

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**"МИРЭА – Российский технологический университет"**

**РТУ МИРЭА**

Институт комплексной безопасности и специального приборостроения

Кафедра КБ-3 «Управление и моделирование систем»

**ОТЧЕТ   
о выполнении лабораторной работы №2**

**«Программная реализация бинарных деревьев»**

**по дисциплине   
«Программная реализация нелинейных структур»**

**Вариант № 37**

Выполнил: студент 2 курса

группы БИСО-03-19

шифр 19Б0637

Юсупов Константин Максимович  
*(фио студента)*

Проверил:

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Москва 2020 г

**Тероетическое ведение**

В информатике , оптимальное бинарное дерево поиска (Оптимальное БСТ) , иногда называемый вес сбалансирован бинарное дерево , представляет собой бинарное дерево поиска , который обеспечивает наименьшее возможное время поиска (или ожидаемое время поиска ) для данной последовательности доступов (или доступа вероятности). Оптимальные BST обычно делятся на два типа: статические и динамические.

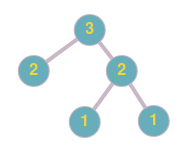
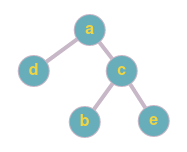
В задаче статической оптимальности дерево не может быть изменено после того, как оно было построено. В этом случае существует определенная структура узлов дерева, которая обеспечивает наименьшее ожидаемое время поиска для заданных вероятностей доступа. Существуют различные алгоритмы для построения или аппроксимации статически оптимального дерева с учетом информации о вероятностях доступа к элементам.

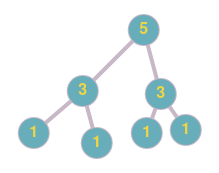
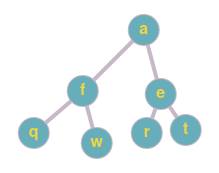
В задаче динамической оптимальности дерево можно изменить в любое время, обычно путем разрешения вращения дерева . Считается, что в дереве есть курсор, начинающийся с корня, который он может перемещать или использовать для выполнения изменений. В этом случае существует некоторая последовательность этих операций с минимальной стоимостью, которая заставляет курсор посещать каждый узел в целевой последовательности доступа по порядку. Предполагается, что расширенное дерево имеет постоянный коэффициент конкуренции по сравнению с динамически оптимальным деревом во всех случаях, хотя это еще не доказано.

**Задание на лабораторную работу № 2.**

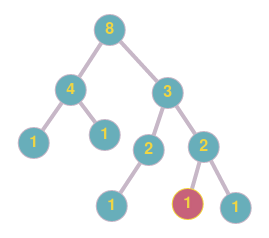
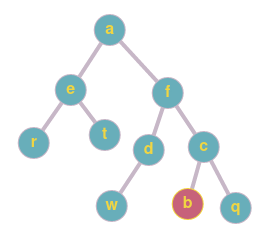
Оптимальное двоичное дерево, обход A прямой, B симметричный, список сыновей, C= A Uобр B

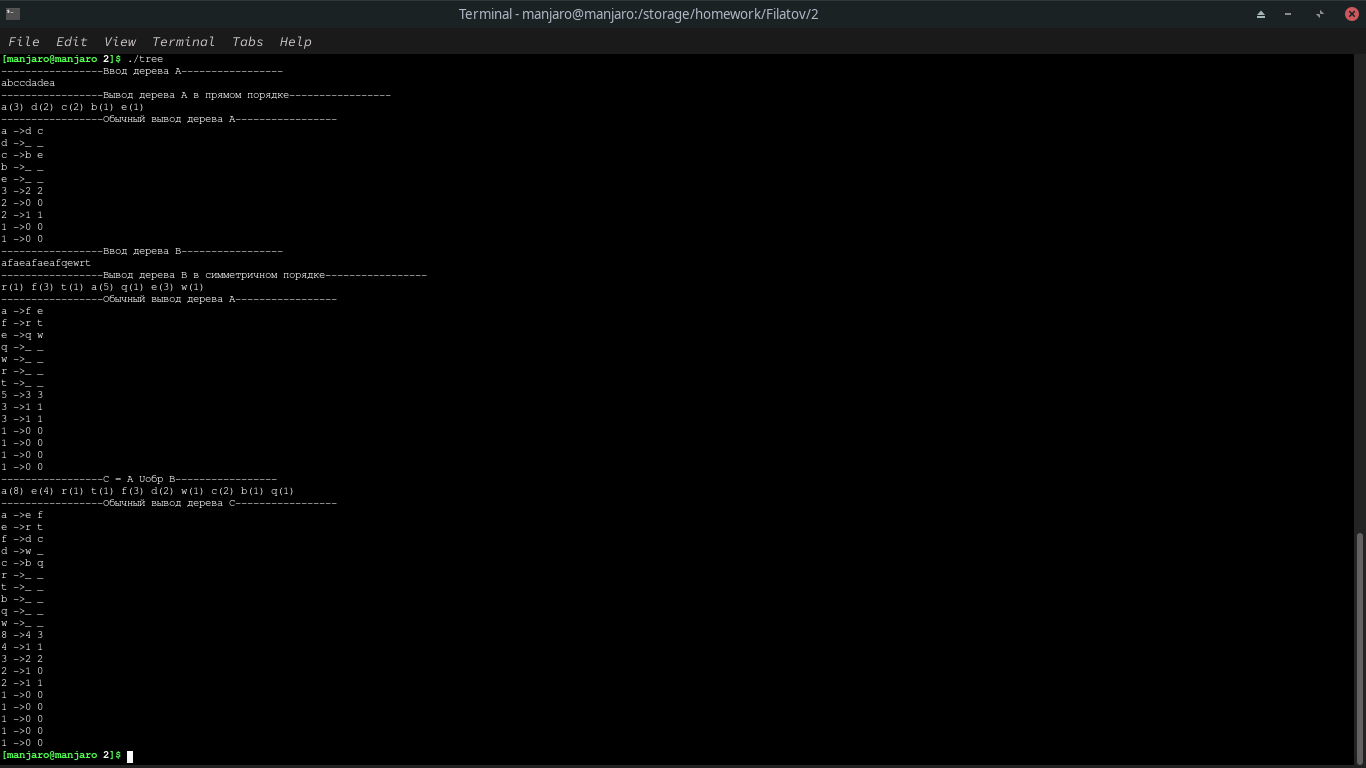
Дерево A: буквы и их количество

Дерево B: буквы и их количество

****

Дерево C: буквы и их количество

****

****

Исходный код:

Файл Makefile

all: tree

edit:

nano tree.h tree.c main.c

tree: tree.o main.o

gcc -o tree -g tree.o main.o

tree.o: tree.c

gcc -c -g tree.c

main.o: main.c

gcc -c -g main.c

clean:

rm \*.o tree

Файл tree.h#ifndef TREE\_H

#define TREE\_H

//V37 , optimal binary tree, list of children, Urev, A-straight, B-reverse

#include <stdio.h>

#include <stdlib.h>

struct tree{

unsigned long long maxcapacity;

struct node \*nodes;

unsigned long long nnodes;

};

struct node{

struct metanode \*next;

unsigned long long key;

char value;

unsigned long long layer;

unsigned long long index;

};

struct metanode{

struct metanode \*next;

struct node \*node;

};

struct returninfo{

struct node \*where;

unsigned long long layer;

};

struct tree\* tree\_new(unsigned long long maxcapacity);

void tree\_output(struct tree \*tree);

char traverse\_and\_add(struct tree \*tree, char in);

struct returninfo \*insert\_node\_rec(struct tree \*tree,struct node \*newnode, struct node \*start, char in, unsigned long long layer);

void insert\_node(struct tree \*tree, char in);

void restructure\_tree(struct tree \*tree);

void tree\_input(struct tree \*tree, char in);

void preOrderOut(struct node \*node) ;

void inOrderOut(struct node \*node) ;

void postOrderMerge(struct tree \*A, struct node \*node); //B in reverse

void Aout(struct tree \*tree);

void Bout(struct tree \*tree);

struct tree \*MergeAB(struct tree \*A, struct tree \*B);

#endif

Файл tree.c

#include "tree.h"

void Aout(struct tree \*tree){

preOrderOut(&(tree->nodes[0]));

printf("\n");

}

void Bout(struct tree \*tree){

inOrderOut(&(tree->nodes[0]));

printf("\n");

}

struct tree \*MergeAB(struct tree \*A, struct tree \*B){

struct tree \*C = tree\_new(A->nnodes+B->nnodes);

postOrderMerge(C,&(A->nodes[0]));

postOrderMerge(C,&(B->nodes[0]));

return C;

}

void preOrderOut(struct node \*node) {

printf("%c(%llu) ", node->value, node->key);

if(node->next)

preOrderOut(node->next->node);

if(node->next)

if(node->next->next)

preOrderOut(node->next->next->node);

}

void inOrderOut(struct node \*node) {

if(node->next)

inOrderOut(node->next->node);

printf("%c(%llu) ", node->value, node->key);

if(node->next)

if(node->next->next)

inOrderOut(node->next->next->node);

}

void postOrderMerge(struct tree \*A, struct node \*node) { //B in reverse

if(node->next)

postOrderMerge(A,node->next->node);

if(node->next)

if(node->next->next)

postOrderMerge(A,node->next->next->node);

for(unsigned long long i=0; i< node->key; i++)

tree\_input(A,node->value);

}

struct tree\* tree\_new(unsigned long long maxcapacity){

struct tree \*tree=calloc(sizeof(struct tree),1);

tree->maxcapacity=maxcapacity;

return tree;

}

void tree\_output(struct tree \*tree){

// for(unsigned long long i=0;i<tree ->nnodes;i++){

// printf("%p\n",&(tree->nodes[i]));

// }

for(unsigned long long i=0;i<tree ->nnodes;i++){

printf("%c ->%c %c\n",tree->nodes[i].value,(tree->nodes[i].next)?(tree->nodes[i].next->node->value):'\_', (tree->nodes[i].next)?((tree->nodes[i].next->next)?(tree->nodes[i].next->next->node->value):'\_'):'\_');

}

for(unsigned long long i=0;i<tree ->nnodes;i++){

printf("%llu ->%llu %llu\n",tree->nodes[i].key,(tree->nodes[i].next)?(tree->nodes[i].next->node->key):0, (tree->nodes[i].next)?((tree->nodes[i].next->next)?(tree->nodes[i].next->next->node->key):0):0);

}

}

char traverse\_and\_add(struct tree \*tree, char in){

for(unsigned long long i=0;i< tree->nnodes; i++){

if(tree->nodes[i].value==in){tree->nodes[i].key+=1; return 1;}

}

return 0;

}

struct returninfo \*insert\_node\_rec(struct tree \*tree,struct node \*newnode, struct node \*start, char in, unsigned long long layer){

struct returninfo \*info=calloc(sizeof(struct returninfo),1);

if(start==NULL)return NULL;

if(!start->key){

return NULL;

}

if(!start->next){

info->where=start;

info->layer=layer+1;

return info;

}

if(!start->next->next){

info->where=start;

info->layer=layer+1;

return info;

}

struct returninfo \*info1=insert\_node\_rec(tree,newnode,start->next->node,in,layer+1);

struct returninfo \*info2=insert\_node\_rec(tree,newnode,start->next->next->node,in,layer+1);

if(info1->layer< info2->layer){

info->where=info1->where;

info->layer=info1->layer;

}else{

info->where=info2->where;

info->layer=info2->layer;

}

free(info1);

free(info2);

return info;

}

void insert\_node(struct tree \*tree, char in){

tree->nnodes+=1;

tree->nodes[tree->nnodes-1].next=NULL;

tree->nodes[tree->nnodes-1].key=1;

tree->nodes[tree->nnodes-1].value=in;

tree->nodes[tree->nnodes-1].index=tree->nnodes-1;

struct returninfo \*info=insert\_node\_rec(tree,&(tree->nodes[tree->nnodes-1]),&(tree->nodes[0]),in,0);

// printf("Inserting %c, inserting into %llu, at %p at %llu layer\n",in,info->where->index,info->where,info->layer);

if(!info->where->next){

info->where->next=calloc(sizeof(struct metanode),1);

info->where->next->node=&(tree->nodes[tree->nnodes-1]);

tree->nodes[tree->nnodes-1].layer=info->layer;

}else

if(!info->where->next->next){

info->where->next->next=calloc(sizeof(struct metanode),1);

info->where->next->next->node=&(tree->nodes[tree->nnodes-1]);

tree->nodes[tree->nnodes-1].layer=info->layer;

}else{

printf("FAIL\n");

return;

}

free(info);

}

void swap(struct node \*node1, struct node \*node2){

char t=0;

unsigned long long tt=0;

t=node1->value;

node1->value=node2->value;

node2->value=t;

tt=node1->key;

node1->key=node2->key;

node2->key=tt;

tt=node1->index;

node1->index=node2->index;

node2->index=tt;

}

char recursive\_check(struct tree \*tree, struct node \*tocheck, struct node \*parent){

if(!tocheck)return 0;

if(tocheck->key>parent->key){

swap(tocheck,parent);

return 1;

}

char result=0;

if(tocheck->next){

result=(result || recursive\_check(tree,tocheck->next->node,tocheck));

if(tocheck->next->next)

result=(result || recursive\_check(tree,tocheck->next->next->node,tocheck));

}

return result;

}

char checktree(struct tree \*tree){

if(tree->nodes[0].next){

if(recursive\_check(tree,tree->nodes[0].next->node,&(tree->nodes[0])))return 1;

if(tree->nodes[0].next->next)

if(recursive\_check(tree,tree->nodes[0].next->next->node,&(tree->nodes[0])))return 1;

}

for(unsigned long long i=0;i< tree->nnodes; i++){

for(unsigned long long j=0;j< tree->nnodes; j++){

if(i==j)continue;

if(tree->nodes[i].layer< tree->nodes[j].layer)

if(tree->nodes[i].key < tree->nodes[j].key){

swap(&(tree->nodes[i]),&(tree->nodes[j]));

return 1;

}

}

}

return 0;

}

void restructure\_tree(struct tree \*tree){

while(checktree(tree)){}

}

void tree\_input(struct tree \*tree, char in){

if(!(tree->nodes)){

tree->nodes=calloc(sizeof(struct node),tree->maxcapacity);

tree->nnodes=1;

tree->nodes[0].key+=1;

tree->nodes[0].value=in;

// tree\_output(tree);

return;

}

if(traverse\_and\_add(tree,in)){

//tree\_output(tree);

restructure\_tree(tree);

//tree\_output(tree);

return;

}

// tree\_output(tree);

insert\_node(tree,in);

// tree\_output(tree);

}

Файл main.c

#include "tree.h"

#include <string.h>

#include <stdio.h>

#include <stddef.h>

int main(){

char A[1000];

printf("-----------------Ввод дерева А-----------------\n");

scanf("%s",A);

struct tree \*treeA = tree\_new(strlen(A));

for(int i=0;i<strlen(A);i++){

tree\_input(treeA,A[i]);

}

printf("-----------------Вывод дерева А в прямом порядке-----------------\n");

Aout(treeA);

printf("-----------------Обычный вывод дерева А-----------------\n");

tree\_output(treeA);

char B[1000];

printf("-----------------Ввод дерева B-----------------\n");

scanf("%s",B);

struct tree \*treeB = tree\_new(strlen(B));

for(int i=0;i<strlen(B);i++){

tree\_input(treeB,B[i]);

}

printf("-----------------Вывод дерева B в симметричном порядке-----------------\n");

Bout(treeB);

printf("-----------------Обычный вывод дерева А-----------------\n");

tree\_output(treeB);

struct tree \*C=MergeAB(treeA,treeB);

printf("-----------------C = A Uобр B-----------------\n");

Aout(C);

printf("-----------------Обычный вывод дерева C-----------------\n");

tree\_output(C);

return 0;

}