МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ»

(национальный исследовательский университет)»

Институт №3 «Системы управления, информатика и электроэнергетика»

Кафедра 304 «Вычислительные машины, системы и сети»

**Структуры и математические модели данных.**

Б дерево.

Выполнил студент группы 3О\_311Б Мишуков Никита

Проверил:

Малашкин А.В.

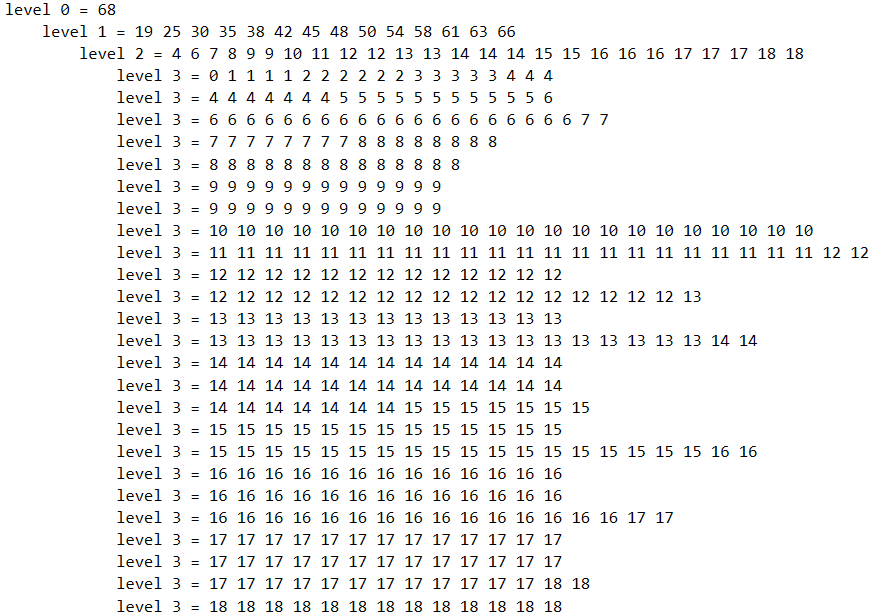
Москва 2020г.

**Структура дерева:**

Дерево является структурой в данной программе.  
Дерево имеет:  
Массив ключей Numbers(указатель на int).  
Массив потомков Leaves(двойной указатель на структуру).  
Указатель на предка Up.  
Числовую переменную-показывающую число ключей n.(наверно можно было сделать неотрицательной).  
Логическую переменную Root-показывающую является ли этот узел корнем или нет.

Глобальные переменные:  
Keys=максимальное количество ключей в узле. (2t-1)  
t= переменная относительно которой считается минимум (t-1).

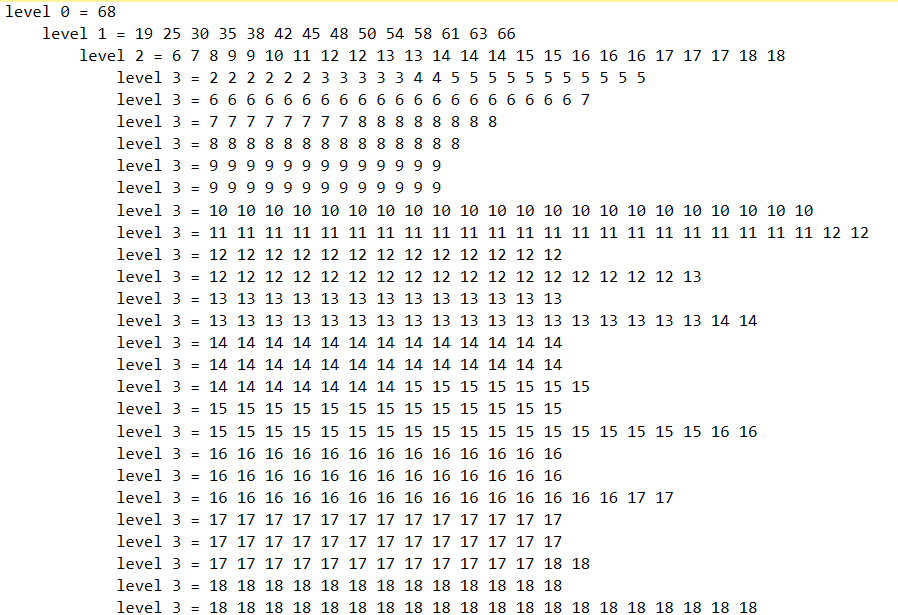
**Функции**

**Add()**-используется как при инициализации дерева так и при добавлении элемента. Пример работы add:

В данном случае распределение лежит от 0 до 100, поэтому такая насыщенность одинаковых элементов.

Add\_Inside()-часть функции Add() добавляет в выбранный узел новый элемент (с ссылкой), заменяет массив Numbers массивом с новым элементом (удаляя Numbers из памяти), заменяет массив Leaves массивом с новой ссылкой, если она не null(тут возникает тонкость, ведь при добавлении ссылка может быть слева от числа или справа, поэтому передается параметр p-который указывает где хранить ссылку слева или справа).

Divide()-деление уровня, создаем две новых ноды и заполняем, перекидываем ссылки на потомков и на родителей, текущий узел заменяем одной из созданных нод(в зависимости левый он или правый) и заносим элемент Add\_Inside(Up,Left) наверх с ссылкой на вторую ноду. В случае заполнения верхнего уровня вызываем рекурсивно.  
**Delete()-** удаляет элемент из дерева, использует в себе перевес, объединение –работает не совсем стабильно. Элемент должен лежать в дереве.



Я удалил много элементов из первого узла Level 3 и 4 из Level 2. Удалял 4 до тех пор, пока Level 3 первый и второй не объединились.

Connect()-объединяет выбранный уровень с его соседом (сложная логика для крайнего правого узла), можно указать до какого уровня надо объединять узлы и элемент, который надо убрать(при этом вместо этого элемента будет лежать ключ из верхнего уровня)

Контр пример удаления – попытка удалять элементы из корня, с какого-то момента он сломает структуру дерева, и в одном из узлов будет больше 2t-1.

Print()-вывод дерева в выходной поток.

Show()-вывод элементов дерева в порядке увеличения в выходной поток.

Read()-чтение цепочки элементов из потока.

Как вывод, работа с потоками файлов идет очень медленно, по-хорошему надо читать и выводить информацию порциями, а не по одному элементу.  
Для удаления не хватает перевеса с использованием нижних уровней. В папке с программой будут лежать два текстовых документа BeeTree с использованием математического ожидания p=0.02\*x и BiTreeRandom со случайными значениями. Так как я не делал интерфейса, код в Main будет связан с выводом одновременно двух случаев в третий файл BiTreeTree. Кроме того, я разрабатывал функции до включения их в единую структуру, поэтому Main->Add(Main, rand()% 1000000, NULL, true); двойное указание корня выглядит немножко странным, я это исправлю

**Код программы**

// BiTree.cpp : Этот файл содержит функцию "main". Здесь начинается и заканчивается выполнение программы.

//

#include "pch.h"

#include <iostream>

#include <ctime>

#include <string>

#include <fstream>

using namespace std;

const int Keys = 25;

const int t = 13;

template <typename T>

void copy(T\* buff, T\* from, int n)//просто копируем n элементов

{

for (int i = 0; i < n; i++)

{

buff[i] = from[i];

}

}

template <typename T>

void move(T\* buff, T move, int len)//просто двигаем массив на Move

{

for (int i = len - 1; i > move; i--)

{

buff[i] = buff[i - 1];

}

}

template <typename T>

struct BiTree

{

bool Root;

int n;

BiTree\* Up;

T\* Numbers;

BiTree\*\* Leaves;

void Add\_Inside(BiTree\*& place, T num, BiTree\* link = NULL, int p = 0)//p-регулирует ссылку, если p-0 левая, p-1 правая

{

T\* buff;

int Move = -1;

int i = 0;

buff = new T[place->n + 1]{ 0 };

copy(buff, place->Numbers, place->n);

place->n += 1;

for (; i < place->n; i++)//Ищем место элемента в массиве

{

if (num < buff[i])

{

Move = i;

break;

}

}

BiTree\*\* Buff\_leaves = new BiTree\*[place->n + 1];

if (Move >= 0)//находим место для ссылки и элемента

{

move<T>(buff, Move, place->n);

buff[Move] = num;

}

else

{

p = 0;

Move = place->n;

buff[place->n - 1] = num;

}

delete place->Numbers;

place->Numbers = buff;

for (int i = 0, j = 0; i < place->n + 1; i++)//Сохраняем ссылки дерева

{

if (i != Move + p)

{

Buff\_leaves[i] = place->Leaves[j];

j++;

}

else

{

Buff\_leaves[i] = link;

}

}

delete place->Leaves;

place->Leaves = Buff\_leaves;

}

BiTree\* TestNeed(BiTree\* Main, T Num)

{

int i = -1;

if (Main == NULL) return Main;

while (true)

{

i++;

if (i > Main->n) break;

if (Num < Main->Numbers[i])//если искомое число меньше, числа в узле идем по левой ссылке

{

if (Main->Leaves[i] != NULL)

{

Main = Main->Leaves[i];

i = -1;

}

}

else if (i == Main->n - 1)//если искомое число больше всех чисел, идем по самой правой ссылке

{

if (Main->Leaves[i + 1] != NULL)

{

Main = Main->Leaves[i + 1];

i = -1;

}

}

}

return Main;

}

void Divide(BiTree\*& Main, BiTree\* place, T num, BiTree\* link = NULL, int p = 0)

{

BiTree\* New = place->Up;//Верхний уровень

BiTree\* Left = NULL;

BiTree\* Right = NULL;

BiTree\* Root = NULL;

Add\_Inside(place, num, link, p);//добавил 07.04

if (place->Root)

{

Add(Root, place->Numbers[place->n / 2], NULL, true);

}

int i = 0;

for (; i < place->n / 2; i++)//сохраняем ссылки и новых родителей

{

if (i != 0)

Add\_Inside(Left, place->Numbers[i], place->Leaves[i + 1], 1);

else Add(Left, place->Numbers[i], NULL, false, place->Leaves[i], place->Leaves[i + 1]);

if (place->Leaves[i] != NULL)

{

if (i!= 0) place->Leaves[i + 1]->Up = Left;

else

{

place->Leaves[i]->Up= Left;

place->Leaves[i+1]->Up= Left;

}

}

}

for (i += 1; i < place->n; i++)//тоже что и сверху

{

if (i != place->n / 2 + 1)

Add\_Inside(Right, place->Numbers[i], place->Leaves[i + 1], 1);

else Add(Right, place->Numbers[i], NULL, false, place->Leaves[i], place->Leaves[i + 1]);

if (place->Leaves[i] != NULL)

{

if (i!= place->n / 2 + 1) place->Leaves[i + 1]->Up = Right;

else

{

place->Leaves[i]->Up = Right;

place->Leaves[i + 1]->Up = Right;

}

}

}

Left->Up = place->Up;

Right->Up = place->Up;

if (place->Root)//новый корень

{

Root->Leaves[0] = Left;

Root->Leaves[1] = Right;//линки для корня

Left->Up = Root;

Right->Up = Root;

Main = Root;//

return;

}

if (place->Numbers[place->n / 2] < New->Numbers[0])//сохраняем разбиение

{

if ((New->n) < Keys) Add\_Inside(place->Up, place->Numbers[place->n / 2], Left);//просто кидаем элемент наверх

else Divide(Main, place->Up, place->Numbers[place->n / 2], Left);//рекурсия если верхнтй уровень заполнен

i = place->n / 2 - 1;//я очень хотел от этого избавиться, но всегда выходило боком

for (i += 1; i < place->n; i++)

{

if (place->Leaves[i] != NULL)

{

if (i != place->n / 2 + 1) place->Leaves[i + 1]->Up = place;

else

{

place->Leaves[i]->Up = place;

place->Leaves[i + 1]->Up = place;

}

}

}

\*place = \*Right;

place->Up = Left->Up;

delete Right, Right->Leaves, Right->Numbers;

return;

}

else//тоже что и сверху, просто тепереь мы передаем элемент как правуюю ссылку

{

if ((New->n) < Keys) Add\_Inside(place->Up, place->Numbers[place->n / 2], Right, 1);

else Divide(Main, place->Up, place->Numbers[place->n / 2], Right, 1);

i =0;

for (;i < place->n / 2; i++)

{

if (place->Leaves[i] != NULL)

{

if (i != 0) place->Leaves[i + 1]->Up = place;

else

{

place->Leaves[i]->Up = place;

place->Leaves[i + 1]->Up = place;

}

}

}

\*place = \*Left;//тоже самое что сверху, просто меняем местами линк и наше место

place->Up = Right->Up;

delete Left, Left->Leaves, Left->Numbers;

return;

}

}

void Add(BiTree\*& Main, T num, BiTree\* link = NULL, bool Root = false, BiTree\* Left = NULL, BiTree\* Right = NULL)

{//список параметров - корень, число для ввода, ссылка - если это число уже связано с листом из низа дерева, корень или нет, две ссылки для инициализации- используется при разбиении корня

BiTree\* place = TestNeed(Main, num);

if (Main == NULL)//инициализация

{

Main = new BiTree;

Main->n = 1;

Main->Root = Root;

Main->Up = NULL;

Main->Numbers = new T[1]{ num };

Main->Leaves = new BiTree\*[2]{ Left,Right };

return;

}

else if ((place->n < Keys))//Если в ячейке дерева меньше элементов чем в нашей ширине выборки

{//функция добавления, по факту здесь лишняя и надо вынести как отдельную, работает стабильно ссылки обнуляет тоже

Add\_Inside(place, num);

}

else if (place->n >= Keys)//деление

{

Divide(Main, place, num);

}

}

void Print(BiTree\* Main, std::ostream& streamOut, int p = 0)//функция вывода

{

if (Main != NULL)

{

for (int j = 0; j < p; j++)

{

streamOut << "\t";

}

streamOut <<"level " << p << " = ";

for (int i = 0; i < Main->n; i++)

{

streamOut << Main->Numbers[i] << " ";

}

streamOut << endl;

}

for (int i = 0; i < Main->n + 1; i++)

{

if (Main->Leaves[i] != NULL)

{

Print(Main->Leaves[i], streamOut, p + 1);

}

}

}

BiTree\* ThisLevel(BiTree\* Main, T Num)//находим узел с данным числом

{

int i = -1;

while (true)

{

i++;

if (Main->Numbers[i] == Num)

{

return Main;

}

if (Num < Main->Numbers[i])//если искомое число меньше, числа в узле идем по левой ссылке

{

if (Main->Leaves[i] != NULL)

{

Main = Main->Leaves[i];

i = -1;

}

}

else if (i == Main->n - 1)//если искомое число больше всех чисел, идем по самой правой ссылке

{

if (Main->Leaves[i + 1] != NULL)

{

Main = Main->Leaves[i + 1];

i = -1;

}

}

}

return Main;

}

void Find(BiTree\* Main, T num, int p = 0)//функция показывает на каком уровне элемент

{

if (Main != NULL)

{

for (int i = 0; i < Main->n; i++)

{

if (num == Main->Numbers[i])

{

cout << "It's here on level=" << p << endl;

return;

}

}

}

for (int i = 0; i < Main->n + 1; i++)

{

if (Main->Leaves[i] != NULL)

{

Find(Main->Leaves[i], num, p + 1);

}

}

}

void Connect(BiTree\*& Main, BiTree\*& Del, BiTree\* loop = NULL, T num = NULL)//написать самому себе комментарий что тут происходит

{//входные данные корень, ссылка на объединяемый уровень, ссылка на уровень до которого обменяем, элемент -если его нужно затереть

BiTree\* Up = Del->Up;//ищем верхний уровень

int j;

if (Up == NULL)//верхнего уровня нет -> меняем корневое значение

{

BiTree\* buff = Main;

Main = Main->Leaves[0];// у нас новый корень лежит в 0 узле,

delete buff;//очищаем память

return;

}

for (j = 0; j < Up->n + 1; j++)//ищем номер нашего узла

{

if (Up->Leaves[j] == Del)// j текущий узел

break;

}

int p = 0;

if (j == Up->n)//это дополнительная переменная необходимая для моделирования самого правого узла(его обработка выходит из правил)

{

p = 2;

}

BiTree\* Seed = NULL;//пересобираем текущий узел

int i;

int shift = 0;//дополнительная переменная учавствует в случае когда нам надо пересобирать дерево(теряем один уровень),используется когда воруем ссылку на потомка у соседнего узла чтобы передавать ссылки начиная со 2

for (i = 0; i < Del->n; i++)

{

if (Del->Numbers[i] != num)

{

if (Seed != NULL)

Add\_Inside(Seed, Del->Numbers[i], Del->Leaves[i + 1], 1);//добавление

else

{

if (Del->Leaves[i + 1] == NULL)//показывает что дерево пересобирается, структура нарушена

{

if (p)

Add(Seed, Del->Numbers[i], NULL, false, Up->Leaves[j - 1]->Leaves[Up->Leaves[j - 1]->n], Del->Leaves[i]);//передаем крайнего потомка левого узла, чтобы не потерять и текущего потомка

else

{

Add(Seed, Del->Numbers[i], NULL, false, Del->Leaves[i], Up->Leaves[j + 1]->Leaves[0]);//обратный случай когда мы находимся справа

shift++;//так как мы забрали у левого узла 0 элемент надо начинать с 1

}

}

else Add(Seed, Del->Numbers[i], NULL, false, Del->Leaves[i], Del->Leaves[i + 1]);//обыкновенное добавление если узел не инициализирован

}

}

else

{

if (Seed != NULL)//все тоже самое но с использованием p

Add\_Inside(Seed, Up->Numbers[j - p / 2]);//так как p=2 значит у нас самый крайний узел, по длине массив узлов на 1 больше массива чисел, поэтому вычитаем 1

else

{

if (Del->Leaves[i + 1] == NULL)//делаем все тоже самое что выше но для крайнего правго узла

{

if (p) Add(Seed, Up->Numbers[j - p / 2], NULL, false, Up->Leaves[j - 1]->Leaves[Up->Leaves[j - 1]->n], Del->Leaves[i]);

else

{

Add(Seed, Up->Numbers[j - p / 2], NULL, false, Del->Leaves[i], Up->Leaves[j + 1]->Leaves[0]);

shift++;

}

}

else Add(Seed, Up->Numbers[j - p / 2], NULL, false, Del->Leaves[i], Del->Leaves[i + 1]);

}

}

}

for (i = 0; i < Up->Leaves[j + 1 - p]->n; i++)//добавляем элементы соседа

{

Add\_Inside(Seed, Up->Leaves[j + 1 - p]->Numbers[i], Up->Leaves[j + 1 - p]->Leaves[i + shift], 0);

}

BiTree\*\* buff = new BiTree\*[Up->n]{ NULL };//нам нужно будет пересобрать ссылки верхнего уровня, так как два узла должны слиться-> их ссылки надо отбросить

bool Write = true;//показатель того что ссылка на соединение узлов сохранена

for (int i = 0, k = 0; i < Up->n + 1; i++)

{

if ((i != j) && (i != j + 1 - p))//сохраняем ссылки все кроме текущей и следующей, где p показывает на соседа слева, в случае если мы находимся в крайнем узле

{

buff[k] = Up->Leaves[i];

k++;

}

else

{

if (Write)//сохранение

{

buff[k] = Seed;

k++;

Write = false;

}

}

}

delete Up->Leaves[j + 1 - p];

Up->Leaves = buff;

Up->Leaves[Up->n] = NULL;

delete Del;

if (loop != NULL)

{

if (Up == loop) return;

else

{

Connect(Main, Up, loop);

return;

}

}

else if (Up->n <= t - 1) Connect(Main, Up, NULL, Up->Numbers[0]);//нужно пересобрать дерево

else

{

Remove(Up->Numbers, Up->Numbers[j - p / 2], Up->n);//просто удаляем элемент

Up->n--;

}

return;

}

void Remove(T\*& Numbers, T num, int n)//наверно я мог это не делать и найти библиотечный вариант лучше)

{

T\* buff = new T[n - 1];

bool Removed = false;

for (int i = 0, j = 0; i < n; i++)

{

if ((Numbers[i] != num) || (Removed))

{

buff[j] = Numbers[i];

j++;

}

else

{

Removed = true;

}

}

delete Numbers;

Numbers = buff;

}

void RemoveLinks(BiTree\*\*& Links, BiTree\* link, int n)//наверно я мог это не делать и найти библиотечный вариант лучше)

{

BiTree\*\* buff = new BiTree\*[n];

bool Removed = false;

for (int i = 0, j = 0; i < n + 1; i++)

{

if ((Links[i] != link) || (Removed))

{

buff[j] = Links[i];

j++;

}

else

{

Removed = true;

}

}

\*Links = \*buff;

}

void Delete(BiTree\*& Main, T num)//удаление

{

BiTree\* Del;

BiTree\* Place;

Del = ThisLevel(Main, num);//находим узел

int i;

for (i = 0; i < Del->n; i++)

{

if (num == Del->Numbers[i])

break;

}

T DelNum;

if ((Del->Leaves[i] == NULL) && (Del->n > t - 1))//просто стираем элемент, если это узел в конце дерева

{

Remove(Del->Numbers, num, Del->n);//простое удаление элемента если у нас конечный узел без ссылок

Del->n--;

return;

}

else if (Del->Leaves[i] != NULL)//Del->Leaves[i] Del->Leaves[i+1] Левый

{

Place = Del->Leaves[i];

while (Place->Leaves[0] != NULL)Place = Place->Leaves[Place->n];

if (Place->n > t - 1)

{

DelNum = Place->Numbers[Place->n - 1];

Delete(Place, DelNum);//вызываем поиск от этого уровня

Del->Numbers[i] = DelNum;//меняем число в верхнем корне

return;

}

Place = Del->Leaves[i + 1];

while (Place->Leaves[0] != NULL)Place = Place->Leaves[0];

if (Place->n > t - 1)

{

DelNum = Place->Numbers[0];

Delete(Place, DelNum);

Del->Numbers[i] = DelNum;

return;

}

}

{

BiTree\* Up = Del->Up;

if (Up != NULL)

{

int j;

for (j = 0; j < Up->n + 1; j++)

{

if (Up->Leaves[j] == Del)// j текущий

break;

}//перекидывание ключей

if (Del->Leaves[0] == NULL) {

if (j >= 1)//Рассматриваем правый узел

{

if (Up->Leaves[j - 1]->n > t - 1)

{

DelNum = Up->Leaves[j - 1]->Numbers[Up->Leaves[j - 1]->n - 1];//Берем максимум у левого соседа

Delete(Main, DelNum);

Del->Numbers[i] = Up->Numbers[j - 1];//Берем ключ с данным значением

Up->Numbers[j - 1] = DelNum;//так как это правый узел, то его число в Up находится по его значению -1

return;

}

}

if (j < Up->n)

{

if (Up->Leaves[j + 1]->n > t - 1)//Рассматриваем левый узел

{

DelNum = Up->Leaves[j + 1]->Numbers[0];//Берем самый малый ключ у правого соседа

Delete(Main, DelNum);

Del->Numbers[i] = Up->Numbers[j];//так как это левый узел, то его число в Up находится по его значению j

Up->Numbers[j] = DelNum;

return;

}

}

}

}

if (true)//Объединение листов i и i+1, удаление элемента из Del, пересохранение ссылок. крайняя мера

{

if (Del->Leaves[0] == NULL) Connect(Main, Del, NULL, num);//если узел конечным, просто объединяем с соседом

else//объединяем все листы до этого узла, удаляем элемент из этого узла

{

BiTree\* buff = Del->Leaves[i];

while (buff->Leaves[0] != NULL) buff = buff->Leaves[0];

Connect(Main, buff, Del);

if (Del->n < t - 1)

Connect(Main, Del, NULL, num);

else

{

Remove(Del->Numbers, num, Del->n);

RemoveLinks(Del->Leaves, NULL, Del->n);

Del->n--;

}

}

}

}

}

void Show(BiTree\* Main, std::ostream& streamOut)//Вывод дерева в отсортированной форме

{

if (Main != NULL)

{

for (int i = 0; i < Main->n + 1; i++)

{

if (Main->Leaves[i] != NULL)

{

Show(Main->Leaves[i], streamOut);

}

if (i < Main->n)

streamOut << Main->Numbers[i] << " ";

}

}

}

void Read(BiTree\*& Main, std::ifstream& streamIn)//Ввод дерева из файла, медленно

{

while (!streamIn.eof())

{

T a;

streamIn >> a;

Main->Add(Main, a);

}

}

};

int main()

{

BiTree<int>\* Main = NULL;

double Disk[101]{ 0 };

for (int i = 1; i < 101; i++)

{

Disk[i] = (Disk[i - 1] + i \* 0.02);

}

for (int i = 1; i < 101; i++)

{

Disk[i] = 100 \* Disk[i];

}

cout << endl;

for (int i = 0; i < 1000000; i++)

{

int Number = rand() % 10101;

for (int j = 0; j < 101; j++)

{

if (Number <= Disk[j])

{

Main->Add(Main, j, NULL, true);

break;

}

}

}

for (int i = 0; i < 1000000; i++)

{

Main->Add(Main, rand()% 1000000, NULL, true);

}

//ifstream file;

ofstream file2;

//file.open("BiTree.txt", ios::in);//Read ios::trunc

file2.open("BiTreeTree.txt", ios::trunc);

Main->Print(Main, file2);

cout << "Hello World!\n";

}

// Запуск программы: CTRL+F5 или меню "Отладка" > "Запуск без отладки"

// Отладка программы: F5 или меню "Отладка" > "Запустить отладку"

// Советы по началу работы

// 1. В окне обозревателя решений можно добавлять файлы и управлять ими.

// 2. В окне Team Explorer можно подключиться к системе управления версиями.

// 3. В окне "Выходные данные" можно просматривать выходные данные сборки и другие сообщения.

// 4. В окне "Список ошибок" можно просматривать ошибки.

// 5. Последовательно выберите пункты меню "Проект" > "Добавить новый элемент", чтобы создать файлы кода, или "Проект" > "Добавить существующий элемент", чтобы добавить в проект существующие файлы кода.

// 6. Чтобы снова открыть этот проект позже, выберите пункты меню "Файл" > "Открыть" > "Проект" и выберите SLN-файл.