Православный Свято-Тихоновский гуманитарный университет

Факультет информатики и прикладной математики

КАФЕДРА ИНФОРМАТИКИ

**Курсовой проект по курсу «Структуры и алгоритмы компьютерной обработки данных»**

**«Идеально-сбалансированные деревья поиска»**

Автор: Грицкова Ксения Анатольевна

Группа: ИПМ02-1-04

Проверил: профессор кафедры информатики, д.т.н. Соловьев А.В.

Москва 2019

# **Постановка задачи проекта**

Реализовать алгоритм построения идеально-сбалансированного дерева поиска (ИСДП) для обеспечения минимального среднего времени поиска числовых данных.

Входные данные — массив из неповторяющихся чисел. На выходе получаем идеально-сбалансированное дерево с возможностью добавления и удаления в нем элементов.

В этом дереве, как в дереве поиска, ключ в каждой его вершине должен быть больше ключа любой вершины в левом поддереве и меньше ключа любой вершине в правом поддереве. Чтобы дерево было идеально-сбалансированным необходимо, чтобы для каждой его вершины размеры левого и правого поддеревьев отличались **не более, чем на 1**. Высота ИСД минимальна и равна hисдп=hmin=[log2(n+1)].

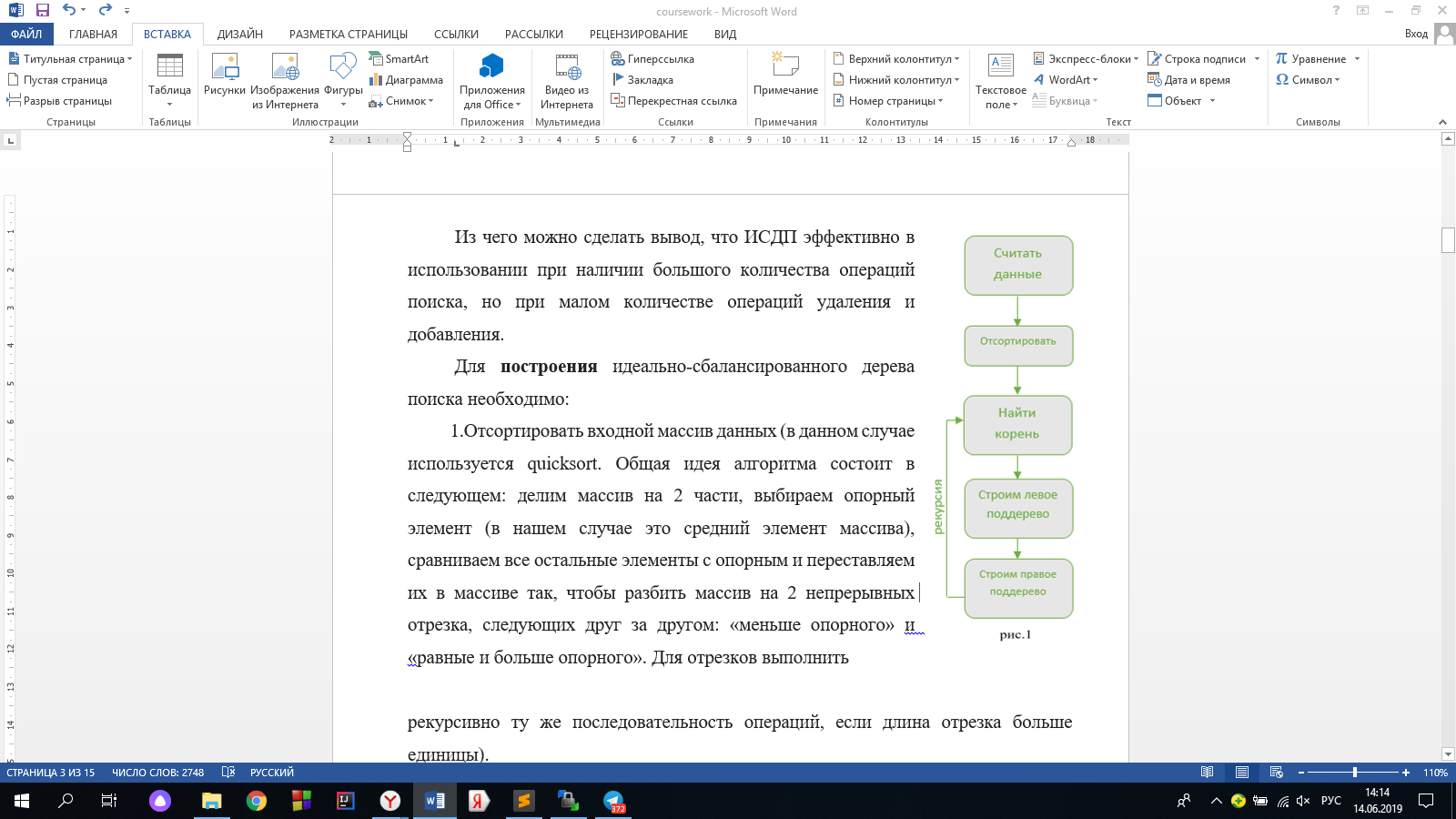
Данный алгоритм предназначен для формирования идеально-сбалансированного дерева из массива чисел, широко используется в учебном процессе для отработки навыков работы с деревьями поиска, включено в обучающие программы ФГОС.

# **Описание алгоритма**

Идеально-сбалансированное дерево — это двоичное дерево, размеры левого и правого поддеревьев которого отличаются не больше, чем на 1.

Следует заметить, что все поддеревья ИСДП также идеально-сбалансированы.

Основным достоинством такого дерева поиска является обеспечение минимального среднего времени поиска числовых данных, а недостатком — необходимость перестраивать дерево при каждой операции добавления и удаления.

Из чего можно сделать вывод, что ИСДП эффективно в использовании при наличии большого количества операций поиска, но при малом количестве операций удаления и добавления.

Для **построения** идеально-сбалансированного дерева поиска необходимо:

1.Отсортировать входной массив данных (в данном случае используется quicksort. Общая идея алгоритма состоит в следующем: делим массив на 2 части, выбираем опорный элемент (в нашем случае это средний элемент массива), сравниваем все остальные элементы с опорным и переставляем их в массиве так, чтобы разбить массив на 2 непрерывных

отрезка, следующих друг за другом: «меньше опорного» и ы«равные и больше опорного». Для отрезков выполнить

рекурсивно ту же последовательность операций, если длина отрезка больше единицы).

2.Средний элемент отсортированного массива берем в качестве корня. Из меньших элементов массива, которые составляют левую половину, строим левое поддерево, из больших — правое. Повторяем пункт 2 для каждого поддерева до тех пор, пока левое и правое поддеревья не станут пустыми.

Реализация осуществляется с помощью рекурсии для каждой половины массива.

Для ИСДП максимальное число сравнений равно Сmax = 2\*log2(n+1). Если считать, что поиск любой вершины происходит с одинаковой частотой, то ИСДП обеспечивает то самое минимальное среднее

время поиска, которое является его достоинством.

**Поиск** по дереву осуществляется следующим образом:

1.Сравниваем вершину дерева с элементом, который

хотим найти.

2.Если элементы совпадают, то выводим на экран сообщение о том, что элемент найден.

3.Если элементы не совпадают и искомый элемент

больше вершины, то ищем его уже в правом поддереве таким же образом. Если не нашли — переходим к поиску в левом поддереве.

**Добавление** элемента в ИСДП:

При добавлении отдельного элемента возникает проблема, которая является тем самым недостатком ИСДП. Для добавления нового элемента программе приходится заново строить дерево поиска.

1.Ключи построенного дерева записываем в массив

2.Проверяеем, есть ли в массиве элемент, который равен новому

3.Если такого элемента нет, он добавляется в массив (иначе на экран выводится сообщение о том, что в массиве есть введенный элемент)

4.Сортируем с помощью Quick Sort (сложность O(n·log n) )

5.Снова строим дерево

**Удаление** **элемента** из ИСДП:

Чтобы удалить элемент, программа сначала перекладывает дерево обратно в массив, затем удаляет введенный элемент, и только потом снова формирует идеально - сбалансированное дерево уже без удаленного элемента.

**Удаление всего дерева**:

1.Рекурсивно удаляем левую часть, затем правую. Удаление осуществляется при помощи поиска среднего элемента в каждой части.

2.Разбиваем каждую часть еще на две: до среднего элемента и после.

3.В каждой части снова ищем средний элемент, и снова происходит разбиение. Продолжается это до тех пор, пока не останется только один последний элемент, который удаляется.

После того, как левая и правая части дерева удалены, удаляется вершина дерева, а само оно инициализируется как NULL.

Работоспособность алгоритма проверялась на простых тестовых задачах, которые строились таким образом:

* Проверить насколько хорошо программа выполняет поставленную задачу при правильных условиях
* Проверить какие ошибки будет выдавать программа, если ввести данные, которые не соответствуют требуемым условиям.

Сложность алгоритма равна О(log n).

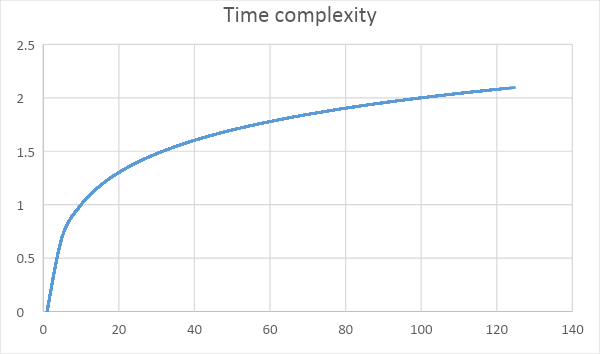


рис.2 – график временной сложности алгоритма

# **Руководство пользователя**

Для входа в программу нужно запустить выделенный файл **main.exe**, который находится в папке **PBST.**

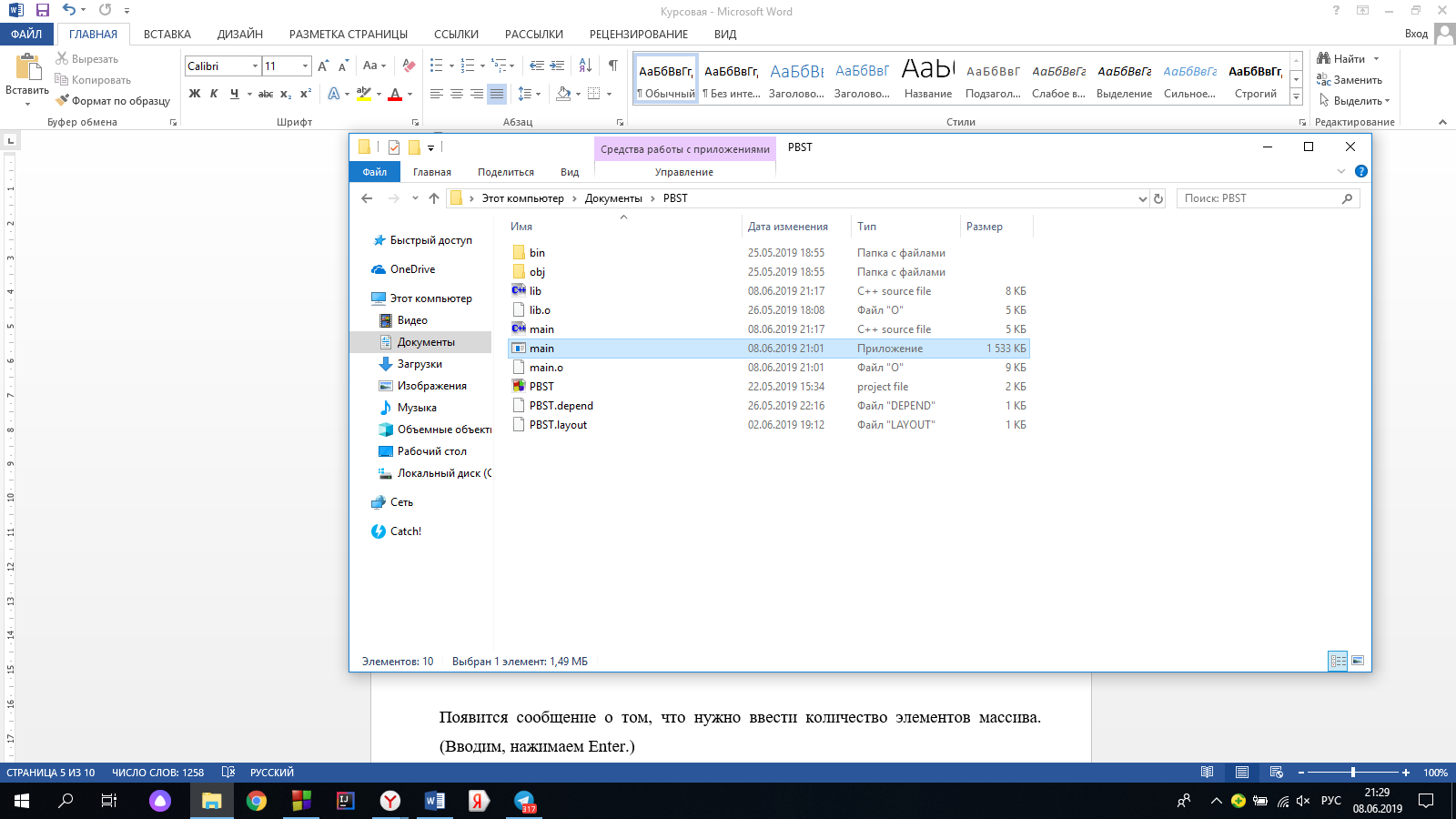


рис.3 – демонстрация запуска программы

Откроется меню:

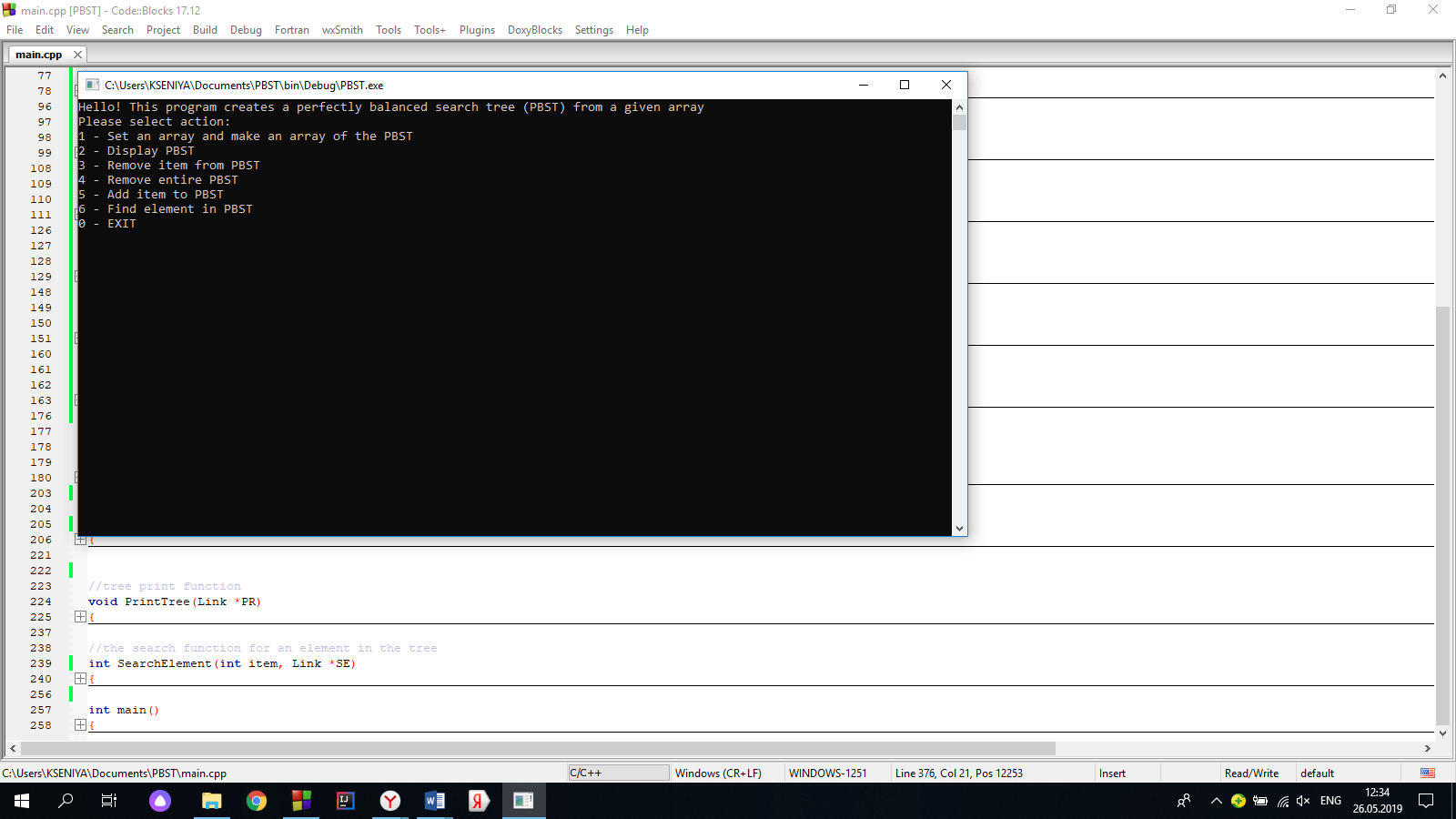


рис.4 – демонстрация входа в программу. Главное меню.

Чтобы задать массив введите цифру 1.

**После ввода каждой команды нужно нажимать Enter**!

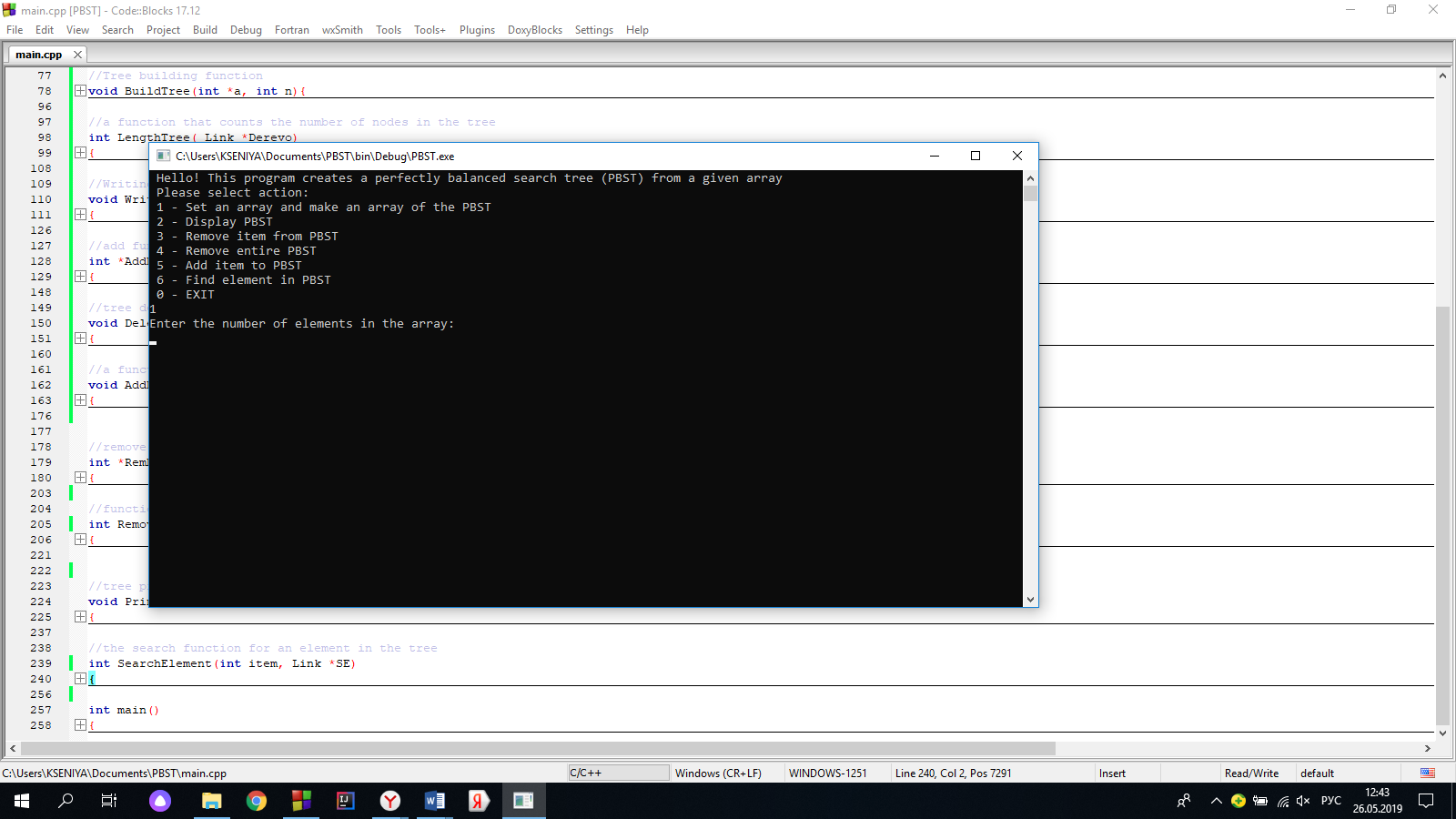


рис.5 – демонстрация вызова команды

Появится сообщение о том, что нужно ввести количество элементов массива. (Вводим, нажимаем Enter.)

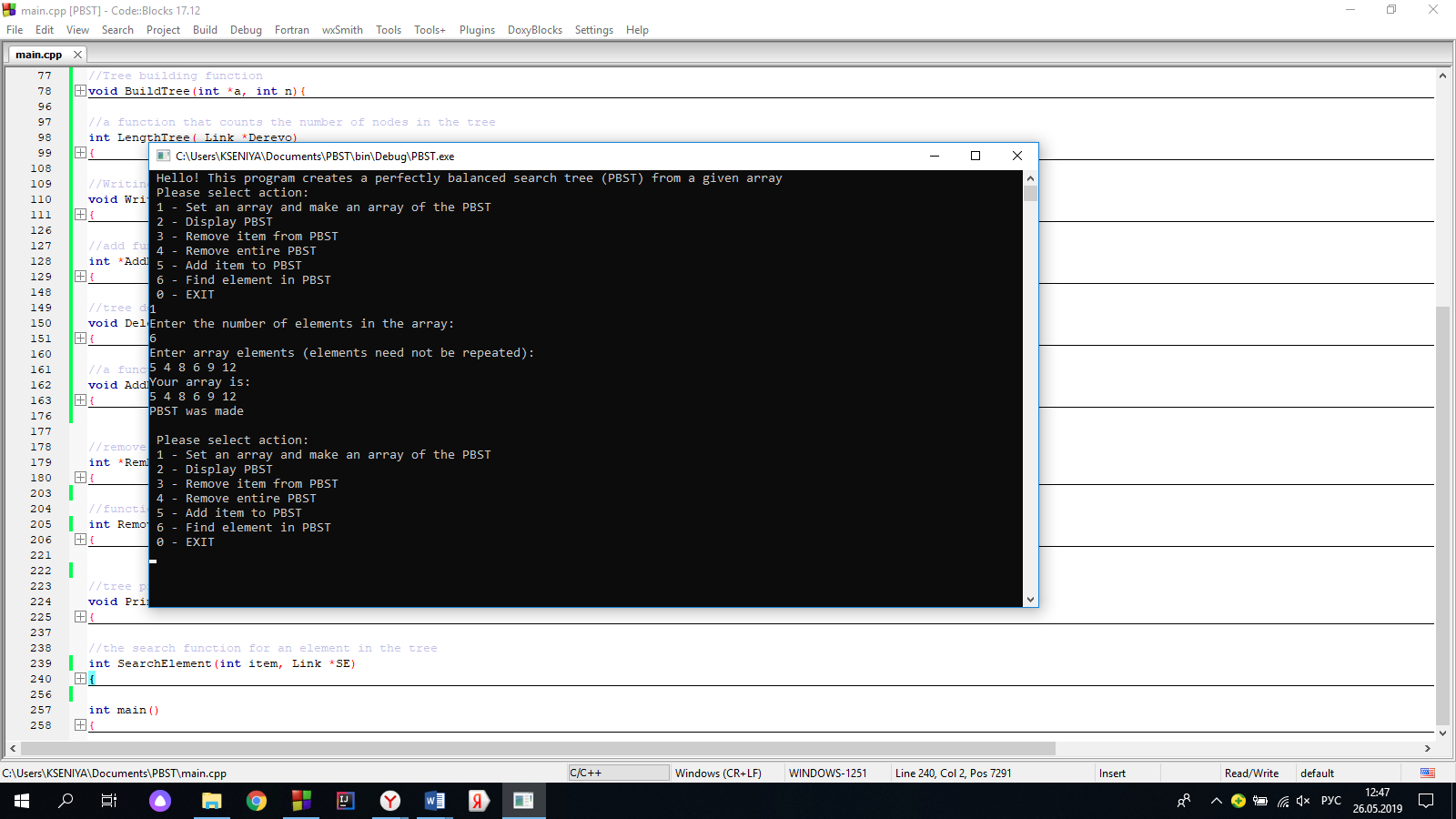


рис.6 – демонстрация создания ИСДП.

Теперь появится новое сообщение: нужно ввести элементы массива, но они не должны повторяться. Важно, что программа не предусматривает проверку на уникальность каждого элемента массива, поэтому важно следить за отсутствием повторений чисел.

Элементы массива отделяются друг от друга клавишами **Space bar**(пробел) или **Enter**. То есть вводим первый элемент, например, 5. Нажимаем одну из двух клавиш (Лучше Enter). Делаем это до тех пор, пока массив не будет заполнен. (Так как вы определили размер массива, то программа не даст ввести вам бóльшее количество элементов, если после каждого элемента нажимать Enter).

После того, как элементы заполнят массив, на экране появится сначала введенный массив, а потом сообщение о том, что наше Идеально-сбалансированное дерево создано. (Также вы увидите меню. Оно будет появляться после каждой завершенной команды).

Попробуем вывести дерево на экран. (Команда под номером 2)

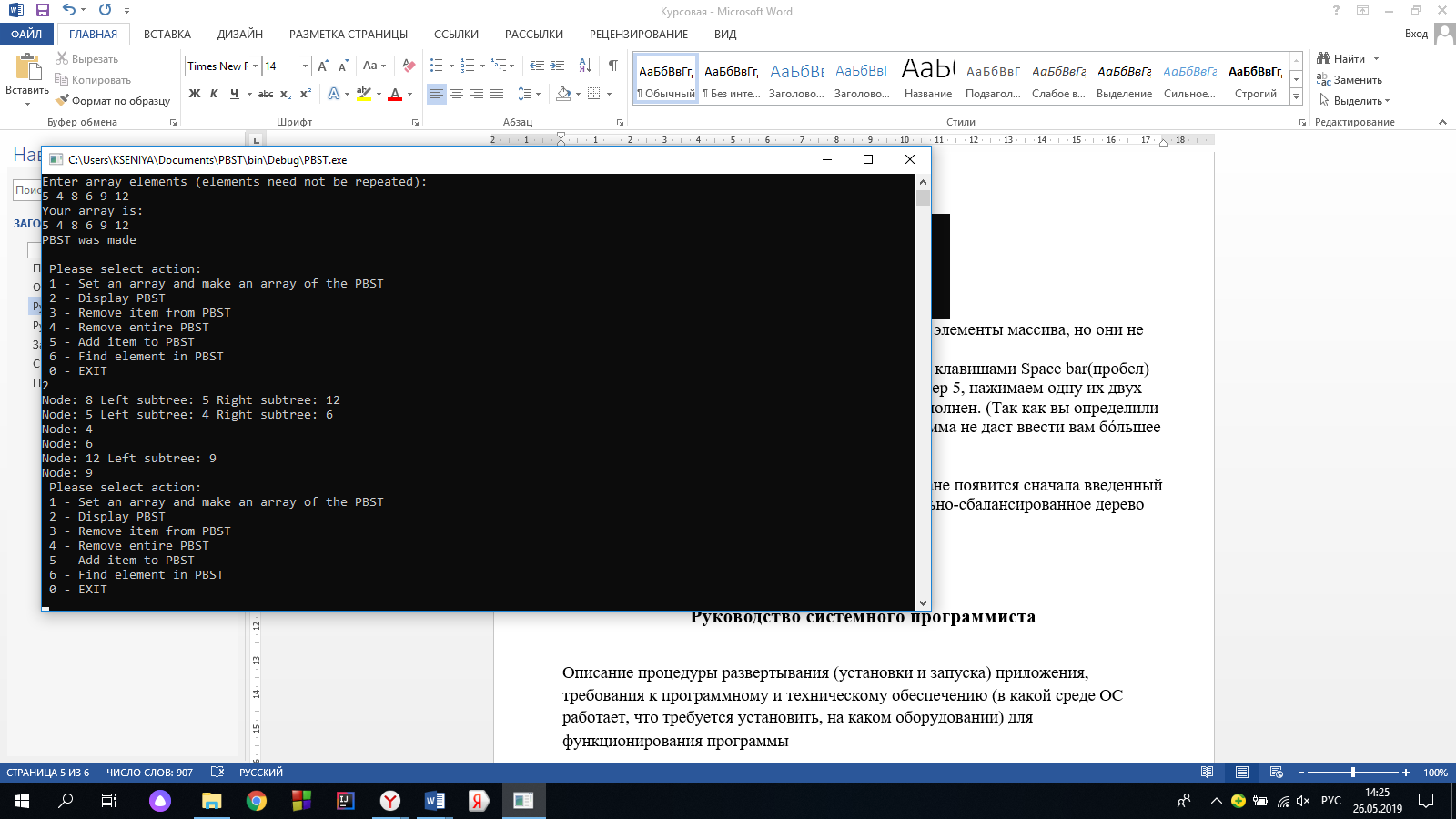


рис.7 – демонстрация вывода ИСДП на экран

Давайте разберемся как понять то, что вывела программа. В графическом виде дерево будет выглядеть так:

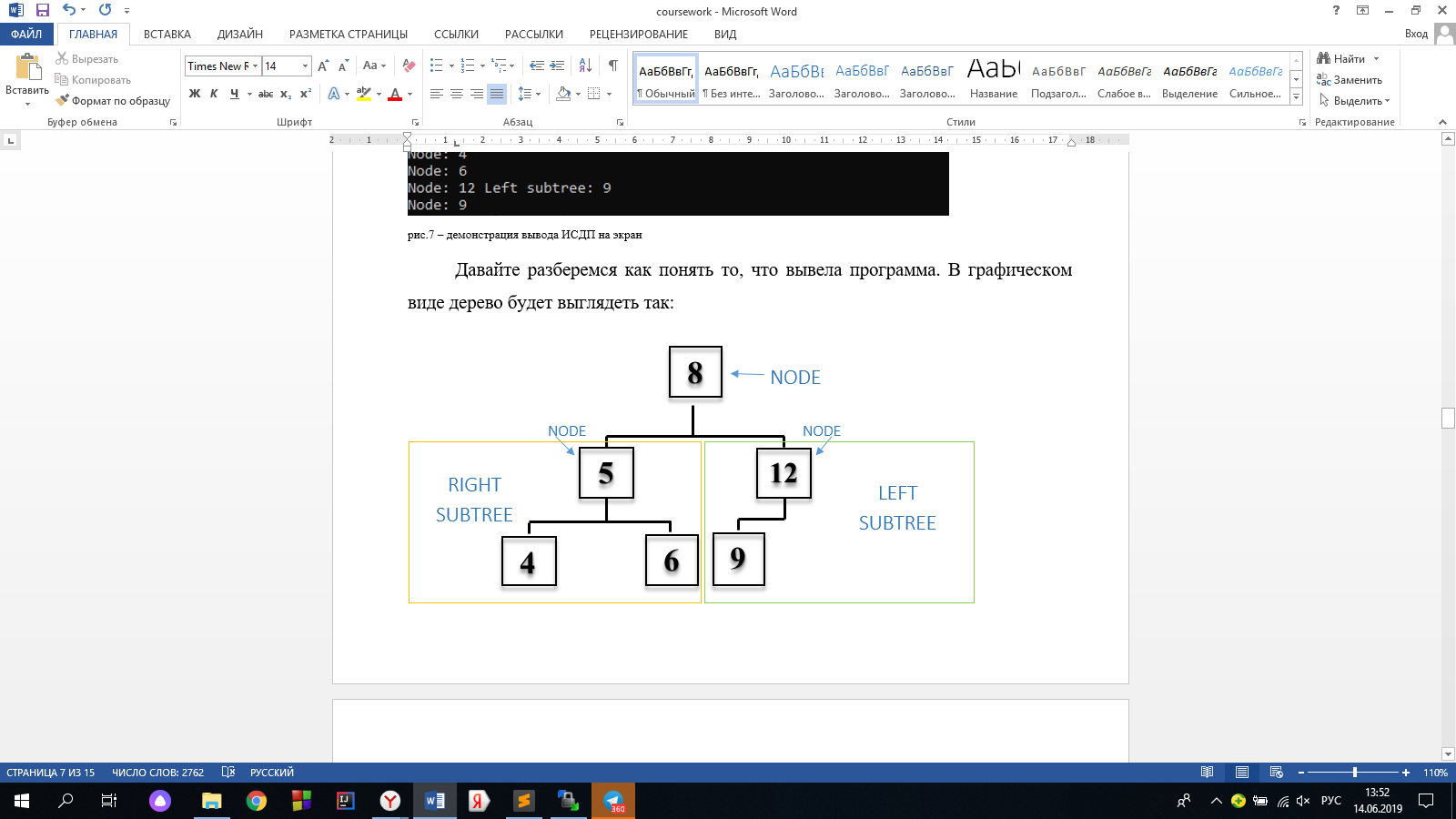


рис.8 – графическое изображение ИСДП.

Node — это узел, и первым узлом и корнем дерева в данном примере является число 8, у него есть 2 «ребенка» — левый (5) и правый (12), дальше рассматриваем левое поддерево: узел равен 5, и есть левый (4) «ребенок» и правый (6), теперь посмотрим на правое поддерево: узел равен 12, и есть только левый «ребенок» и его значение 9.

Давайте проверим команду **удаления** элемента, команда под номером 3.

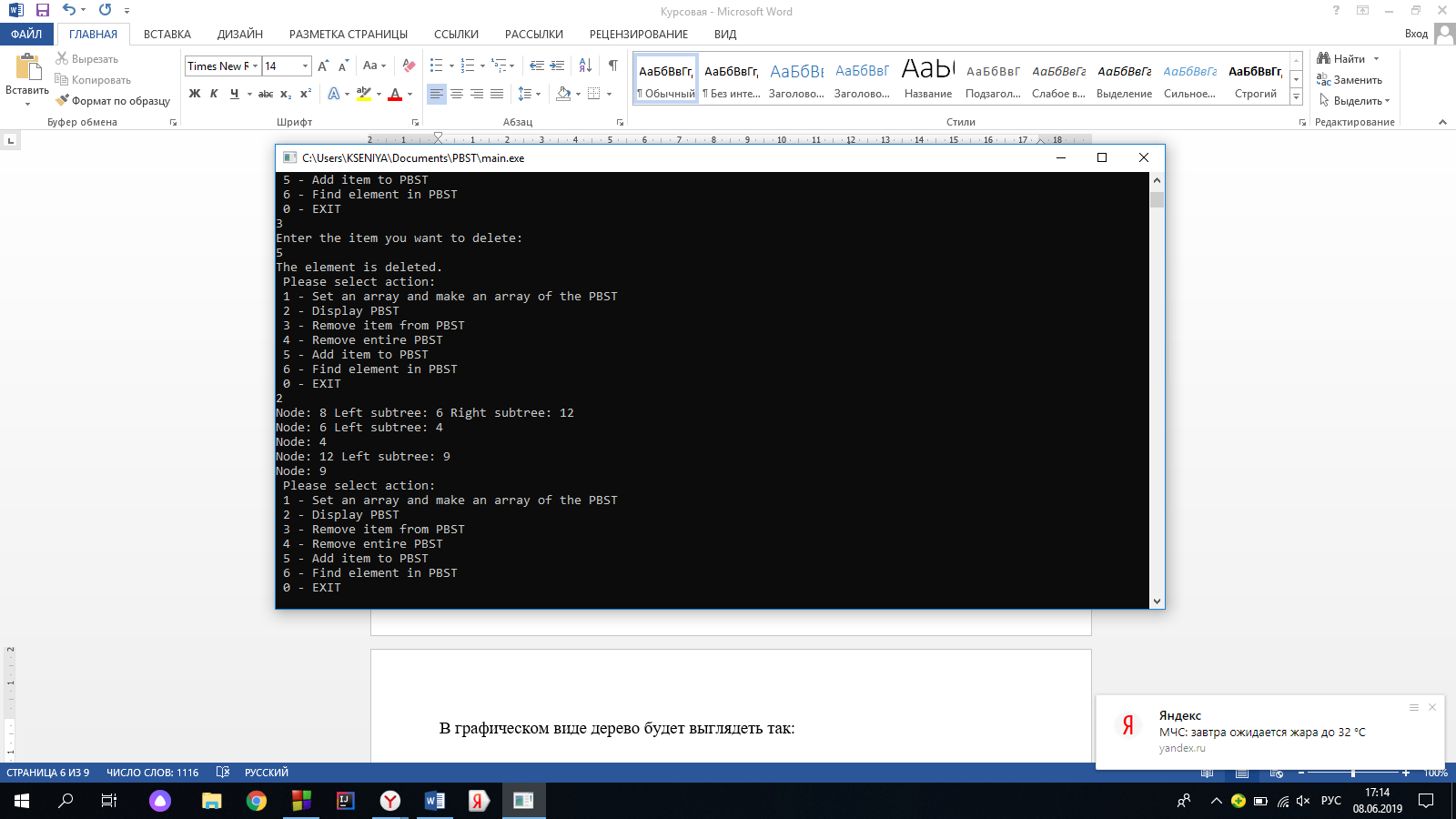


рис.9 – демонстрация удаления элемента.

Удалим элемент 5 и выведем дерево на экран, чтобы убедиться, что элемент действительно был удален.

Теперь попробуем добавить новый элемент в дерево. Для этого вызываем команду под номером 5.

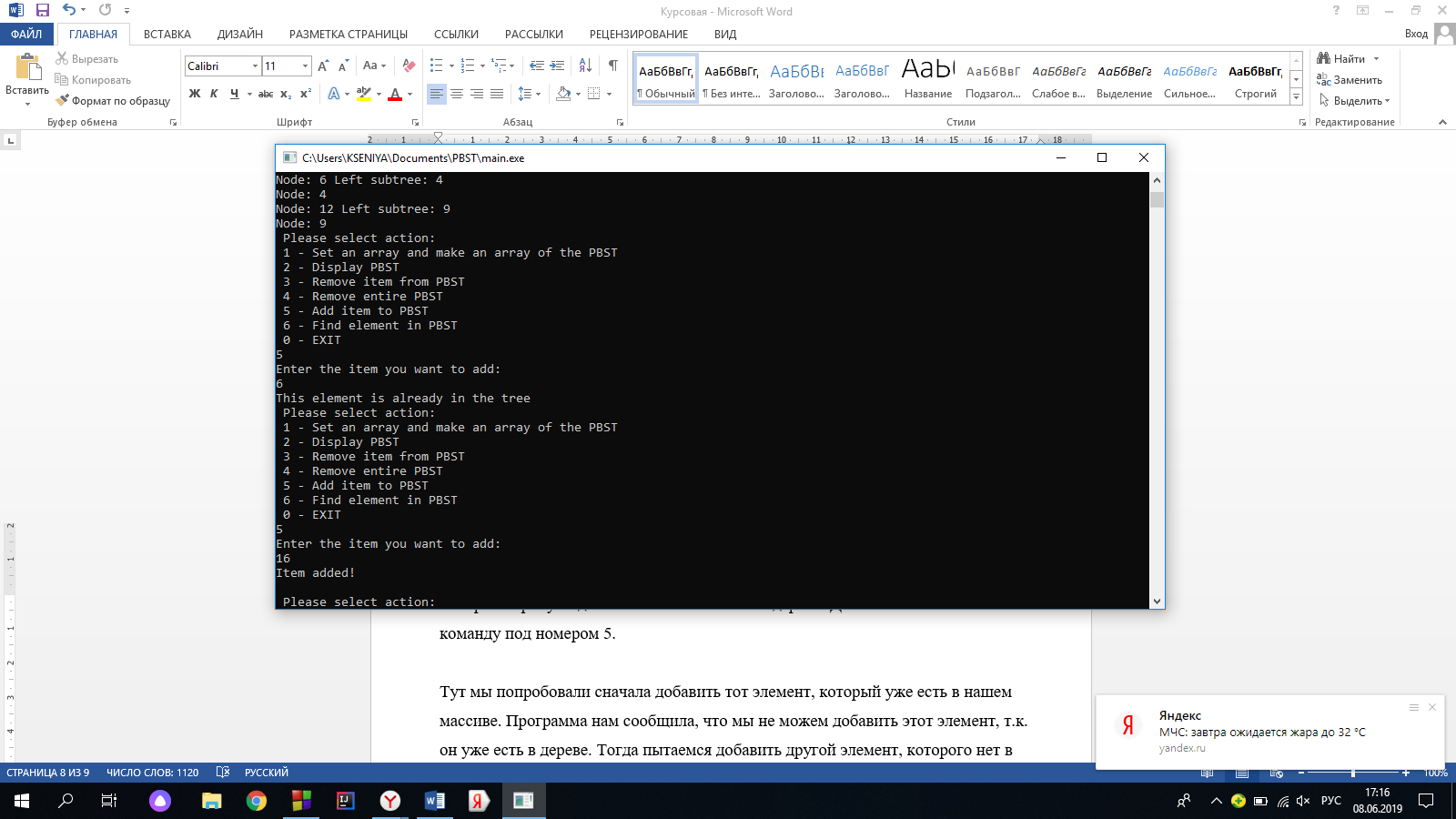


рис.10 – демонстрация добавления элемента в дерево.

Для начала введем тот элемент, который уже есть в дереве. Программа сообщает, что такой уже есть и не добавляет его в дерева, тогда мы вызываем еще раз эту же команду и введем элемент, которого еще нет в дереве. Нам сообщают, что элемент добавлен.

С помощью команды 6 можно убедиться есть ли элемент в дереве. Давайте проверим как это работает.

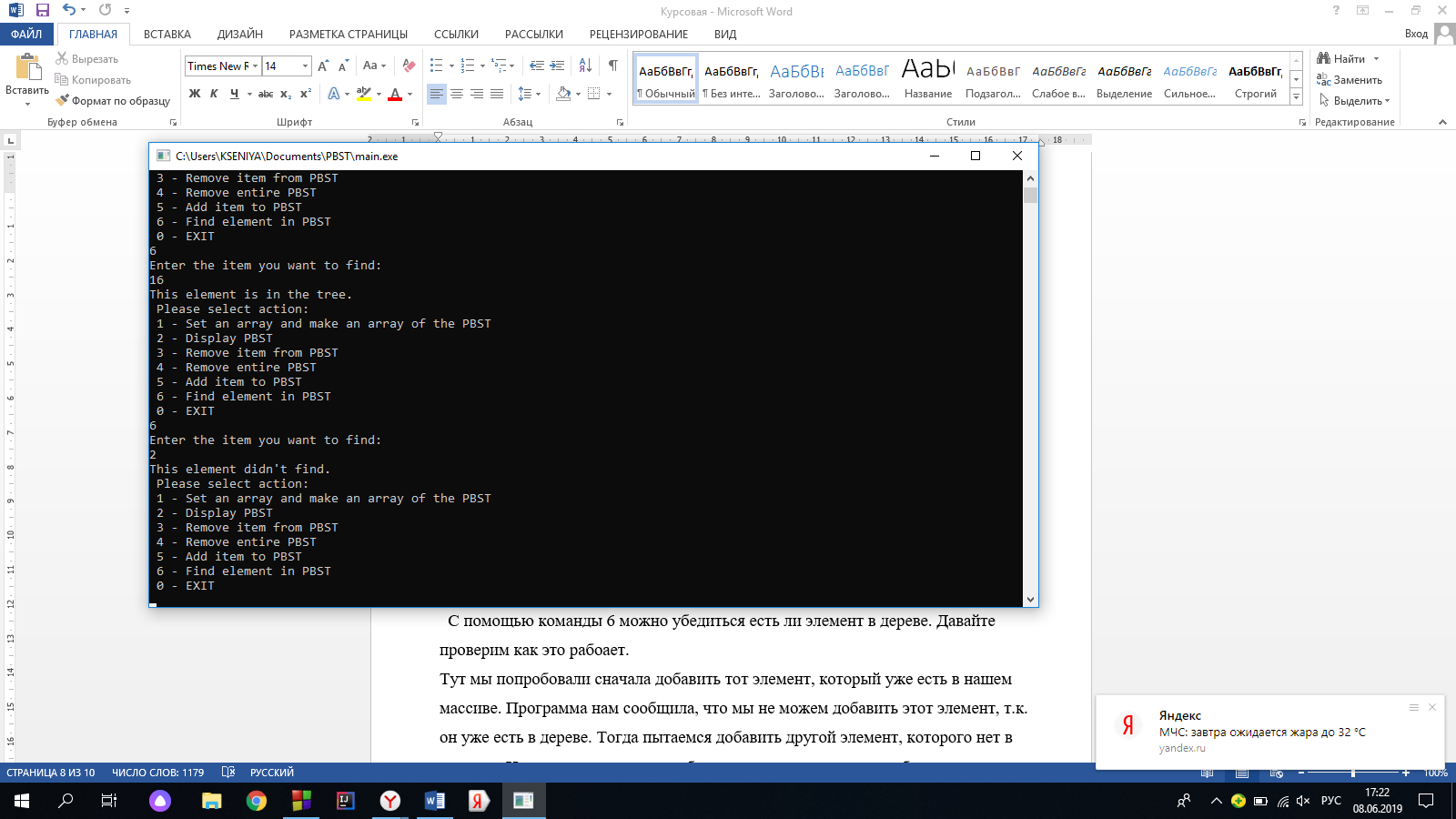


рис.11 – демонстрация поиска элемента

А что будет, если ввести элемент, которого нет в дереве?

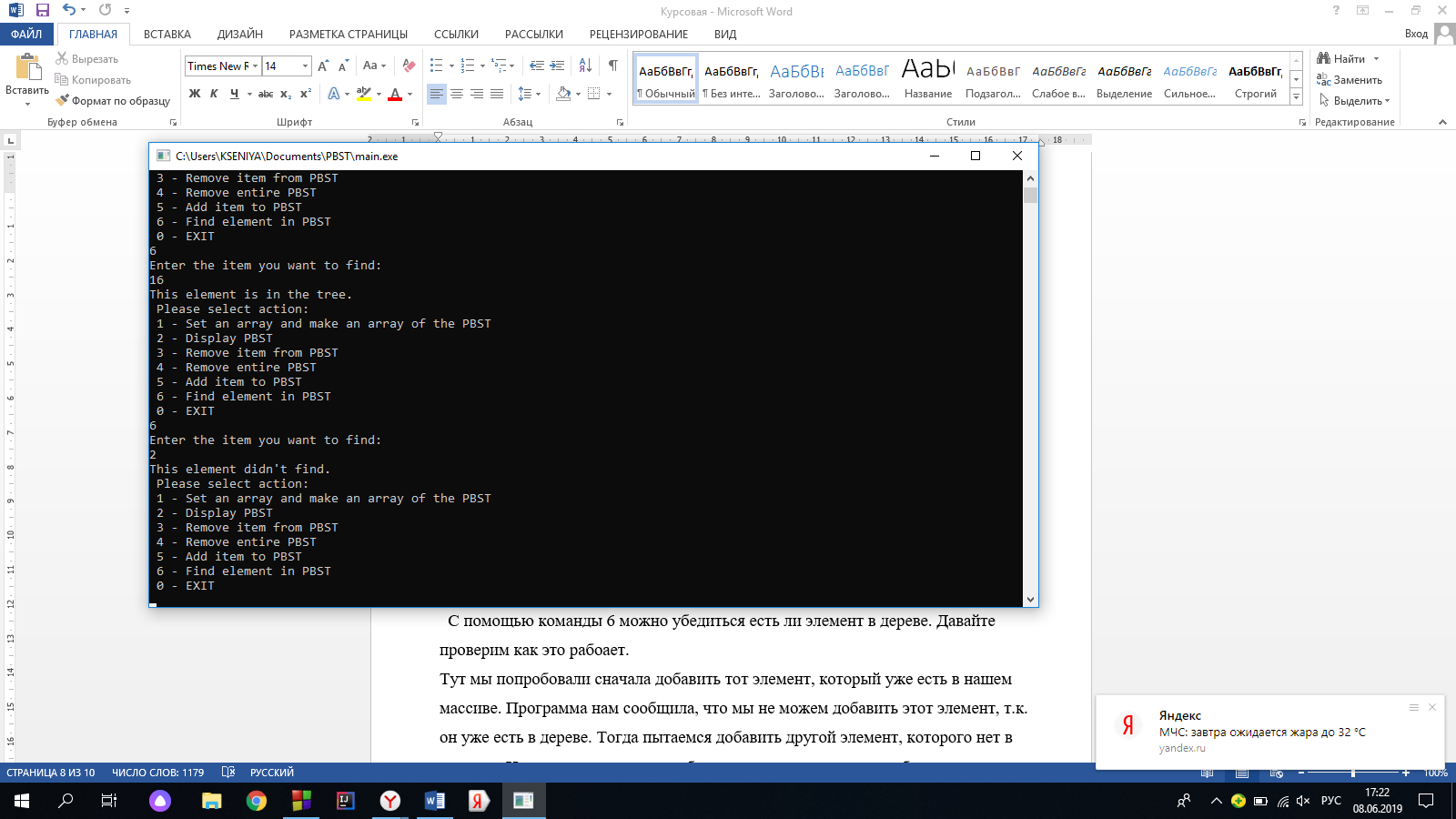


рис.12 – демонстрация поиска несуществующего в дереве элемента.

Хорошо, теперь надо попробовать удалить дерево. (команда 4)

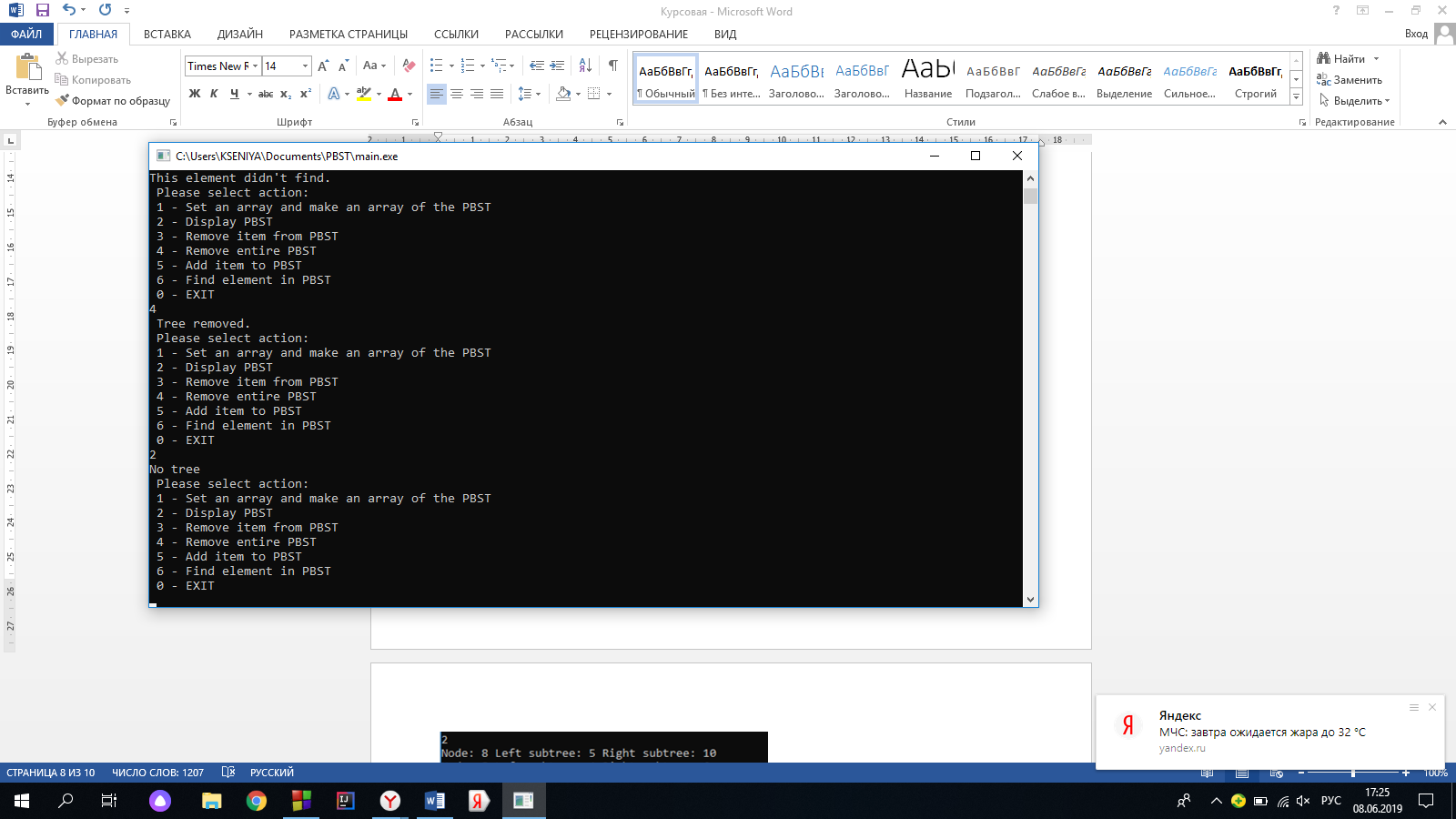


рис.13 – демонстрация удаления дерева и проверки.

С помощью команды 4 мы удалили дерево. Чтобы наверняка в этом убедиться, воспользуемся командой 2. Как видим, дерево удалено.

Для завершения работы нужно воспользоваться командой 0.

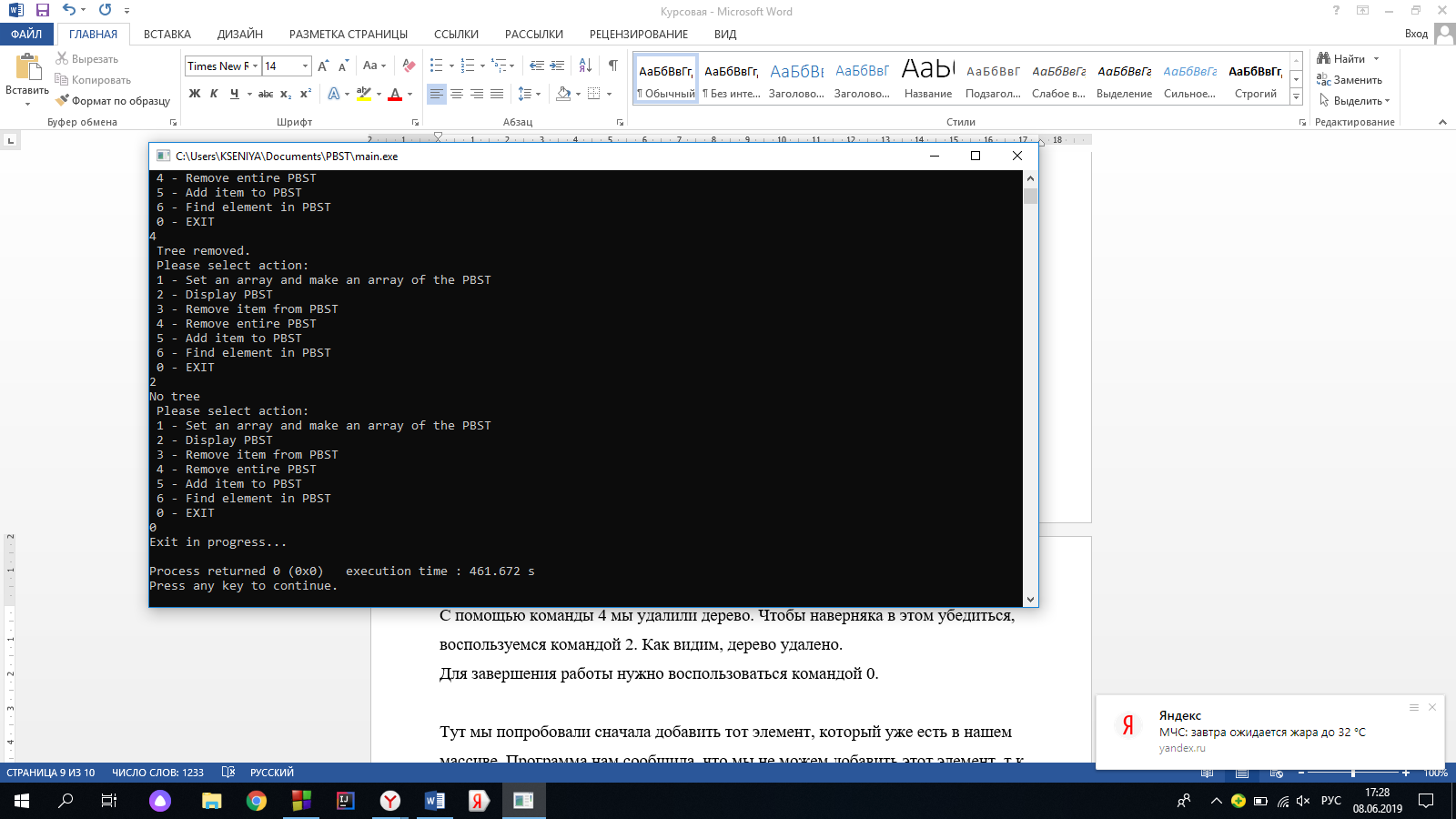


рис.14 – демонстрация выхода из программы

# **Руководство системного программиста**

Программа запускается на ОС Windows (начиная с версии XP).

# **Заключение**

В данной работе, выполнена программная реализация алгоритма построения идеально-сбалансированного дерева поиска. Программа добавляет, удаляет и находит элементы в дереве, а также может удалить его целиком. Как было описано ранее, из-за своего довольно большого недостатка, который уменьшает эффективность алгоритма в разы, данная программа используется только в учебных целях для отработки навыков работы с деревьями поиска.

Работоспособность алгоритма была проверена на наборе тестовых данных, оценочная временная сложность составляет О(log n).

# **Список литературы**

1. «Структуры и алгоритмы компьютерной обработки данных» А.В. Соловьев. - М.: Изд-во ПСТГУ, 2013. - 128 с. (учебно-методическое пособие).

# **Приложение**

Класс lib.cpp:

#include <iostream>

#include <ctime>

#include <cstdlib>

using namespace std;

struct Link{

int x; //the number we will record

Link \*l, \*r; //address fields(left and right)

};

Link \*Tree=NULL; //pointer whose type is "tree link"

void Sort(int \*a, int n) { //The sort function of the array

int left=0;

int right=n-1;

int middle=a[n/2];

while(left<=right){

while(a[left]<middle) {

left++; //move the pointer to the right until it approaches the middle element

}while(a[right]>middle){

right--;

}if(left<=right){

swap(a[left], a[right]); //change the items in places

left++;

right--;

}

}if (right>0) Sort(a, right+1);

if (left < n-1) Sort(a+left, n-left);

}

void Add(Link \*Node, Link \*Branch){ //a function that adds value to the tree

//work with the left part of the tree

if(Node->x<=Branch->x){ //condition: the entered element is smaller than the node

if(Branch->l) //condition: there is a left tree

Add(Node,Branch->l); //recursively write the element to the left in the free space

else

Branch->l=Node; //otherwise, the left subtree becomes a node

}else{ //else a free site is not

if(Branch->r) //condition: there is a right tree

Add(Node,Branch->r); //recursively write the element to the right in the free space

else Branch->r=Node;//otherwise, the right subtree becomes a node

}

}

void NewLink(int x) { //parent and 2 child creation function

Link \*Node = new Link;

Node->x = x; //write data to our node

Node->l = NULL; //create the left link and initialize to avoid errors

Node->r = NULL; //create the left link and initialize

if(Tree){

Add(Node, Tree); //call the add function from Node if there is a tree

}else Tree = Node; //put in the base of out node if there is not a tree

}

void BuildTree(int \*a, int n){ //Tree building function

int middle=0; //this is a variable for the middle element

if(n==1){

NewLink(a[0]); //create node

}else if(n==2){

middle = n/2; //calculate middle element

NewLink(a[n/2]); //make it a node

BuildTree(a,1); //recursively arrange other elements

}else{

middle = n/2;

NewLink(a[n/2]);

BuildTree(a, n/2); //recursively work with the left part or the array; build the left subtree

BuildTree(a+n/2+1, n-n/2-1); //recursively work with the right part or the array; build the right subtree

}

}

int LengthTree(Link \*Derevo) { //a function that counts the number of nodes in the tree

if(Derevo==NULL) { //if there is no tree

return (0);

}else{

return(LengthTree(Derevo->l)+1+LengthTree(Derevo->r)); //count and return the number of nodes

}

}

int \*AddElToArray (int \*olditem, int number, int newel){ //add function to add an element to an array

int numberTemp=number+1; // create a variable for more elements;

int \*temp = new int[numberTemp]; //create an array with a large number of elements;

if(number==1){

temp[0]=olditem[0]; // shift the array element

temp[number]=newel; //add new

}else{

for(int i=0; i<number; i++){

temp[i] = olditem[i]; //shift the array elements

}

temp[number] = newel; // in the last place put element that you want to add

return temp; //return new array

}

return temp;

}

void WritingTreeToAnArray(int \*arr, int b, Link \*MR) { //Writing a tree to an array

if(b==1){

arr[0]=MR->x; //writing a single element to the array

}else if(b==2){

arr[b/2]=MR->x; //write the node as the middle element of the array

WritingTreeToAnArray(arr, 1, MR->l); //recursively insert the last element into the array

}else{

arr[b/2]=MR->x; //write the node as the middle element

WritingTreeToAnArray(arr, b/2, MR->l); // recursively write the left subtree to the left part of the array

WritingTreeToAnArray(arr+b/2+1, b-b/2-1, MR->r); //recursively write the right subtree to the right part of the array

}

}

void DeleteTree(Link \*Emptily){ //tree delete function

if(Emptily){

if(Emptily->l)DeleteTree(Emptily->l); //recursively remove left part

if(Emptily->r)DeleteTree(Emptily->r); //recursively remove right part

delete Emptily; //remove last item

Emptily==NULL;

}

}

void AddElement(int element, Link \*T){//a function that adds value to the ready tree

int m=0;//create a variable

m=LengthTree(T);//assign the number of nodes in the tree to the element

int \*ARR=new int[m];//create new array with the number of elements equal number of node in the tree

WritingTreeToAnArray(ARR, m, T); //write the tree to the new array

int g = m + 1; //create a variable with the number of elements 1 more

int \*p = new int[g]; //create new array with the number of elements 1 more

p = AddElToArray(ARR, m, element); //add a new item to array

Sort(p, g); //sort array

DeleteTree(Tree); //delete old tree

Tree = NULL;

BuildTree(p, g);

}

int \*RemElFromArr( int \*old, int num, int E){//remove element from array

int p = 0;

int b = 0;

int numTemp = num-1; //create a variable with a smaller of items 1

int \*Temp = new int[numTemp]; //create new array with a smaller of items 1

for(int i = 0; i < num; i++){

if(old[i]!= E || p==1) //conditions: elements of the array is not equal to that which is necessary to remove or token =1

{

Temp[b] = old[i]; // shift the array elements

b++;

}else{

p = 1;

}

}if (p == 0) return NULL;

return Temp;

}

int RemoveElement(int element, Link \*RM){//function of removing an element from the tree

int k = LengthTree(RM); //put in a new variable the number of nodes in the tree

int N[k]; //create new array with this number of elements

WritingTreeToAnArray(N, k, RM); //shift tree to a new array

int s = k -1; //create a variable with a smaller of items 1

int \*L = new int[s]; //create array with this number elements

if(!L) return 0;

L = RemElFromArr(N, k, element); //remove an element from the array

Sort(L, s);

DeleteTree(Tree); //remove old tree

Tree = NULL;

BuildTree(L, s);

}

void PrintTree(Link \*PR) { //tree print function

if(PR){

cout << "Node: " << PR->x; //print Node

if(PR->l) cout << " Left subtree: " << PR->l->x;

if(PR->r) cout << " Right subtree: " << PR->r->x;

cout << endl;

PrintTree(PR->l); //recursively print left subtree

PrintTree(PR->r); //recursively print right subtree

}

}

int SearchElement(int item, Link \*Tr){//the search function for an element in the tree

if(item<Tr->x) {//condition: the sought element is smaller than the vertex

if(Tr->l)//condition: there is a left subtree

SearchElement(item, Tr->l);//recursively looking for item the left subtree

else return 0;//else nothing return

}else if(item>Tr->x) { //condition: the sought element is bigger than the vertex

if(Tr->r)//condition: there is a right subtree

SearchElement(item,Tr->r);//recursively looking for item the right subtree

else return 0;

}else { //item equal the vertex

return item;

}

}

Класс main.cpp:

#include <iostream>

#include <ctime>

#include <cstdlib>

#include "lib.cpp";

using namespace std;

int main(){

cout << " Hello! This program creates a perfectly balanced search tree (PBST) from a given array " << endl;

int choice; //variable that indicates the selected action

int n; //number of array elements

int addel; // the item that the user wants to add

int rm; // the item that the user wants to remove

int findel; // the item that the user wants to find

while(!0) //menu

{

cout << " Please select action: " << endl;

cout << " 1 - Set an array and make an array of the PBST" << endl;

cout << " 2 - Display PBST" << endl;

cout << " 3 - Remove item from PBST" << endl;

cout << " 4 - Remove entire PBST" << endl;

cout << " 5 - Add item to PBST" << endl;

cout << " 6 - Find element in PBST" << endl;

cout << " 0 - EXIT" << endl;

cin >> choice;

switch(choice)

{

case 0://Exit

{

cout << "Exit in progress..." << endl;

return 0;

break;

}

case 1: //Make tree

{

cout << "Enter the number of elements in the array: " << endl;

cin >> n;

int \*a = new int[n];

cout << "Enter array elements (elements need not be repeated): " << endl;

for(int i = 0; i < n; i++) { //set an array

cin >> a[i];

}

cout << "Your array is: " << endl;

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

cout << a[i] << " ";

}

cout << endl;

Sort(a,n); //sort array

BuildTree(a,n);

cout << "PBST was made" << endl;

cout << endl;

break;

}

case 2: //display tree

{

if(Tree){

PrintTree(Tree);//if the tree is print it

}else{

cout << "No tree" << endl;

}

break;

}

case 3: //remove item

{

cout << "Enter the item you want to delete: " << endl;

cin >> rm;

int v = SearchElement(rm, Tree); //check the element for presence in the tree

if(v) {

cout << "The element is deleted." << endl;

RemoveElement(rm, Tree); //remove element

}else{

cout << "Ops...Item not found." << endl;

}

break;

}

case 4: //remove tree

{

DeleteTree(Tree); //remove tree

Tree = NULL; //initialize the tree with zero

cout << " Tree removed." << endl;

break;

}

case 5: //Add element

{

cout << "Enter the item you want to add:" << endl;

cin >> addel;

int k;

k = SearchElement(addel, Tree); //check the element for uniqueness

if(k){

cout << "This element is already in the tree" << endl;

}else{

AddElement(addel, Tree); //add this item

cout << "Item added! " << endl;

cout << endl;

}

break;

}

case 6: { //find element in the tree

cout << "Enter the item you want to find: " << endl;

cin >> findel;

int w;

w = SearchElement(findel, Tree); //find item

if(w){

cout << "This element is in the tree. " << endl;

}else{

cout << "This element didn't find. " << endl;

}}}}}