Белорусский государственный технологический университет

Факультет информационных технологий

Кафедра программной инженерии

Лабораторная работа 12

По дисциплине «Основы алгоритмизации и программирования»

На тему «Разработка проекта с использованием бинарного дерева»

Выполнила:

Студентка 1 курса 2 группы

Глухова Д.В.

Преподаватель: асс. Андронова М.В.

2023, Минск

Вариант 2

4. Добавить к проекту  функции смешанного и нисходящего обхода дерева с выводом  на консоль, проверки сбалансированности дерева и функцию в соответствии с вариантом из таблицы, представленной в лабораторной работе № 11, изменив ее так, чтобы функция соответствовала проекту данной лабораторной работы.

Задание из лабораторной работы №11

Дан указатель **p1** на корень бинарного дерева. Написать функцию вывода количества листьев дерева, которые являются правыми дочерними вершинами.

|  |
| --- |
| Главный файл |
| #include "Header.h"  #include <fstream>  void main()  {  int countChet = 0;  setlocale(LC\_ALL, "Russian");  Tree\* mainRoot = new Tree;  int choice;  // 13 8 17 4 11 15 25 2 6 22 27 20 33  while (1)  {  int numLeft = 1;  cout << "\n";  cout << "1 - Добавить элемент;\n";  cout << "2 - Вывести дерево;\n";  cout << "3 - Удалить элемент;\n";  cout << "4 - Прямой обход с выводом;\n";  cout << "5 - Обратный обход с выводом;\n";  cout << "6 - Cмешанный обход с выводом;\n";  cout << "7 - Балансировка дерева;\n";  cout << "8 - Задание из 11 лаб 2 вар;\n";  cout << "0 - Выход\n";  cout << "Выберите:\n";  cin >> choice;  cout << "\n";  switch (choice)  {  case 1:  {  cout << "\nДля остановки введите отрицательное число." << endl;  cout << "Введите элемент: "; cin >> choice;  while (choice > 0)  {  mainRoot->AddNode(choice);  cout << "Введите элемент: "; cin >> choice;  if (choice % 2 == 0)  {  countChet++;  }  }  break;  }  case 2:  {  if (!mainRoot->node) cout << "В дереве ничего нет.\n";  else mainRoot->PrintTree(mainRoot->node, 0);  break;  }  case 3:  {  int key;  cout << "Введите ключ элемента, который хотите удалить: "; cin >> key;  mainRoot->Delete(key);  break;  }  case 4:  {  if (!mainRoot->node) cout << "В дереве ничего нет.\n";  else mainRoot->DirectBypass(mainRoot->node);  break;  }  case 5:  {  if (!mainRoot->node) cout << "В дереве ничего нет.\n";  else mainRoot->DownScan(mainRoot->node);  break;  }  case 6:  {  if (!mainRoot->node) cout << "В дереве ничего нет.\n";  else mainRoot->ReverseBypass(mainRoot->node);  break;  }  case 7:  {  mainRoot->counter = 0;  if (!mainRoot->node) cout << "В дереве ничего нет.\n";  else  if (mainRoot->Balance(mainRoot->node))  cout << "Дерево сбалансировано." << endl;  else  cout << "Дерево не сбалансировано." << endl;  break;  }  case 8:  {  numLeft = mainRoot->check\_right(mainRoot->node);  cout << "Количество правых дочерних вершин = " << numLeft << endl;  break;  }  case 0:  {  exit(0);  break;  }  default:  {  cout << "Такого пункта нет. Выберите снова.\n";  break;  }  }  }  } |
| Файл с функциями |
| #include "Header.h"  int sum = 0;  void Tree::AddNode(int k) // фунцкия для добавления узла  {  if (this->node == NULL) // если узла нет, то есть он равен NULL  {  this->node = new Node; // выделяем память  this->node->count = 1;  this->node->key = k; // присваиваем значение  this->node->left = this->node->right = NULL; // очищаем память для след роста  return;  }  Node\* temp = this->node;  while (true)  {  if (k == temp->key) // если k = одному из ключей  {  temp->count++;  return;  }  if (k < temp->key) // для левого поддерева  {  if (temp->left == NULL)  {  temp->left = new Node;  temp->left->count = 1;  temp->left->key = k;  temp->left->left = temp->left->right = NULL;  return;  }  else  {  temp = temp->left;  continue;  }  }  if (k > temp->key) // для правого поддерева  {  if (temp->right == NULL)  {  temp->right = new Node;  temp->right->count = 1;  temp->right->key = k;  temp->right->left = temp->right->right = NULL;  return;  }  else  {  temp = temp->right;  continue;  }  }  }  }  void pushFromFile(Node\*\* t, int a) {  if ((\*t) == NULL) { // если дерева не существует  (\*t) = new Node; // выделение памяти  (\*t)->key = a; // присваиваем  (\*t)->left = (\*t)->right = NULL; // очищаем память для след роста  return;  }  if (a > (\*t)->key) // если а > чем текущий элемент, то  pushFromFile(&(\*t)->right, a); // направо  else // иначе  pushFromFile(&(\*t)->left, a); // налево  }  void Tree::PrintTree(Node\* temp, int tab)  {  int i;  if (temp != NULL) // если дерево не пустое  {  PrintTree(temp->right, tab + 1); // рекурсия для правого поддерева  for (i = 1; i <= tab; i++)  cout << " ";  cout << temp->key << endl; // печатаем  PrintTree(temp->left, tab + 1); // рекурсия для левого поддерева  }  }  void Tree::Delete(int key)  {  Node\* Del, \* Prev\_Del, \* R, \* Prev\_R;  Del = this->node;  Prev\_Del = NULL;  while (Del != NULL && Del->key != key)  {  Prev\_Del = Del;  if (Del->key > key)  Del = Del->left;  else  Del = Del->right;  }  if (Del == NULL)  {  puts("\nНе найден");  }  if (Del->right == NULL)  R = Del->left;  else  if (Del->left == NULL)  R = Del->right;  else  {  Prev\_R = Del;  R = Del->left;  while (R->right != NULL)  {  Prev\_R = R;  R = R->right;  }  if (Prev\_R == Del)  R->right = Del->right;  else  {  R->right = Del->right;  Prev\_R->right = R->left;  R->left = Prev\_R;  }  }  if (Del == this->node) this->node = R;  else  if (Del->key < Prev\_Del->key)  Prev\_Del->left = R;  else  Prev\_Del->right = R;  int tmp = Del->key;  cout << "\nБыл удален элемент с ключом " << tmp << endl;  delete Del;  }  // Функция печатает значения бинарного дерева поиска в прямом порядке.  // Вместо печати первой инструкцией функции может быть любое действие  // с данным узлом  void Tree::DirectBypass(Node\* temp)  {  if (temp)  {  cout << temp->key << " ";  if (temp->left)  DirectBypass(temp->left); // рекурсивный вызов для левого  if (temp->right)  DirectBypass(temp->right); // рекурсивный вызов для правого  }  }  // Функция печатает значения бинарного дерева поиска в симметричном порядке.  // То есть в отсортированном порядке  void Tree::ReverseBypass(Node\* temp)  {  if (temp)  {  if (temp->left)  ReverseBypass(temp->left);  cout << temp->key << " ";  if (temp->right)  ReverseBypass(temp->right);  }  }  // Функция печатает значения бинарного дерева поиска в обратном порядке.  // Не путайте обратный и обратноотсортированный (обратный симметричный).  void Tree::DownScan(Node\* temp)  {  if (temp != NULL)  {  DownScan(temp->left);  DownScan(temp->right);  cout << temp->key << " ";  }  }  bool Tree::Balance(Node\* temp)  {  if (temp)  {  if (temp->right)  {  this->counter++;  Balance(temp->right);  }  if (temp->left)  {  this->counter--;  Balance(temp->left);  }  if (this->counter >= -1 && this->counter <= 1)  return true;  else  return false;  }  }  int Tree::check\_right(Node\* temp) // правые дочерние узлы  {  if (temp) {  if (temp->left)  {  check\_right(temp->left);  sum++;  }  if (temp->right)  check\_right(temp->right);  return sum;  }  } |
| Заголовочный файл Header.h |
| #pragma once  #include <iostream>  using namespace std;  struct Node  {  int key; // ключ  int count; // счетчик  Node\* left; // левый узел  Node\* right; // правый узел  };  struct Tree  {  Node\* node = nullptr;  int counter;  void AddNode(int k); // добавление элемента  void PrintTree(Node\* temp, int l); // вывод дерева  void Delete(int k); // удаление узла  bool Balance(Node\* temp); // сбалансированность дерева  void DirectBypass(Node\* temp); // прямой обход  void ReverseBypass(Node\* temp); // смешанный обход  void DownScan(Node\* temp); // обратный обход  int check\_right(Node\* temp); // правые дочерние узлы  }; |
| Консоль отладки |
|  |

Дополнительные задания

1. Оператор мобильной связи организовал базу данных абонентов, содержащую сведения о телефонах, их владельцах и используемых тарифах, в виде бинарного дерева. Разработать программу, которая обеспечивает начальное формирование базы данных в виде бинарного дерева; производит вывод всей базы данных; поиск владельца по номеру телефона; выводит наиболее востребованный тариф (по наибольшему числу абонентов).

|  |
| --- |
| Исходный код |
| #include <iostream>  #include <algorithm>  #include <windows.h>  using namespace std;  struct Tree  {  double key;  char text[20];  int rate;  Tree\* Left, \* Right;  };  Tree\* makeTree(Tree\* Root); //Создание дерева  Tree\* list(double i, char\* s, int rate); //Создание нового элемента  Tree\* insertElem(Tree\* Root, double key, char\* s, int rate); //Добавление нового элемента  Tree\* search(Tree\* n, double key); //Поиск элемента по ключу  Tree\* delet(Tree\* Root, double key); //Удаление элемента по ключу  void view(Tree\* t, int level); //Вывод дерева  void delAll(Tree\* t); //Очистка дерева  void check(int rate);  void outRate();  int r1, r2, r3; // счетчики для тарифов  Tree\* Root = NULL; //указатель корня  int R\_num = 0;  float coun, sum;  void main()  {  SetConsoleCP(1251);  SetConsoleOutputCP(1251);  setlocale(LC\_ALL, ".1251");  double key; int rate, choice, n;  Tree\* rc; char s[20], letter;  for (;;)  {  cout << "1 - создание дерева\n";  cout << "2 - добавление элемента\n";  cout << "3 - поиск по телефону\n";  cout << "4 - удаление элемента\n";  cout << "5 - вывод дерева\n";  cout << "6 - вывод самого популярного тарифа\n";  cout << "7 - очистка дерева\n";  cout << "8 - выход\n";  cout << "Выберите, что сделать:\n";  cin >> choice;  cout << "\n";  switch (choice)  {  case 1: Root = makeTree(Root); break;  case 2: cout << "\nВведите телефон: ";  cin >> key;  cout << "Введите фамилию: "; cin >> s;  cout << "Введите тариф: "; cin >> rate;  insertElem(Root, key, s, rate); break;  case 3: cout << "\nВведите телефон: "; cin >> key;  rc = search(Root, key);  cout << "Владелец = ";  puts(rc->text); break;  case 4: cout << "\nВведите удаляемый номер телефона: "; cin >> key;  Root = delet(Root, key); break;  case 5: if (Root->key >= 0)  {  cout << "Дерево повернуто на 90 градусов влево:" << endl;  view(Root, 0);  }  else cout << "Дерево пустое\n"; break;  case 6: outRate(); break;  case 7: delAll(Root); break;  break;  case 8: exit(0);  }  }  }  Tree\* makeTree(Tree\* Root) //Создание дерева  {  double key, rate; char s[20];  cout << "Для остановки ввода введите отрицательное число:\n\n";  if (Root == NULL) // если дерево не создано  {  cout << "Введите телефон: "; cin >> key;  cout << "Введите Фамилию: "; cin >> s;  cout << "Введите тариф (1,2,3): "; cin >> rate;  check(rate);  Root = list(key, s, rate); // установка указателя на корень  }  while (1) //добавление элементов  {  cout << "\nВведите телефон: ";  cin >> key;  if (key < 0) break; //признак выхода (ключ < 0)  cout << "Введите фамилию: "; cin >> s;  cout << "Введите тариф: "; cin >> rate;  check(rate);  insertElem(Root, key, s, rate);  }  return Root;  }  void check(int rate)  {  if (rate == 1)  r1++;  if (rate == 2)  r2++;  if (rate == 3)  r3++;  }  void outRate()  {  int maximum = max(r1, max(r2, r3));  if (maximum == r1)  cout << "Наиболее популярным тарифом является 1, с количеством абонентов " << r1 << endl;  if (maximum == r2)  cout << "Наиболеепопулярным тарифом является 2, с количеством абонентов " << r2 << endl;  if (maximum == r3)  cout << "Наиболее популярным тарифом является 3, с количеством абонентов " << r3 << endl;  }  Tree\* list(double i, char\* s, int rate) //Создание нового элемента  {  Tree\* t = new Tree[sizeof(Tree)];  t->key = i;  t->rate = rate;  for (i = 0; i < 20; i++)  \*((t->text) + (int)i) = \*(s + (int)i);  t->Left = t->Right = NULL;  return t;  }  Tree\* insertElem(Tree\* t, double key, char\* s, int rate) //Добавление нового элемента  {  Tree\* Prev = NULL; // Prev - элемент перед текущим  int find = 0; // признак поиска  while (t && !find)  {  Prev = t;  if (key == t->key)  find = 1; //ключи должны быть уникальны  else  if (key < t->key) t = t->Left;  else t = t->Right;  }  if (!find) //найдено место с адресом Prev  {  t = list(key, s, rate); //создается новый узел  if (key < Prev->key) // и присоединяется либо  Prev->Left = t; //переход на левую ветвь,  else  Prev->Right = t; // либо на правую  }  return t;  }  Tree\* delet(Tree\* Root, double key) //Удаление элемента по ключу  { // Del, Prev\_Del - удаляемый элемент и его предыдущий ;  // R, Prev\_R - элемент, на который заменяется удаленный, и его родитель;  Tree\* Del, \* Prev\_Del, \* R, \* Prev\_R;  Del = Root;  Prev\_Del = NULL;  while (Del != NULL && Del->key != key)//поиск элемента и его родителя  {  Prev\_Del = Del;  if (Del->key > key)  Del = Del->Left;  else  Del = Del->Right;  }  if (Del == NULL) // элемент не найден  {  puts("\nНет такого ключа");  return Root;  }  if (Del->Right == NULL) // поиск элемента R для замены  R = Del->Left;  else  if (Del->Left == NULL)  R = Del->Right;  else  {  Prev\_R = Del; //поиск самого правого элемента в левом поддереве  R = Del->Left;  while (R->Right != NULL)  {  Prev\_R = R;  R = R->Right;  }  if (Prev\_R == Del) // найден элемент для замены R и его родителя Prev\_R  R->Right = Del->Right;  else  {  R->Right = Del->Right;  Prev\_R->Right = R->Left;  R->Left = Prev\_R;  }  }  if (Del == Root) Root = R; //удаление корня и замена его на R  else  // поддерево R присоединяется к родителю удаляемого узла  if (Del->key < Prev\_Del->key)  Prev\_Del->Left = R; // на левую ветвь  else  Prev\_Del->Right = R; // на правую ветвь  double tmp = Del->key;  cout << "\nУдален элемент с ключом " << tmp << endl;  delete Del;  return Root;  }  Tree\* search(Tree\* n, double key) //Поиск элемента по ключу  {  Tree\* rc = n;  if (rc != NULL)  {  if (key < (key, n->key))  rc = search(n->Left, key);  else  if (key > (key, n->key))  rc = search(n->Right, key);  }  else  cout << "Нет такого элемента\n";  return rc;  }  void view(Tree\* t, int level) //Вывод дерева  {  if (t)  {  view(t->Right, level + 1); //вывод правого поддерева  for (int i = 0; i < level; i++)  cout << " ";  long long tm = long long(t->key);  int tm1 = t->rate;  cout << tm << ' ' << tm1 << ' ';  puts(t->text);  view(t->Left, level + 1); //вывод левого поддерева  }  }  void delAll(Tree\* t) //Очистка дерева  {  if (t != NULL)  {  delAll(t->Left);  delAll(t->Right);  delete t;  }  } |
| Консоль отладки |
|  |

2. Дано N чисел, N > 0. Создать дерево из N вершин, в котором каждая левая дочерняя вершина является листом, а правая дочерняя вершина является внутренней. Для каждой внутренней вершины вначале создавать левую дочернюю вершину, а затем правую (если она существует); каждой создаваемой вершине присваивать очередное значение из исходного набора.

|  |
| --- |
| Исходный код |
| #include <iostream>  #include <algorithm>  using namespace std;  struct Tree  {  int value;  Tree\* left;  Tree\* right;  };  int\* create();  Tree\* makeTree(Tree\* Root, int key);  Tree\* list(int i);  Tree\* insertElem(Tree\* t, int key);  void view(Tree\* t, int level);  int N;  int main()  {  setlocale(LC\_ALL, "rus");  int\* delta = create();  Tree\* Root = new Tree;  Root = makeTree(Root, delta[0]);  for (int i = 1; i < N; i++)  {  insertElem(Root, delta[i]);  }  view(Root, 0);  system("pause");  return 0;  }  int\* create()  {  cout << "Введите количество элементов: "; cin >> N;  int\* mass = new int[N];  for (int i = 0; i < N; i++)  {  cout << "Введите новый элемент массива: "; cin >> mass[i];  }  std::sort(mass, mass + N);  for (int i = 1; i < N; i += 2)  {  int t = mass[i - 1];  mass[i - 1] = mass[i];  mass[i] = t;  }  for (int i = 0; i < N; i++)  {  cout << "Элементы массива: "; cout << mass[i] << endl;  }  return mass;  }  Tree\* list(int i) //Создание нового элемента  {  Tree\* t = new Tree[sizeof(Tree)];  t->value = i;  t->left = t->right = NULL;  return t;  }  Tree\* makeTree(Tree\* Root, int key) //Создание дерева  {  Root = list(key); // установка указателя на корень  return Root;  }  Tree\* insertElem(Tree\* t, int key) //Добавление нового элемента  {  Tree\* Prev = NULL; // Prev - элемент перед текущим  int find = 0; // признак поиска  while (t && !find)  {  Prev = t;  if (key == t->value)  find = 1; //ключи должны быть уникальны  else  if (key < t->value) t = t->left;  else t = t->right;  }  if (!find) //найдено место с адресом Prev  {  t = list(key); //создается новый узел  if (key < Prev->value) // и присоединяется либо  Prev->left = t; //переход на левую ветвь,  else  Prev->right = t; // либо на правую  }  return t;  }  void view(Tree\* t, int level) //Вывод дерева  {  if (t)  {  view(t->right, level + 1); //вывод правого поддерева  for (int i = 0; i < level; i++)  cout << " ";  int tm = t->value;  cout << tm << endl;  view(t->left, level + 1); //вывод левого поддерева  }  } |
| Консоль отладки |
|  |

3. Изучить работу с красно-черными деревьями. Реализовать основные операции над красно-черным деревом. В красно-черном дереве найти путь от корня к некоторому листу, содержащий минимальное количество красных вершин.

|  |
| --- |
| Исходный код |
| /\*  Свойства красно-черных деревьев:  1) Каждый узел окрашен либо в красный, либо в черный цвет (в структуре данных узла появляется дополнительное поле – бит цвета).  2) Корень окрашен в черный цвет.  3) Листья(так называемые NULL-узлы) окрашены в черный цвет.  4) Каждый красный узел должен иметь два черных дочерних узла. Нужно отметить, что у черного узла могут быть черные дочерние узлы. Красные узлы в качестве дочерних могут иметь только черные.  5) Пути от узла к его листьям должны содержать одинаковое количество черных узлов(это черная высота).  \*/  #include <iostream>  #include <string>  #define black 0  #define red 1  using namespace std;  struct Tree //структура элемента дерева  {  int val;  bool color; // доп поле для красно-черных - бит цвета  Tree\* left;  Tree\* right;  Tree\* parent;  } \*head;  int add\_k = 0;  void create(int); //Создание дерева  void add(int); //Функция добавления элементов в дерево  void output(); //Вывод дерева  void check\_1(Tree\*);  void check\_2(Tree\*);  void check\_3(Tree\*);  void check\_4(Tree\*);  void check\_5(Tree\*);  void rotate\_right(Tree\*);  void rotate\_left(Tree\*);  Tree\* Ded(Tree\*);  Tree\* uncle(Tree\*);  int main()  {  create(13); // создаем  add(8); // и закидываем числа  add(17);  add(1);  add(11);  add(15);  add(25);  add(6);  add(22);  add(27);  output();  Tree\* iu = head;  system("pause");  return 0;  }  void create(int data) //Создание дерева (корень черного цвета)  {  add\_k++;  head = new Tree;  head->val = data;  head->left = new Tree; //Левый лист черный  head->left->val = NULL;  head->left->left = NULL;  head->left->right = NULL;  head->left->color = black;  head->right = new Tree; //Правый лист черный  head->right->val = NULL;  head->right->left = NULL;  head->right->right = NULL;  head->right->color = black;  head->color = 0; // корень окрашен в черный  head->parent = NULL;  }  void add(int data) // Добавление элементов  {  add\_k++;  Tree\* kul = head;  Tree\* buff = NULL;  while (kul->val != NULL) //Добавление как в бинароном делеве  {  buff = kul;  if (data > buff->val)  {  kul = kul->right;  }  else  {  kul = kul->left;  }  }  kul->val = data;  kul->left = new Tree; //Левый черный лист  kul->left->val = NULL;  kul->left->left = NULL;  kul->left->right = NULL;  kul->left->color = black;  kul->right = new Tree;//Правый черный лист  kul->right->val = NULL;  kul->right->left = NULL;  kul->right->right = NULL;  kul->right->color = black;  kul->color = red; // Цвет вершины  kul->parent = buff;  if (data < buff->val) // Проверка на какую ветку добавить  {  buff->left = kul;  }  else  {  buff->right = kul;  }  check\_1(kul);  }  Tree\* Ded(Tree\* n) //Проверка на существование элементов 2 колена  {  if ((n != NULL) && (n->parent != NULL))  return n->parent->parent;  else  return NULL;  }  Tree\* uncle(Tree\* n)  {  Tree\* g = Ded(n);  if (g == NULL)  return NULL;  if (n->parent == g->left) //Проверка  return g->right;  else  return g->left;  }  void check\_1(Tree\* n) // Если это первый элемент (head) то цвет черный  {  if (n->parent == NULL)  n->color = black;  else  check\_2(n);  }  void check\_2(Tree\* n) //Проверка родителя на цвет  {  if (n->parent->color == black) //если черный возвращаемся и оставляем его красным  return;  else  check\_3(n);  }  void check\_3(Tree\* n) //Проверка на колчество шагов от вершины до листа  {  Tree\* u = uncle(n), \* g;  if ((u != NULL) && (u->color == red)) {  n->parent->color = black;  u->color = black;  g = Ded(n);  g->color = red;  check\_1(g);  }  else {  check\_4(n);  }  }  void check\_4(Tree\* n)  {  Tree\* g = Ded(n);  if ((n == n->parent->right) && (n->parent == g->left)) {  rotate\_left(n->parent);  n = n->left;  }  else if ((n == n->parent->left) && (n->parent == g->right)) {  rotate\_right(n->parent);  n = n->right;  }  check\_5(n);  }  void check\_5(Tree\* n)  {  Tree\* g = Ded(n);  n->parent->color = black;  g->color = red;  if ((n == n->parent->left) && (n->parent == g->left)) {  rotate\_right(g);  }  else {  rotate\_left(g);  }  }  void rotate\_left(Tree\* n)  {  Tree\* pivot = n->right;  pivot->parent = n->parent;  if (n->parent != NULL) {  if (n->parent->left == n)  n->parent->left = pivot;  else  n->parent->right = pivot;  }  n->right = pivot->left;  if (pivot->left != NULL)  pivot->left->parent = n;  n->parent = pivot;  pivot->left = n;  }  void rotate\_right(Tree\* n)  {  Tree\* pivot = n->left;  pivot->parent = n->parent; /\* при этом, возможно, pivot становится корнем дерева \*/  if (n->parent != NULL) {  if (n->parent->left == n)  n->parent->left = pivot;  else  n->parent->right = pivot;  }  n->left = pivot->right;  if (pivot->right != NULL)  pivot->right->parent = n;  n->parent = pivot;  pivot->right = n;  }  char get\_color(bool x)  {  return (x == true ? 'r' : 'b');  }  void print\_Tree(Tree\* p, int level) //вывод дерева  {  if (p)  {  print\_Tree(p->left, level + 1);  for (int i = 0; i < level; i++) cout << " ";  cout << p->val << get\_color(p->color) << endl;  print\_Tree(p->right, level + 1);  }  }  void output() // Вывод дерева  {  Tree\* temp = new Tree;  temp = head;  while (temp->parent != NULL) //Ищем head  temp = temp->parent;  print\_Tree(temp, 0);  } |
| Консоль отладки |
|  |