# 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 деревья поиска – вставка и исключение. Исследование (в среднем и в худшем случае)

Студент гр. 7381 Кортев Ю.В.

Преподаватель Фирсов М.А.

Санкт-Петербург 2018

# ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

Студент Кортев Ю.В. Группа 7381

Тема работы: Исследование алгоритмов вставки и удаления в случайном дереве поиска Исходные данные: размер случайно генерируемого массива для исследования алгоритмов вставки и исключения бинарного дерева

Содержание пояснительной записки:

«Содержание», «Введение», «Заключение».

Дата выдачи задания: Дата сдачи реферата: Дата защиты реферата:

Студент Кортев Ю.В.

Преподаватель Фирсов М.А.

# АННОТАЦИЯ

В результаты данной курсовой работы написана программа на языке СИ, генерирующая тесты для исследования алгоритмов вставки и исключения из случайного дерева поиска. Результаты прогона алгоритмов выводятся в файл, на их основе генерируются изображения зависимости количества итераций алгоритма от размера входных данных. По сгенерированным изображениям можно эмпирически установить сложность алгоритмов вставки и исключения.

# СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы.

Исследования алгоритмов вставки и исключения из случайного дерева поиска.

Задача.

Получить экспериментальные значения скорости работы алгоритмов и сопоставить их с теорией.

Методы решения поставленных задач.

На основе сгенерированных тестов инициализируются случайные деревья поиска. После прогона алгоритмов в файл записываются экспериментальные значения, такие как время работы и число проделанных сравнений. На их основе строится зависимость экспериментальных значений от объема входных. По зависимости определяется сложность алгоритма и сопоставляется с теорией.

# СЛУЧАЙНОЕ ДЕРЕВО ПОИСКА

* 1. **Определение.**

При решении многих типов задач объем данных заранее неизвестен, но необходима такая структура данных, для которой достаточно быстро выполняются операции поиска, добавления и удаления вершин. Одно из решений этой проблемы построение *случайного дерева поиска*. При построении случайного дерева поиска данные поступают последовательно в произвольном порядке и добавление нового элемента происходит в уже имеющееся дерево.

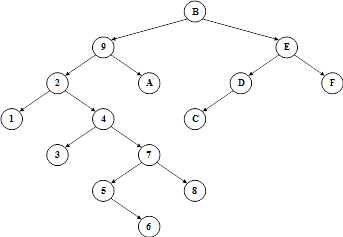


Рисунок 1 – Случайное дерево поиска B 9 2 4 1 7 E F A D C 3 5 8 6

* 1. **Алгоритмы вставки.**

Алгоритм добавления вершины в СДП заключается в следующем. Если дерево пустое, то создается корневая вершина, в которую записываются данные. В противном случае вершина добавляется к левому или правому поддереву в зависимости от результата сравнения с данными в текущей вершине.

* 1. **Алгоритм удаления.**

Алгоритм удаления вершины с некоторым ключом из случайного дерева поиска состоит в следующем. Сначала нужно найти вершину с искомым ключом. Если найденная вершина – лист, то ее просто удаляем. Если вершина имеет одно поддерево, то перекидываем указатель родительского узла на дочернее. Если же вершина имеет два поддерева, то тогда порядок действий следующий. На место удаляемой вершины ставится наибольшая вершина из левого поддерева, т.е. самая правая вершина левого поддерева, которая не имеет правого поддерева.

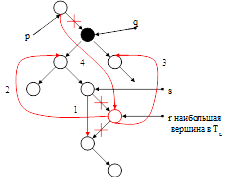


Рисунок 2 – порядок перекидывания указателей при удалении

Трудоемкость удаления вершины складывается из конечного количества операций сравнения и присваивания на каждом шаге поиска в дереве. Таким образом, трудоемкость удаления и добавления, такая же как и в случае алгоритма поиска в бинарном дереве.

# Худший случай.

В худшем случае каждый элемент входной последовательности больше предыдущего. В таком случае дерево вырождается в линейный список. Для того, чтобы обратиться к N-му элементу списка необходимо N итераций, таким образом зависимость количества итераций от объема входных данных линейная, следовательно сложность алгоритмов вставки и исключения в худшем случае О(N).

* 1. **Средний случай.**

При поиске в случайном дереве поиска среднее число сравнений

, при получаем, что сложность алгоритма составляет O(log n). Как уже было сказано выше, трудоемкость алгоритмов поиска, вставки и удаления одна и та же, таким образом сделать выводы о характеристиках алгоритмов можно сделать посчитав количества проделанных сравнений.

# Реализация алгоритмов вставки и исключения.

* + 1. **Итеративная.**

Алгоритмы вставки и удаления по своей сути являются модификациями алгоритма поиска в бинарном дереве. Возможны два типа реализации алгоритмов: итеративная и рекурсивная. В итеративном случае алгоритмы начинаются с цикла, в котором происходит сравнения добавляемого/удаляемого ключа с корневым и, в зависимости от результата, переход к левому или правому поддереву до тех пор пока, в случае добавления цикл не наткнется на нулевой указатель, в случае удаления на узел с искомым ключом. При добавлении нулевой указатель перекидывается на новый узел, при удалении указатели перекидываются по описанным выше правилам.

* + 1. **Рекурсивная**

Рекурсивный тип реализации отличается тем, что после сравнения функция алгоритма вызывается рекурсивно для правого или левого поддерева. Преимущество этого типа реализацию, лишь в ясности кода, т.е. работа алгоритма соответствует рекурсивному определению бинарного дерева поиска. В данной работе используется хвостовая рекурсия, которую компилятор gcc оптимизирует. Рекурсия в функциях находится перед выходом, из этого следует, что возвращаемое значение будет находиться в самом последнем рекурсивном вызове, “понимая” это компилятор не сохраняет адреса возврата к старым функциям на вершине стека, а лишь заменяет передаваемые значения. Из этого следует, что при использовании хвостовой рекурсии не может произойти переполнение стека, не выделяется память для новых переменных и не сохраняется адрес возврата к прошлой функции. Несмотря на это, при исследовании алгоритмов в худшем случае будет заметна разница в скорости при большом объеме входных данных. В среднем случае разница не значительна.

# Генерация множества входных данных.

* + 1. **Генерация множества для алгоритма добавления.**

На вход программе дается объем входных данных, функция TestGenAdd возвращает указатель на массиву размером с переданный объем, каждый элемент массива – неповторяющийся, со случайным значением от 1 до переданного функции значения. Сгенерированное таким образом множество удовлетворяет определению последовательности характеризующей случайное дерево поиска.

* + 1. **Генерация множества для алгоритма удаления.**

Для генерации множества, которым исследуется алгоритм удаление, берется предыдущее множество, затем каждый элемент меняется с другим случайным элементом. Затем на вход алгоритму удаления последовательно подаются элементы множества.

* + 1. **Генерация множества для алгоритма добавления в худшем случае.**

В худшем случае бинарное дерево вырождается в линейный список, для этого каждый элемент входной последовательности должен быть больше (или меньше) предыдущего. На вход функции GenWorstAdd подается объем входных данных. Последовательность начинается с единицы, заканчивается входным значением, каждый элемент последовательности больше предыдущего на единицу.

* + 1. **Генерация множества для алгоритма удаления в худшем случае**

При исследовании алгоритма удаления, на вход алгоритму подается последовательность из предыдущего пункта, но с конца.

# Описание заголовочных файлов

# gen\_tests.h

Содержит функции генерирующие описанные выше множества входных последовательностей.

* + 1. **func\_rbt.h**

Содержит функции реализации случайного дерева поиска, в том числе функции исследуемых алгоритмов add\_iter, add\_rec, delete\_iter и delete\_rec.

* + 1. **func\_graph.h**

Содержит функции генерирующие, на основе полученных экспериментальных значений, изображение зависимости количества сравнений от объема входных данных.

* 1. **Исследование алгоритмов вставки и удаления**
     1. **Получение экспериментальных значений**

Программа генерирует 4 различных теста и прогоняет на них алгоритмы, полученные экспериментальные значения, такие как количество сравнений, проделанных алгоритмом, и тики, характеризующие время работы алгоритма, записываются в таблицы, в текстовые файлы rbt\_rec.txt, rbt\_iter.txt, rbt\_worst\_iter.txt, rbt\_worst\_rec.txt.

* + 1. **Анализ экспериментальных значений.**

На основе 4х прогонов алгоритма генерируется изображение с точками, каждая точка соответствует объему прогнанных входных данных и количеству сравнений проделанных алгоритмом для данного значения. Каждому из 4х прогонов соответствует свой цвет.

* + 1. **Асимптотический анализ.**

Прогнав алгоритмы несколько раз для разных деревьев с большим объемом входных данных и получив изображение зависимости, можно сделать выводы о сложности алгоритма. Алгоритм прогнан через 4 дерева, по 1000 узлов в каждом. На рисунке отдельное дерево характеризуется отдельным цветом.

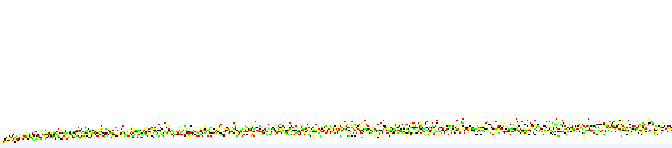


Рисунок 3 – зависимость количества сравнений от объема входных данных алгоритма вставки в случайное дерево поиска

Проанализировав рис. 3, можно сделать вывод, что зависимость логарифмическая, отсюда сложность алгоритма вставки – O(log n).

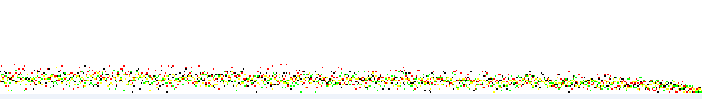


Рисунок 4 – зависимость количества сравнений от объема входных данных алгоритма удаления из случайного дерева поиска

Проанализировав рис. 4, можно сделать вывод, что зависимость логарифмическая, отсюда сложность алгоритма удаления – O(log n).

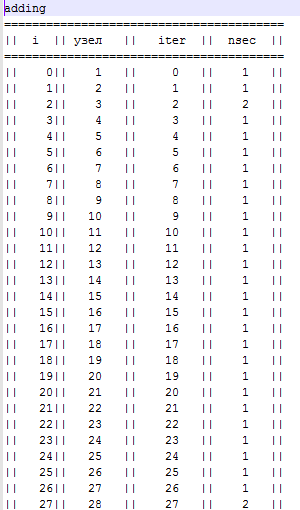


Рисунок 5 – экспериментальные значения прогона алгоритма вставки в худшем случае.

Как видно из рис. 5, зависимость количества сравнений от объема входных данных имеет линейный характер. Таким образом сложность алгоритма в худшем случае – O(n). Аналогичную сложность имеет алгоритм удаления в худшем случае.

* + 1. Сравнение с теоретическими значениями.

Как и ожидалось, полученные экспериментальным путем характеристики алгоритмов совпадают с теоретическими, полученными в пунктах **1.4.** и **1.5.**

**заключение**

# В итоге курсовой работы была эмпирически установлена сложность алгоритмов вставки и удаления в случайном дереве поиска в среднем случае – O(log n) и в худшем случае – O(n). Полученные зависимости соотносятся с теоретическими.

# Случайное дерево поиска – отличная структура данных, в тех случаях, когда нужно часто обращаться с операциями вставки или удаления, если же обращения к структуре происходят, преимущественно, с целью поиска ключей, можно воспользоваться более сбалансированными видами бинарных деревьев поиска.

# ПРИЛОЖЕНИЕ А

# ИСХОДНЫЙ КОД ГОЛОВНОЙ ФУНКЦИИ

# #include <stdlib.h>

# #include <stdio.h>

# #include <string.h>

# #include <time.h>

# #include <math.h>

# #include "gen\_tests.h"

# #include "func\_rbt.h"

# #include "func\_graph.h"

# void study\_adding(char\* fname,int count,void (\*add)(Rbt\*\*,int,int\*)){

# Rbt\* tree=NULL;

# int iter=0,time1,n=0,i;

# FILE\* ptr=fopen(fname,"w");

# if(ptr==NULL){printf("не удалось открыть файл\n"); return;}

# fputs("adding: \n",ptr);

# int compl[count];

# 

# int time=clock();

# int\* rbt=TestGenAdd(count);

# 

# fopen(res\_add,"w");

# remove(res\_add);

# for(i=0;i<4;i++){

# fprintf(ptr,TABLE\_HAT1);

# rbt=TestGenAdd(count);

# n=0;

# while(n<count){

# time1=clock();

# add(&tree,rbt[n],&iter);

# fprintf(ptr,TABLE2,n,rbt[n],iter,(int)clock()-time1);

# compl[n]=iter;

# iter=0;

# n++;

# 

# }

# fprintf(ptr,TABLE1);

# graph(compl,100,count,i,res\_add);

# free\_rbt(tree);

# tree=NULL;

# }

# }

# void study\_deleting(char\* fname,int count,void(\*delete)(Rbt\*\*,int,int\*),void (\*add)(Rbt\*\*,int,int\*)){

# int i,n,iter=0,time1;

# int\* rbt;

# int compl[count];

# FILE\* ptr=fopen(fname,"a+");

# if(ptr==NULL){printf("не удалось открыть файл\n"); return;}

# 

# Rbt\* tree=NULL;

# fopen(res\_del,"w");

# remove(res\_del);

# for(i=0;i<4;i++){

# n=0;

# rbt=TestGenAdd(count);

# while(n<count){

# add(&tree,rbt[n],&iter);

# iter=0;

# n++;

# 

# }

# TestGenDel(rbt,count);

# 

# n=0,iter=0;

# fprintf(ptr,TABLE\_HAT2);

# while(n<count){

# time1=clock();

# delete(&tree,rbt[n],&iter);

# compl[n]=iter;

# fprintf(ptr,TABLE2,n,rbt[n],iter,(int)clock()-time1);

# 

# iter=0;

# n++;

# }

# graph(compl,100,count,i,res\_del);

# fprintf(ptr,TABLE1);

# }

# }

# void study\_worst(char\* fname,void(\*delete)(Rbt\*\*,int,int\*),void (\*add)(Rbt\*\*,int,int\*)){

# int count,n,time1,iter=0;

# Rbt\* tree=NULL;

# FILE\* ptr=fopen(fname,"w");

# if(ptr==NULL){printf("не удалось открыть файл\n"); return ;}

# 

# printf("Введите кол-во узлов в худшем случае\n");

# scanf("%d",&count);

# int\* rbt=GenWorstAdd(count);

# n=0;

# 

# fprintf(ptr,TABLE\_HAT1);

# while(n<count){

# time1=clock();

# add(&tree,rbt[n],&iter);

# fprintf(ptr,TABLE2,n,rbt[n],iter,(int)clock()-time1);

# iter=0;

# n++;

# }

# fprintf(ptr,TABLE1);

# 

# fprintf(ptr,TABLE\_HAT2);

# n--;

# while(n>0){

# time1=clock();

# delete(&tree,rbt[n],&iter);

# fprintf(ptr,TABLE2,count-n-1,rbt[n],iter,(int)clock()-time1);

# iter=0;

# n--;

# }

# fprintf(ptr,TABLE1);

# fclose(ptr);

# }

# int main()

# {

# void (\*add) (Rbt\*\*,int,int\*);

# void (\*delete) (Rbt\*\*,int,int\*);

# char\* fname;

# char\* fname\_worst;

# printf("Исследовать итерационный тип реализации - 0\nРекурсивный - 1\n");

# int flag;

# scanf("%d",&flag);

# 

# if(flag==0){add=add\_iter; delete=delete\_iter;fname=f\_iter;fname\_worst=fworst\_iter;}

# if(flag==1){add=add\_rec; delete=delete\_rec;fname=f\_rec;fname\_worst=fworst\_rec;}

# if(flag!=0 && flag!=1)return 0;

# 

# 

# int count;

# printf("Введите кол-во узлов\n");

# scanf("%d",&count);

# 

# study\_adding(fname,count,add);

# 

# study\_deleting(fname,count,\*delete,add);

# 

# study\_worst(fname\_worst,\*delete,add);

# 

# return 0;

# }

**ПРИЛОЖЕНИЕ Б**

**ИСХОДНЫЙ КОД ФАЙЛА gen\_tests.c**

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <time.h>

#include "gen\_tests.h"

void swap(int\* a,int\* b){int c=\*a; \*a=\*b; \*b=c;}

int\* TestGenAdd(int count){

srand(clock());

int\* arr=(int\*)malloc(count\*sizeof(int));

int n=0,i=0,j=0;

int dup\_flag;

int rand\_val;

char fname[100];

FILE\* ptr=fopen("test\_rbt.txt","w");

if(ptr==NULL){printf("не удалось открыть файл\n"); return NULL;}

fputs("add:\n",ptr);

for ( i= 0; i<count; i++ ) {

for ( ; ; ) {

rand\_val= 1+rand()%count;

dup\_flag= 0;

for ( j= 0; j<i; j++ ) {

if ( rand\_val == arr[j] ) { dup\_flag= 1; break; }

}

if ( !dup\_flag ) { break; }

}

arr[i]= rand\_val;

fprintf(ptr,"%d ",rand\_val);

}

fputs("\n",ptr);

fclose(ptr);

return arr;

}

int\* GenWorstAdd(int count){

int\* arr=(int\*)malloc(sizeof(int)\*count);

int i=0;

for(i;i<count;i++)

arr[i]=i+1;

return arr;

}

void TestGenDel(int\* arr,int size){

int i;

FILE\* ptr=fopen("test\_rbt.txt","a+");

if(ptr==NULL){printf("не удалось открыть файл\n"); return;}

fputs("\ndel:\n",ptr);

srand(clock());

for(i=0;i<size;i++)

swap(&arr[i],&arr[rand()%size]);

for(i=0;i<size;i++)

fprintf(ptr,"%d ",arr[i]);

fputs("\n",ptr);

fclose(ptr);

return;

}

**ПРИЛОЖЕНИЕ В**

**ИСХОДНЫЙ КОД ФАЙЛА gen\_tests.h**

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

void swap(int\* ,int\* );

int\* TestGenAdd(int );

void TestGenDel(int\* ,int );

int\* GenWorstAdd(int );

**ПРИЛОЖЕНИЕ Г**

**ИСХОДНЫЙ КОД ФАЙЛА gen\_graph.c**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <string.h>**

**#include "func\_graph.h"**

**void newPIC(BMPHEADER\* bmphd,INFOHEADER\* infohd,CLR\*\* arr,char\* fname){**

**FILE\* f=fopen(fname,"w");**

**size\_t pad = 0;**

**while(((infohd->w\*3+pad) % 4)!=0)**

**pad++;**

**fwrite(bmphd,sizeof(BMPHEADER),1,f);**

**fwrite(infohd,sizeof(INFOHEADER),1,f);**

**int i,j;**

**for(i=0;i< infohd->h;i++){**

**for(j=0;j< infohd->w;j++)**

**fwrite(&arr[i][j],sizeof(CLR),1,f);**

**if(pad!=0)**

**fseek(f,pad,SEEK\_CUR);**

**}**

**fclose(f);**

**return;**

**}**

**CLR\*\* initCLR(int w){**

**int i,j;**

**CLR\*\* clr=(CLR\*\*)malloc(100 \* sizeof(CLR\*));**

**for(i=0; i < 100; i++)**

**clr[i]=(CLR\*)malloc(w \* sizeof(CLR));**

**for(i=0;i< 100;i++)**

**for(j=0;j< w; j++){**

**clr[i][j].r=255;**

**clr[i][j].g=255;**

**clr[i][j].b=255;**

**}**

**return clr;**

**}**

**CLR\*\* crtCLRS(int\* arr,int h, int w){**

**CLR\*\* clr=initCLR(w);**

**int k,j,i=0;**

**for(i=0;i<w;i++){**

**clr[arr[i]][i].r=0;**

**clr[arr[i]][i].g=0;**

**clr[arr[i]][i].b=0;**

**}**

**return clr;**

**}**

**CLR\*\* getCLRS(FILE\* f, BMPHEADER\* bmphd, INFOHEADER\* infohd){**

**size\_t pad = 0;**

**while(((infohd->w\*3+pad) % 4)!=0)**

**pad++;**

**CLR\*\* arr=(CLR\*\*)malloc((infohd->h) \* sizeof(CLR\*));**

**int i=0;**

**for(i; i < infohd->h; i++)**

**arr[i]=(CLR\*)malloc((infohd->w) \* sizeof(CLR));**

**int k,j;**

**for(i=0;i< infohd->h;i++){**

**for(j=0;j< infohd->w; j++)**

**fread(&arr[i][j],sizeof(CLR),1,f);**

**if(pad!=0)**

**fseek(f,pad,SEEK\_CUR);**

**}**

**return arr;**

**}**

**void graph(int\* arr,int h,int w,int colour,char\* fname){**

**int i,j;**

**FILE\* f;**

**BMPHEADER\* bmphd=(BMPHEADER\*)malloc(sizeof(BMPHEADER));**

**INFOHEADER\* infohd=(INFOHEADER\*)malloc(sizeof(INFOHEADER));**

**if((f=fopen(fname,"rb"))==NULL){**

**CLR\*\* clr=crtCLRS(arr,h,w);**

**f=fopen(fname,"w");**

**bmphd->type=19778;**

**bmphd->size=h\*w\*3;**

**bmphd->reserved=0;**

**bmphd->offset=54;**

**infohd->size=40;**

**infohd->w=w;**

**infohd->h=h;**

**infohd->planes=1;**

**infohd->bits=24;**

**infohd->compression=0;**

**infohd->imsize=sizeof(BMPHEADER)+sizeof(INFOHEADER)+h\*w\*3;**

**infohd->xres=0;**

**infohd->yres=0;**

**infohd->nclrs=0;**

**infohd->importantclrs=0;**

**size\_t pad = 0;**

**while(((infohd->w\*3+pad) % 4)!=0)**

**pad++;**

**fwrite(bmphd,sizeof(BMPHEADER),1,f);**

**fwrite(infohd,sizeof(INFOHEADER),1,f);**

**for(i=0;i< infohd->h;i++){**

**for(j=0;j< infohd->w;j++)**

**fwrite(&clr[i][j],sizeof(CLR),1,f);**

**if(pad!=0)**

**fseek(f,pad,SEEK\_CUR);**

**}**

**}**

**else{**

**fread(bmphd,sizeof(BMPHEADER),1,f);**

**fread(infohd,sizeof(INFOHEADER),1,f);**

**CLR\*\* clr=getCLRS(f,bmphd,infohd);**

**for(i=0;i<infohd->w;i++){**

**switch (colour){**

**case 0:**

**clr[arr[i]][i].r=0;**

**clr[arr[i]][i].g=0;**

**clr[arr[i]][i].b=255;**

**break;**

**case 1:**

**clr[arr[i]][i].r=0;**

**clr[arr[i]][i].g=255;**

**clr[arr[i]][i].b=0;**

**break;**

**case 2:**

**clr[arr[i]][i].r=255;**

**clr[arr[i]][i].g=0;**

**clr[arr[i]][i].b=0;**

**break;**

**case 3:**

**clr[arr[i]][i].r=255;**

**clr[arr[i]][i].g=255;**

**clr[arr[i]][i].b=0;**

**break;**

**}**

**}**

**newPIC(bmphd,infohd,clr,fname);**

**}**

**fclose(f);**

**return;**

**}**

**ПРИЛОЖЕНИЕ Д**

**ИСХОДНЫЙ КОД ФАЙЛА gen\_graph.h**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define res\_add "../result\_add.bmp"

#define res\_del "../result\_del.bmp"

#define res\_worst\_add "../result\_worst\_add.bmp"

#define res\_worst\_del "../result\_worst\_del.bmp"

#pragma pack(1)

typedef struct {

unsigned short type;

unsigned int size;

unsigned int reserved;

unsigned int offset;

} BMPHEADER;

typedef struct {

unsigned size;

int w,h;

unsigned short planes;

unsigned short bits;

unsigned int compression;

unsigned int imsize;

int xres,yres;

unsigned int nclrs;

unsigned int importantclrs;

} INFOHEADER;

#pragma pack(0)

typedef struct {

char b,g,r;

} CLR;

void newPIC(BMPHEADER\*,INFOHEADER\* ,CLR\*\*,char\*);

CLR\*\* getCLRS(FILE\* , BMPHEADER\* , INFOHEADER\* );

CLR\*\* initCLR(int );

CLR\*\* crtCLRS(int\* , int, int);

void graph(int\*,int,int,int ,char\*);

**ПРИЛОЖЕНИЕ Е**

**ИСХОДНЫЙ КОД ФАЙЛА func\_rbt.h**

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <time.h>

#include <math.h>

#define max(a,b) (((a) > (b)) ? (a) : (b))

#define TABLE\_HAT1 "adding\n========================================\n|| i || узел || iter || nsec ||\n========================================\n"

#define TABLE\_HAT2 "deleting\n========================================\n|| i || узел || iter || nsec ||\n========================================\n"

#define TABLE1 "========================================\n"

#define TABLE2 "||% 5d||% 5d || % 5d ||% 5d ||\n"

#define f\_rec "../rbt\_rec.txt"

#define f\_iter "../rbe\_iter.txt"

#define fworst\_rec "../rbt\_worst\_rec.txt"

#define fworst\_iter "../rbt\_worst\_iter.txt"

typedef struct node{

int key;

struct node\* left;

struct node\* right;

}Rbt;

void swap(int\* ,int\* );

void free\_rbt(Rbt\* );

int height(Rbt\* );

void add\_iter(Rbt\*\* , int ,int\*);

void add\_rec(Rbt\*\* ,int ,int\* );

void delete\_iter(Rbt\*\* ,int ,int\* );

void delete\_rec(Rbt\*\* , int ,int\* );

**ПРИЛОЖЕНИЕ Ж**

**ИСХОДНЫЙ КОД MAKEFILE**

all: main.o func\_graph.o func\_rbt.o gen\_tests.o

gcc main.o func\_graph.o func\_rbt.o gen\_tests.o -lm

main.o: main.c func\_graph.h gen\_tests.h func\_rbt.h

gcc -c main.c

func\_grapf.o: func\_graph.c

gcc -c func\_graph.c

func\_rbt.o: func\_rbt.c

gcc -c func\_rbt.c

gen\_tests.o: gen\_tests.c

gcc -c gen\_tests.c

clean:

rm main.o func\_rbt.o gen\_tests.o func\_graph.o