованием фреймворка Qt, обеспечивающая построения случайных бинарных деревьев поиска, выполняющая вставку и удаления элементов дерева. Было проведено исследование алгоритмов вставки и удаления в среднем и в худшем случае.

SUMMARY

The program was implemented in the C ++ programming language using the Qt framework, which provides for the construction of random binary search trees that performs the insertion and deletion of tree elements. A study was conducted on the insertion and deletion algorithms on average and in the worst case.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 5
ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ 6
ОПИСАНИЕ ФУНКЦИЙ РАБОТЫ СО СЛУЧАЙНЫМИ БИНАРНЫМИ ДЕРЕВЬЯМИ ПОИСКА 7
ПРИМЕРЫ РАБОТЫ ПРОГРАММЫ 8
ИССЛЕДОВАНИЕ 9
ЗАКЛЮЧЕНИЕ 14
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 15
ПРИЛОЖЕНИЕ А А ИСХОДНЫЙ КОД ПРОГРАММЫ 16

ВВЕДЕНИЕ

Целью работы является создание программы на языке C++ с использованием фреймворка Qt для работы со случайными бинарными деревьями поиска.

Для достижения цели были реализованы функции для работы с БДП: создание дерева, вставка и удаление элемента дерева по ключу, удаление дерева.

ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Бинарное дерево поиска (БДП) — это двоичное дерево, для которого выполняются следующие дополнительные условия (*свойства дерева поиска*):

- Оба поддерева — левое и правое — являются двоичными деревьями поиска.
- У всех узлов *левого* поддерева произвольного узла X значения ключей данных *меньше*, нежели значение ключа данных самого узла X.
- У всех узлов *правого* поддерева произвольного узла X значения ключей данных *больше либо равны*, нежели значение ключа данных самого узла X.

Очевидно, данные в каждом узле должны обладать ключами, на которых определена операция сравнения *меньше*.

Если структура БДП полностью зависит от того порядка, в котором элементы расположены во входной последовательности, такое БДП называется случайным.

ОПИСАНИЕ ФУНКЦИЙ РАБОТЫ СО СЛУЧАЙНЫМИ БИНАРНЫМИ ДЕРЕВЬЯМИ ПОИСКА

1. Структура бинарного дерева **node** состоит из указателей на левое и правое поддерево, целое значение узла, а также поля, в котором хранится размер (в вершинах) дерева с корнем в данном узле.
2. **Insert –** функция вставки нового элемента в дерево. Также инициализирует дерево, если в качестве указателя на дерево подается нулевой указатель. Производит вставку элемента по правилам построения бинарного дерева поиска.
3. **Remove –** функция удаления элемента из дерева. При удалении БДП перестраивает дерево таким образом, чтобы сохранялись все свойства БДП.
4. **Join –** функция объединения двух поддеревьев. Используется в работе функции **remove**.
5. Функции **getsize** и **fixsize** используются соответственно для получения размера в вершинах дерева с корнем в данном узле и для изменения размера.
6. **Destroy –** функция, удаляющая дерево. Возвращает нулевой указатель.

ПРИМЕРЫ РАБОТЫ ПРОГРАММЫ

Программа собрана в операционной системе Windows 10 в Qt 4.7.1. В других ОС и средах тестирование не проводилось.

На рисунке 1 представлен пример работы программы.

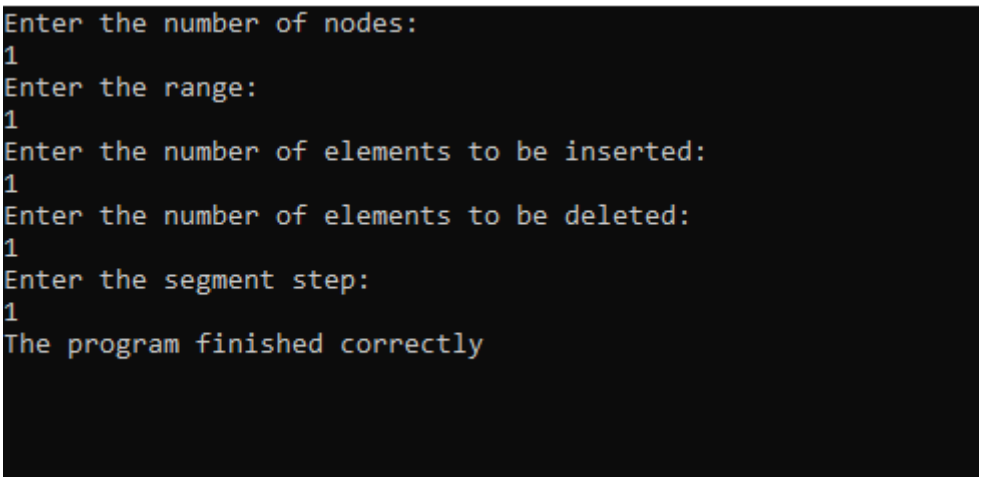

Рисунок 1 – Пример работы программы

Пользователю предлагается ввести начальное количество элементов БДП и диапазон их значений. Элементы генерируются случайным образом.

Затем программа просит ввести количество вставляемых и удаляемых элементов и шаг вывода значений функций.

На рисунке 2 представлен пример генерируемого программой файла, отражающий работу функций для 10000 начальных узлов, 100000 добавляемых и 100000 удаляемых с шагом 20000.

```
Number of el    Elapsed insertion time in seconds
20000               0.021
40000               0.05
60000               0.079
80000               0.108
100000              8.262

Number of el    Elapsed time to delete in seconds
20000               0.013
40000               0.056
60000               0.1
80000               0.179
100000              0.235
```

Рисунок 2 – Иллюстрация файла лога программы

ИССЛЕДОВАНИЕ

Для проведения исследования реализованных алгоритмов в среднем и в худшем случае была использована зависимость времени от количества вставляемых или удаляемых узлов. Предположим, что времени на вставку или удаление нужно тем больше, чем дальше от корня дерева требуется вставить или удалить элемент и чем, соответственно, больше сравнений со значением в элементе дерева требуется провести.

Время зависит от количества сравнений, значит в худшем случае времени на вставку или удаление элемента должно потребоваться больше, чем в среднем. Худший случай возникает, когда БДП вырождается в линейный список. Это происходит, когда элементы подаются последовательно в порядке возрастания или убывания. В невырожденном случае времени должно затрачиваться меньше.

Найдем соотношение времени, затраченном на добавление n элементов в среднем и в худшем случае для вставки и для удаления по отдельности. Для этого в первом случае будем подавать элементы, сгенерированные случайном образом в случайном порядке, а во втором случае будем последовательно подавать упорядоченные по возрастанию элементы. В обоих случаях создадим дерево, состоящее из одного элемента и добавим к нему 10^7 элементов с шагом 500000 элементов. Зависимость времени вставки от количества элементов показана в таблице 1.

Таблица 1 – Зависимость времени вставки от количества элементов.

В среднем случае		В худшем случае	
N	t	N	t
500000	0.795	500000	3,203
1000000	2.043	1000000	7,04
1500000	3.305	1500000	10,603
2000000	5.278	2000000	14,346
2500000	6.692	2500000	18,051
3000000	7.974	3000000	21,353
3500000	9.55	3500000	25,643

4000000	10.606	4000000	29,077
4500000	12.575	4500000	32,909
5000000	14.326	5000000	36,505
5500000	16.25	5500000	39,857
6000000	18.446	6000000	43,35
6500000	20.12	6500000	46,69
7000000	22.107	7000000	50,225
7500000	23.469	7500000	53,903
8000000	25.419	8000000	58,14
8500000	26.781	8500000	61,574
9000000	28.576	9000000	65,433
9500000	30.025	9500000	69,372

На рисунках 3 и 4 соответственно изображены графики зависимостей времени *t* вставки от количества элементов *N* в среднем и в худшем случае и их аппроксимация линейной функцией.

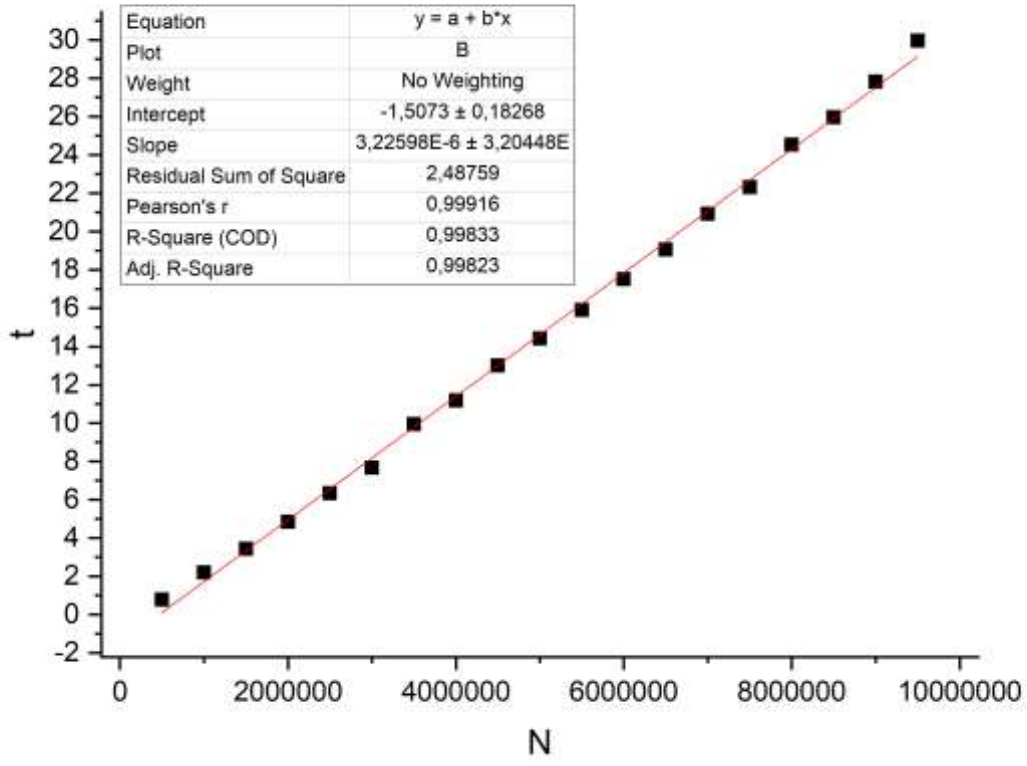

Рисунок 3 – Зависимость *t* вставки от *N* в среднем

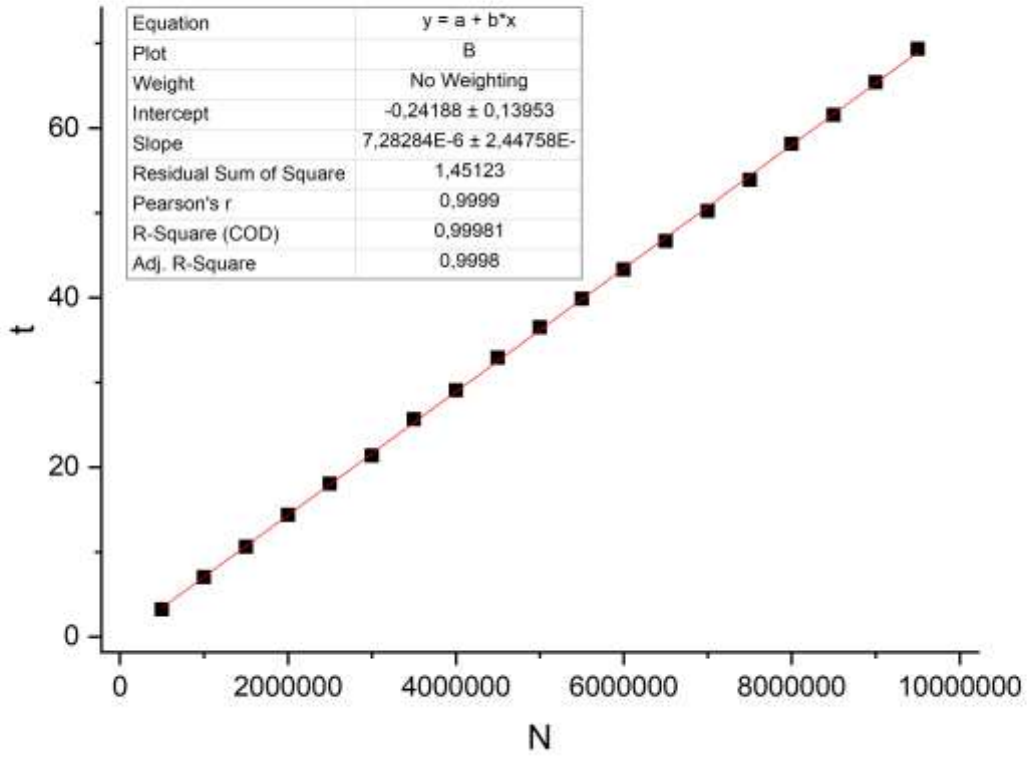

Рисунок 4 – Зависимость t вставки от N в худшем случае

Разделив тангенс угла наклона прямых в среднем и в худшем случае, получим величину $\delta_1 \approx 2{,}26$. То есть в среднем вставка в случайные БДП работают быстрее вставки в линейный список в 2,26 раз.

Аналогично для удаления элементов. Данные представлены в таблице 2.

Таблица 2 – Зависимость времени удаления от количества элементов.

В среднем случае		В худшем случае	
N	t	N	t
500000	1,9875	500000	8,0075
1000000	5,1075	1000000	17,6
1500000	8,2625	1500000	26,5075
2000000	13,195	2000000	35,865
2500000	16,73	2500000	45,1275
3000000	19,935	3000000	53,3825
3500000	23,225	3500000	64,1075
4000000	26,515	4000000	72,6925
4500000	31,4375	4500000	82,2725

5000000	35,815	5000000	91,2625
5500000	40,625	5500000	99,6425
6000000	46,115	6000000	108,375
6500000	48,921	6500000	116,725
7000000	55,2675	7000000	125,5625
7500000	58,6725	7500000	134,7575
8000000	63,5475	8000000	145,35
8500000	66,9525	8500000	153,935
9000000	71,44	9000000	163,5825
9500000	75,0625	9500000	173,43

На рисунках 5 и 6 соответственно изображены графики зависимостей времени *t* удаления от количества элементов *N* в среднем и в худшем случае и их аппроксимация линейной функцией.

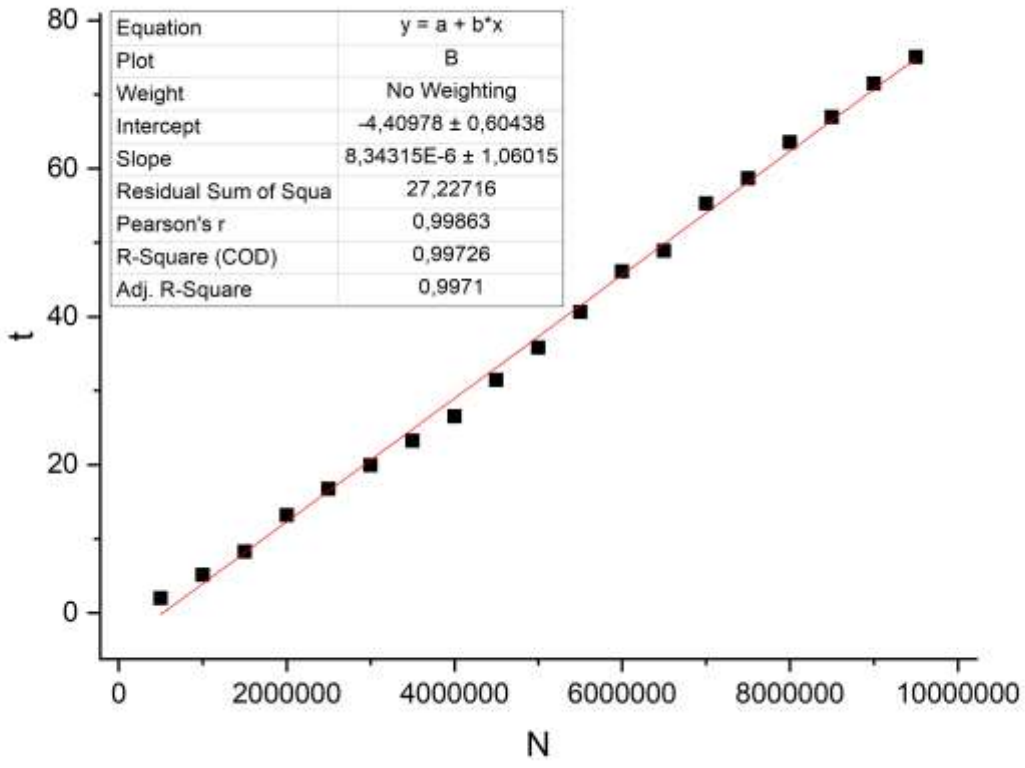

Рисунок 5 – Зависимость *t* удаления от *N* в среднем

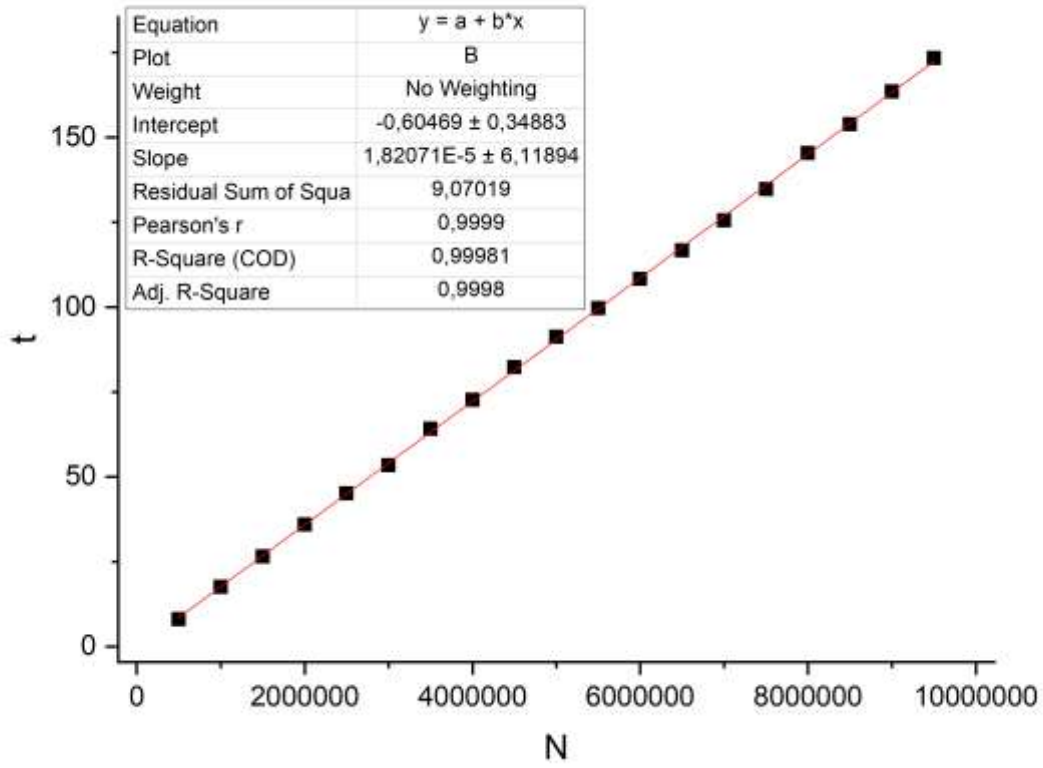

Рисунок 6 – Зависимость t удаления от N в худшем случае

Отношение тангенсов углов наклонов кривой в данном случае $\delta_2 \approx 2{,}22$.

ЗАКЛЮЧЕНИЕ

В данной работе создана программа для исследования свойств случайных бинарных деревьев. Исследовано поведение написанных функций для работы со случайными БДП в среднем и в худшем случае. В худшем случае БДП вырождается в линейный список. Выявлена линейная зависимость между временем выполнения вставки или удаления элемента в дерево и количеством элементов в дереве, также выявлена зависимость между поведением случайных БДП в среднем и в худшем случае. В худшем случае произведение операции вставки и удаления потребует в два раза больше времени, чем в среднем.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Хабр. URL: https://habr.com/post/145388/
2. Randomized Binary Search Trees. URL: http://akira.ruc.dk/~keld/teaching/algoritmedesign_f08/Artikler/03/Martinez97.pdf

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <ctime>
#include <vector>
#include <random>
#include <algorithm>
#include <iterator>
#include <iomanip>
#include <fstream>

using namespace std;

struct node // структура для представления узлов дерева
{
    int key;
    int size;
    node* left;
    node* right;
    node(int k) { key = k; left = right = 0; size = 1; }
};

int getsize(node* p) // обертка для поля size, работает с пустыми
деревьями (t=NULL)
{
    if( !p ) return 0;
    return p->size;
}

void fixsize(node* p) // установление корректного размера дерева
{
    p->size = getsize(p->left)+getsize(p->right)+1;
}

node* insert(node* p, int k) // вставка нового узла с ключом k в дерево p
{
    if( !p ) return new node(k);
    if( p->key>k )
        p->left = insert(p->left,k);
    else
        p->right = insert(p->right,k);
    fixsize(p);
    return p;
}

node* join(node* p, node* q) // объединение двух деревьев
{
    if( !p ) return q;
    if( !q ) return p;
    if( rand()%(p->size+q->size) < p->size )
```

```
    {
        p->right = join(p->right,q);
        fixsize(p);
        return p;
    }
    else
    {
        q->left = join(p,q->left);
        fixsize(q);
        return q;
    }
}

node* remove(node* p, int k) // удаление из дерева p первого найденного
узла с ключом k
{
    if( !p ) return p;
    if( p->key==k )
    {
        node* q = join(p->left,p->right);
        delete p;
        return q;
    }
    else if( k<p->key )
        p->left = remove(p->left,k);
    else
        p->right = remove(p->right,k);
    return p;
}

node* destroy(node* T){
        if (T->left)
             delete T->left;
        if (T->right)
                delete T->right;
        delete T;
        return T = NULL;
}

int main()
{
    srand(unsigned(time(0)));
    node *T = NULL;
    unsigned int count, range, ins, del, step;
    double seconds;
    ofstream file;
    file.open("output.txt");

    cout << "Enter the number of nodes:" << endl;
    cin >> count;
    cout << "Enter the range: " << endl;
```

```
    cin >> range;
    cout << "Enter the number of elements to be inserted: " << endl;
    cin >> ins;
    cout << "Enter the number of elements to be deleted: " << endl;
    cin >> del;
    cout << "Enter the segment step:" << endl;
    cin >> step;

    vector<int> x(count+ins);
    vector<int> temp(count+ins);
    for(int i = 0; i < count+ins; i++){
        x[i] = range/count*i;
    }
    random_shuffle(x.begin(), x.end());
    for(int i = 0; i < count; i++){
        T = insert(T, x[i]);
    }

    file << "Number of el\tElapsed insertion time in seconds" << endl;
    seconds = 0;
    for(int i = step; i <= ins; i+=step){
        for (int k = count+i; k < count+i+step; k++) {
            clock_t starti = clock();
            T = insert(T, x[k]);
            clock_t endi = clock();
            seconds += (double)(endi - starti) / CLOCKS_PER_SEC;
        }
        file << i << "\t\t\t\t";
        file << setprecision(5) << seconds << endl;
    }

    file << endl;
    if(del > count+ins) del = count + ins;
    file << "Number of el\tElapsed time to delete in seconds" << endl;
    seconds = 0;
    for(int i = step; i <= del; i+=step){
        for (int k = count; k < count + i; k++) {
            clock_t starti = clock();
            T = remove(T, x[k]);
            clock_t endi = clock();
            seconds += (double)(endi - starti) / CLOCKS_PER_SEC;
        }
        file << i << "\t\t\t\t";
        file << setprecision(5) << seconds << endl;
    }

    cout << "The program finished correctly";
    return 0;
}
```