**Лабораторная работа 2.** **Создание и обход бинарных деревьев**

**Цель работы: Получение практических навыков объектно-ориентированного программирования задач с использованием бинарных деревьев.**

**Краткие теоретические сведения**

1. **Основные определения**

Для описания различных аспектов деревьев применяются разные термины. На рис. 1 показаны некоторые термины применительно к двоичному дереву.

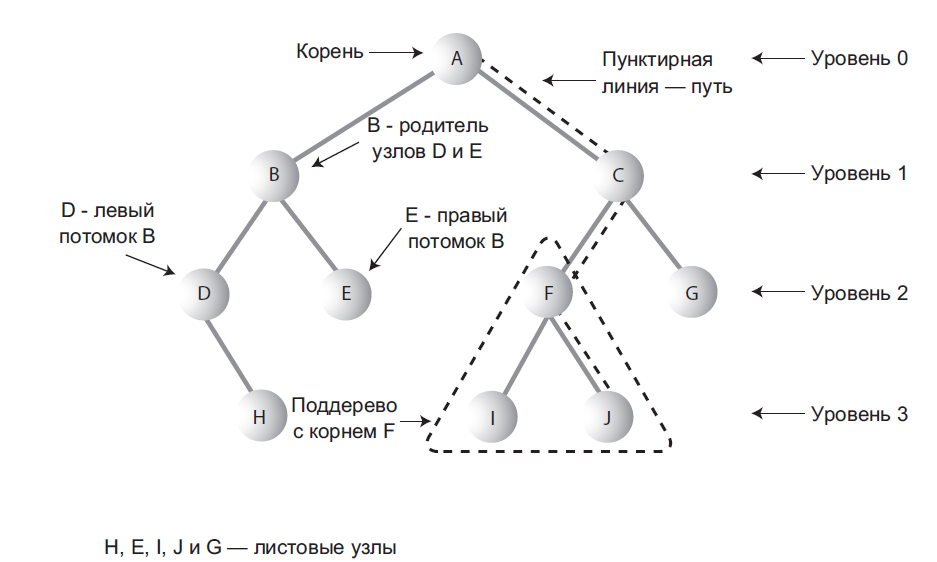


Рис. 1. Термины, используемые при работе с деревьями

**Путь**

Последовательность узлов, полученная при перемещении по дереву от узла к узлу по соединяющим их ребрам называется *путем* (path).

**Корень**

Узел на верхнем уровне дерева называется *корневым узлом* (*корнем*). Дерево имеет только один корень. Чтобы совокупность узлов и ребер могла называться деревом, от корня к любому другому узлу должен вести один (и только один!) путь. Структура на рис. 2, деревом не является, поскольку нарушает это правило.

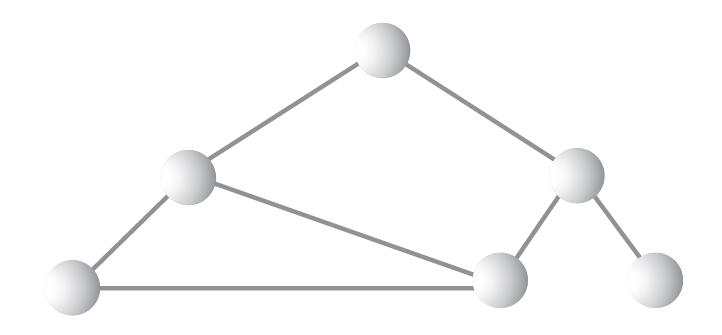


Рис. 2. Структура, не являющаяся деревом

**Родитель (предок)**

Любой узел (кроме корневого) имеет ровно одно ребро, уходящее вверх к другому узлу. Узел, расположенный выше него, называется *родительским узлом* (или просто *родителем или предком)*) по отношению к данному узлу.

**Потомок**

Любой узел может иметь одно или несколько ребер, соединяющих его с узлами более низкого уровня. Такие узлы, находящиеся ниже заданного узла, называются его *потомками*.

**Лист**

Узел, не имеющий потомков, называется *листовым узлом* (или просто *листом*). Дерево всегда имеет только один корень, но листьев может быть несколько.

**Поддерево**

Любой узел может рассматриваться как корень поддерева, состоящего из его потомков, потомков его потомков и т. д.

**Посещение**

Переход программы к узлу (обычно с целью выполнения некоторой операции, например проверки значения одного из полей данных или вывода) называется *посещением*. Простое прохождение мимо узла на пути от одного узла к другому посещением не считается.

**Обход**

*Обходом* дерева называется посещение всех его узлов в некотором заданном порядке. Например, все узлы дерева могут перебираться в порядке возрастания ключей.

**Уровни**

*Уровнем* узла называется количество поколений, отделяющих его от корня. Если считать, что корень находится на уровне 0, то его потомки находятся на уровне 1, потомки потомков — на уровне 2 и т. д.

**Ключи**

Одно из полей данных объекта часто назначается *ключевым*. Ключ используется при поиске элемента или выполнения с ним других операций. На древовидных диаграммах узел, содержащий данные, обычно обозначается кружком, а внутри кружка отображается значение ключа.

**Двоичное дерево**

Если каждый узел дерева имеет не более двух потомков, такое дерево называется *двоичным*.

Два потомка каждого узла двоичного дерева называются *левым потомком* и *правым потомком* в зависимости от позиции на изображении дерева. Число потомков узла двоичного дерева не обязано быть равным 2; узел может иметь только левого или только правого потомка или не иметь потомков вообще (листовой узел).

**Двоичное дерево поиска**

Двоичное дерево, у которого в каждом узле левый потомок меньше предка, а правый потомок больше предка, называется двоичным деревом поиска. Пример двоичного дерева поиска представлен на рис. 3.

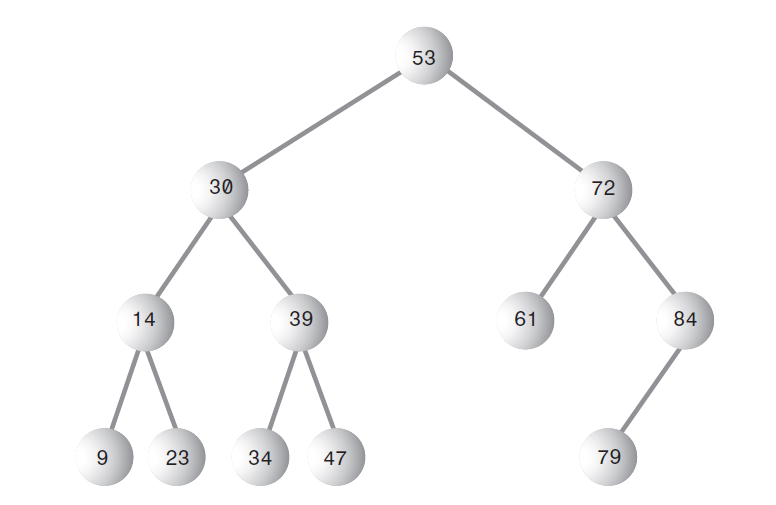


Рис. 3. Дерево двоичного поиска

Основными операциями с двоичными деревьями являются:

* вставка нового узла
* удаление узла
* обход дерева
* поиск узла с заданным ключом.

1. **Основные операции с двоичным деревом**

Дерево в памяти может быть представлено в виде массива или динамической структуры на основе списков; узлы могут храниться как в соответствующих позициях массива так и в случайных участках памяти, связанных между собой ссылками (указателями).

В следующих примерах кода используется модель с соединением узлов посредством ссылок.

**Класс Node (узел дерева)**

Класс должен содержать данные, представляющие хранимые объекты (например, описания товаров в магазине), а также ссылки (указатели) на каждого из двух потомков текущего узла. Определение класса выглядит так:

***class Node***

***{***

***int Key; // Данные, используемые в качестве ключа***

***node leftChild; // Ссылка на левого потомка узла***

***node rightChild; // Ссылка на правого потомка узла***

***}***

Во многих случаях в класс включается также ссылка на родительский узел.

Вместо размещения данных непосредственно в узле можно воспользоваться ссылкой на объект, представляющий набор данных.

**Класс Tree (дерево)**

Класс содержит только одно поле: переменную Node, в которой хранится корень дерева. Поля для других узлов не нужны, поскольку доступ к ним осуществляется через корневой узел. Класс Tree содержит ряд методов для выполнения операций с деревом (поиск, вставка, удаление узлов, различные виды обхода и вывода содержимого дерева). Структура класса выглядит так:

***class Tree***

***{***

***private Node root; // Единственное поле данных***

***public void Insert(int Key)***

***{***

***}***

***public void Delete(int Key)***

***{***

***}***

***public void Find(int Key)***

***{***

***}***

***// Другие методы***

***} // Конец класса Tree***

В классе Tree определены (но пока не реализованы) методы

Insert (вставка узла)

Delete (удаление узла)

Find(поиск узла)

Интерфейс программы для работы с деревом представлен на рисунке 4.

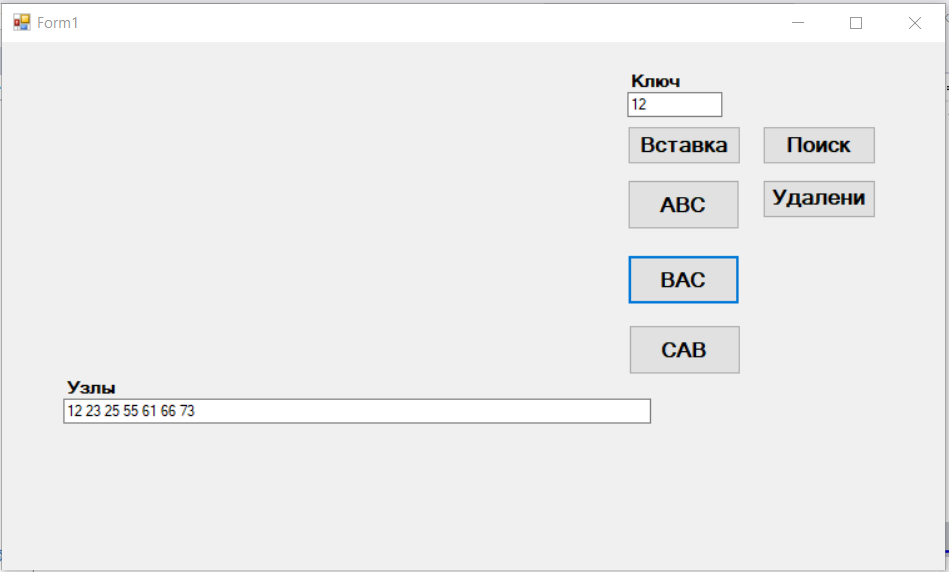


Рисунок 4. Интерфейс программы для работы с деревом

* 1. **Вставка узла**

Для вставки узла необходимо сначала создать экземпляр дерева

***public Tree theTree = new Tree(); // Создание экземпляра дерева***

Чтобы вставить узел, необходимо сначала создать узел, затем найти место для его вставки (найти родителя) и затем вставить узел как левого или правого потомка. Если строится двоичное дерево поиска, то вставляемый узел становится левым потомком если значение его ключа меньше ключа родителя и правым потомком, если значение ключа вставляемого узла больше родительского ключа. Вызов метода Insert может быть осуществлен нажатием кнопки «Вставка» (рис.4)

***private void btInsert\_Click(object sender, EventArgs e)***

***{***

***theTree.Insert(Convert.ToInt32(txtKey.Text)); ;***

***}***

Процедура вставки узла начинается с поиска позиции для вставки. На рис. 5, *а* показано, как это происходит.

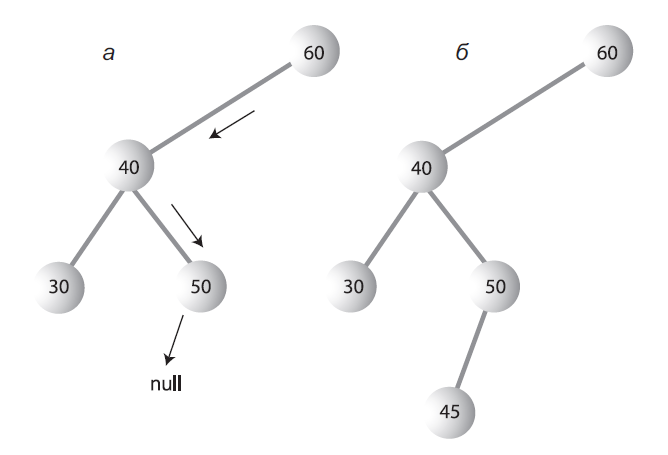


Рис. 5. Вставка узла: а — до вставки; б — после вставки

Значение 45 меньше 60, но больше 40 — переходим к узлу 50. Теперь нужно двигаться налево, потому что 45 меньше 50, но у узла 50 нет левого потомка; его поле leftChild равно null. При обнаружении null алгоритм вставки считает, что место для присоединения нового узла найдено. Приложение создает новый узел с ключом 45 и связывает его с узлом 50 в качестве левого потомка, как показано на рис. 5, *б*.

**Реализация вставки**

Код метода insert может иметь следующий вид

***public void Insert(int key)***

***{***

***//Создание узла***

***Node newNode = new Node();***

***newNode.Key = key;***

***newNode.leftChild = null;***

***newNode.rightChild = null;***

***if (root == null) // Если корневой узел не существует***

***root = newNode;***

***else //корневой узел существует***

***{***

***Node current = root;//определение текущего узла***

***Node parent;***

***while (true)***

***{***

***parent = current; //запоминание родителя***

***if (key < current.Key) //движение налево***

***{***

***current = current.leftChild;***

***if (current == null) //достигнут конец ветви***

***{***

***parent.leftChild = newNode;//вставка узла***

***return;***

***}***

***}***

***else //движение направо***

***{***

***current = current.rightChild;***

***if (current == null) // Если достигнут конец ветви,***

***{***

***parent.rightChild = newNode; //всиавка узла***

***return;***

***}***

***}***

***}***

***}***

***}***

Сначала метод insert() создает новый узел на основе данных c ключом, указанным в поле «Ключ». Далее необходимо определить место для вставки нового узла. Если дерево пустое, то созданный узел становится корневым узлом. Если дерево не пустое, то запускается цикл, в котором выбирается путь для нахождения места для нового узла. Поскольку строится двоичное дерево поиска, в котором для каждого узла выполняется правило «Ключ левого потомка всегда меньше ключа родителя, а ключ правого потомка всегда больше ключа родителя». Поэтому ключ нового узла (key) сравнивается с ключом текущего узла и выбирается направление перехода к следующему потомку – налево или направо. Выбранный потомок становится текущим узлом, и операция повторяется до тех пор, пока не будет найден «пустой» потомок (current=null). Это означает, что место для нового узла найдено и для его вставки в дерево необходимо только ссылку на соответствующего потомка в родительском узле установить на новый узел. (parent.leftChild = newNode; или parent.rightChild = newNode;)

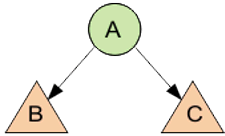
Место для вставки нового узла всегда находится успешно (если в системе хватит памяти); когда это произойдет, новый узел включается в дерево, а выполнение цикла while завершается командой return.

**2.2. Обход дерева**

Обходом дерева называется посещение всех его узлов в определенном порядке. На практике обход используется не так часто, как поиск, вставка и удаление узлов. Одна из причин заключается в том, что алгоритмы обхода не отличаются быстротой. Однако обход дерева бывает полезным в некоторых обстоятельствах, и он представляет интерес с теоретической точки зрения.

Существуют три простых алгоритма обхода дерева:

* *прямой* (preorder, префиксный), ABC
* *симметричный* (inorder, инфиксный) BAC
* *обратный* (postorder, постфиксный). CAB



**Симметричный обход**

При симметричном обходе двоичного дерева поиска все узлы перебираются в порядке возрастания ключей. Если требуется создать отсортированный список данных двоичного дерева — это одно из возможных решений. Простейший способ обхода основан на использовании рекурсии. При вызове рекурсивного метода для обхода всего дерева в аргументе передается узел. В исходном состоянии этим узлом является корень дерева. Метод должен выполнить только три операции:

1. Вызов самого себя для обхода левого поддерева узла.

2. Посещение узла.

3. Вызов самого себя для обхода правого поддерева узла.

*Посещение* узла подразумевает выполнение некоторой операции: вывод данных, запись в файл и т. д.

Обход работает с любым двоичным деревом, не только с деревьями двоичного поиска. Алгоритм обхода не обращает внимания на значения ключей; его интересует только наличие у узла потомков.

**Реализация симметричного обхода**

Метод inOrder(), реализующий симметричный обход выполняет три операции, упоминавшиеся ранее. Код метода может иметь следующий вид.

Код метода inOrder():

***private void inOrder(node localRoot)***

***{***

***if(localRoot != null)***

***{***

***inOrder(localRoot.leftChild);***

***txtnodes+=(localRoot.Key + " ");***

***inOrder(localRoot.rightChild);***

***}***

***}***

Посещение узла сводится к отображению его содержимого в поле «Узлы». Как и любая рекурсивная функция, метод должен обладать базовым ограничением — условием, при котором он немедленно возвращает управление без рекурсивного вызова. В методе inOrder() это происходит при передаче аргумента null.

В исходном вызове метода в аргументе передается корневой узел root:

***inOrder(theTree.root);***

Далее метод действует самостоятельно, рекурсивно вызывая самого себя до тех пор, пока не останется узлов для обхода.

**Прямой и oбратный обход**

Кроме симметричного обхода, также существует два других алгоритма обхода дерева, называемые прямым и обратным.

Код для прямого и обратного обходов может иметь вид:

***public void PreOrder(Node localRoot)***

***{***

***if (localRoot != null)***

***{***

***txtnodes += (localRoot.Key + " ");***

***PreOrder(localRoot.leftChild);***

***PreOrder(localRoot.rightChild);***

***}***

***}***

***public void PostOrder(Node localRoot)***

***{***

***if (localRoot != null)***

***{***

***PostOrder(localRoot.leftChild);***

***PostOrder(localRoot.rightChild);***

***txtnodes += (localRoot.Key + " ");***

***}***

***}***

**Поиск минимума и максимума**

Поиск минимального и максимального значения в дереве двоичного поиска выполняется тривиально. Чтобы получить минимальное значение ключа в дереве необходимо перейти от корня к левому потомку; затем перейдите к левому потомку этого потомка и т. д., пока не встретится узел, не имеющий левого потомка. Этот узел и содержит минимальное значение ключа (рис. 8.).

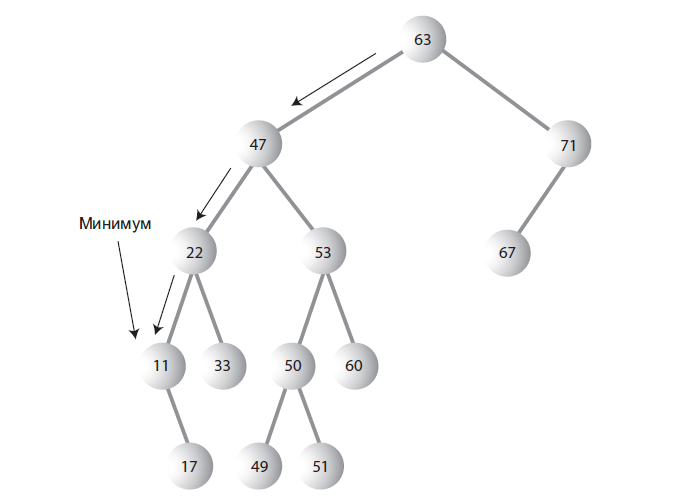


Рис. 6. Поиск минимума по дереву

Следующий метод возвращает узел с минимальным значением ключа:

***public Node minimum() // Возвращает узел с минимальным ключом***

***{***

***Node current, last;***

***current = root; // Обход начинается с корневого узла***

***while(current != null) // и продолжается до низа***

***{***

***last = current; // Сохранение узла***

***current = current.leftChild; // Переход к левому потомку***

***}***

***return last;***

***}***

Минимальное значение пригодится при удалении узла.

Поиск максимума в дереве выполняется аналогичным образом, но вместо перехода налево следует переходить от правого потомка к правому потомку, пока не будет найден узел, не имеющий правого потомка. Этот узел и содержит максимальное значение ключа. Код метода выглядит точно так же, не считая того, что последняя команда в цикле принимает вид

***current = current.rightChild; // Переход к правому потомку***

* 1. **Поиск узла**

Поиск узла с заданным ключом — простейшая из основных операций с деревьями.

Узлы в дереве двоичного поиска соответствуют объектам, содержащим информацию. Это могут быть объекты person с ключевым полем табельного номера, а также полями имени, адреса, телефона, зарплаты и т. д., или объекты, представляющие детали машин, с ключевым полем кода детали и полями текущего количества, цены и т. д. Узел создается с этими характеристиками и сохраняет их на всем протяжении своего существования. Например, пусть требуется найти узел, представляющий элемент с ключом 57 (рис. 7). .

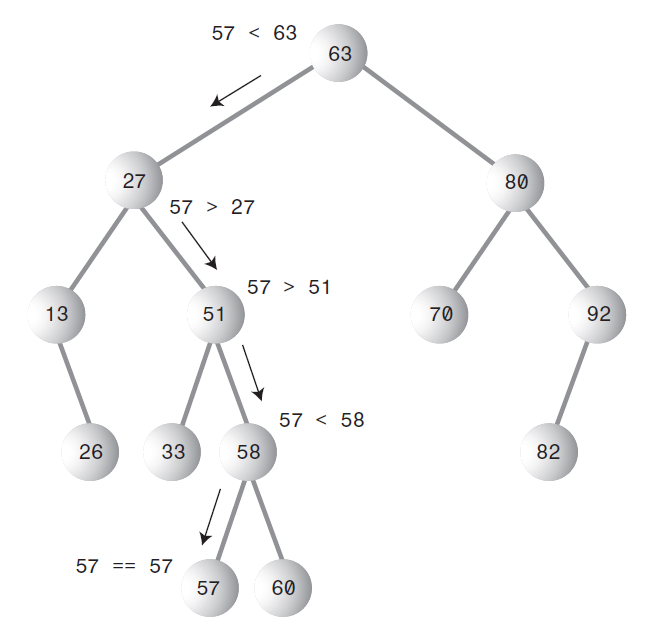


Рис. 7. Поиск узла 57

На рис. 7 стрелками показано перемещение по дереву от корневого узла. Программа сравнивает ключ 57 с ключом корневого узла, равным 63. Искомый ключ меньше, поэтому программа знает, что искомый узел находится в левой части дерева — либо это левый потомок корневого узла, либо один из его потомков. Ключ левого потомка корневого узла равен 27; сравнение 57 и 27 показывает, что искомый узел принадлежит правому поддереву узла 27. Стрелка переходит к узлу 51 — корню этого поддерева. Значение 57 больше 51, поэтому процесс поиска сначала переходит направо к 58, а затем налево к 57. На этот раз ключ 57 совпадает с искомым значением — узел успешно найден.

В реальных программах с найденным узлом обычно выполняется какая-нибудь операция — вывод его содержимого, изменение одного из полей и т. д.

**Реализация поиска узла**

Код функции поиска find(), оформленной в виде метода класса Tree:

***public Node find(int key) // Поиск узла с заданным ключом***

***{ // (предполагается, что дерево не пустое)***

***Node current = root; // Начать с корневого узла***

***while (current.Key != key) // Пока не найдено совпадение***

***{***

***if (key < current.Key) // Двигаться налево?***

***current = current.leftChild;***

***else***

***current = current.rightChild; // Или направо?***

***if (current == null) // Если потомка нет,***

***return null; // поиск завершился неудачей***

***}***

***return current; // Элемент найден***

***}***

Для хранения текущего проверяемого узла используется переменная current. Искомое значение хранится в аргументе key. Поиск начинается с корневого узла (это необходимо, так как в дереве напрямую доступен только этот узел), то есть в начале работы current присваивается ссылка на корневой узел. Затем в цикле while искомое значение key сравнивается со значением поля Key (ключевого поля) текущего узла. Если значение Key меньше, то current присваивается ссылка на левого потомка, а если больше (или равно) — то ссылка на правого потомка узла.

**Неудачный поиск**

Если ссылка current становится равной null, значит, найти следующего потомка не удалось; перебор достиг конца дерева, искомый узел не найден, а следовательно, не существует. Метод сообщает об этом факте, возвращая null.

**Узел успешно найден**

Если условие цикла while нарушено, то есть выполнение продолжается после тела цикла, поле Keya объекта current равно key; это означает, что искомый узел был успешно найден. Метод возвращает узел, чтобы код, вызвавший find(), смог обратиться к полям этого узла.

Код обработчика события «нажатие кнопки ПОИСК» может иметь следующий вид

***private void btFind\_Click(object sender, EventArgs e)***

***{***

***int key = Convert.ToInt32(txtKey.Text);***

***if (theTree.find(key) == null)***

***MessageBox.Show("Узел не найден");***

***else***

***MessageBox.Show("Узел найден");***

***}***

**Эффективность поиска по дереву**

Как видно из описания, время поиска узла зависит от количества уровней. Это время *O*(log2 *N* ) (логарифм по основанию 2).

* 1. **Удаление узла**

Удаление узлов является самой сложной из стандартных операций с деревьями двоичного поиска. Удаление начинается с поиска удаляемого узла — процедура поиска уже была продемонстрирована ранее в методах find() и insert(). Когда узел будет найден, необходимо рассмотреть три возможных случая:

1. Удаляемый узел является листовым (не имеет потомков).

2. Удаляемый узел имеет одного потомка.

3. Удаляемый узел имеет двух потомков.

Первый случай совсем прост; второй почти так же прост; с третьим дело обстоит сложнее.

**Случай 1. Удаляемый узел не имеет потомков**

Чтобы удалить листовой узел, достаточно изменить поле соответствующего потомка в родительском узле, сохранив в нем null вместо ссылки на узел. Узел продолжает существовать, но перестает быть частью дерева (рис. 8).

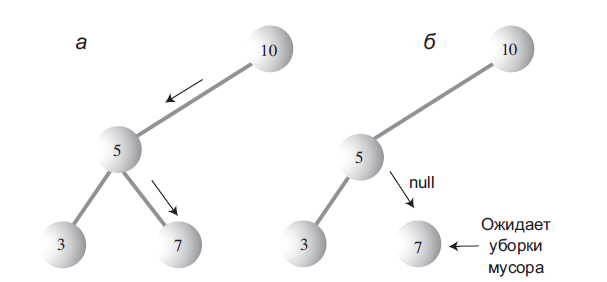


Рис. 8. Удаление узла, не имеющего потомков: *а* — до удаления; *б* — после удаления

Благодаря механизму уборки мусора, используемому в C# и Java, не нужно беспокоиться об уничтожении самого узла. Когда уборщик мусора поймет, что программа не содержит ни одной ссылки на узел, последний будет удален из памяти.

(В языках C и C++ пришлось бы выполнить команду free() или delete() для удаления узла из памяти.)

**Удаление листового узла**

Начало метода delete() почти не отличается от методов find() и insert(). Удаляемый узел необходимо прежде всего найти. Как и в случае с insert(), необходимо запомнить родителя удаляемого узла для изменения полей его потомков. Если узел будет найден, цикл while прерывается; при этом parent содержит удаляемый узел. Если поиск оказался неудачным, delete() просто возвращает значение false.

***public boolean delete(int key) // Удаление узла с заданным ключом***

***{ // (предполагается, что дерево не пусто)***

***Node current = root;***

***Node parent = root;***

***boolean isLeftChild = true;***

***while(current.iData != key) // Поиск узла***

***{***

***parent = current;***

***if(key < current.iData) // Двигаться налево?***

***{***

***isLeftChild = true;***

***current = current.leftChild;***

***}***

***else // Или направо?***

***{***

***isLeftChild = false;***

***current = current.rightChild;***

***}***

***if(current == null) // Конец цепочки***

***return false; // Узел не найден***

***}***

***// Удаляемый узел найден***

***// Продолжение...***

***}***

Когда узел будет найден, необходимо сначала убедиться в том, что у него нет потомков. Если условие выполнено, проверяется особый случай корневого узла. Если удаляемый узел является корневым, нужно просто присвоить ему null, что приводит к полной очистке дерева. В противном случае полю leftChild или rightChild родителя присваивается null, чтобы отсоединить узел от родителя.

***// Продолжение delete()...***

***// Если узел не имеет потомков, он просто удаляется.***

***if(current.leftChild==null && current.rightChild==null)***

***{***

***if(current == root) // Если узел является корневым,***

***root = null; // дерево очищается***

***else if(isLeftChild)***

***parent.leftChild = null; // Узел отсоединяется***

***else // от родителя***

***parent.rightChild = null;***

***}***

***// Продолжение...***

**Случай 2. Удаляемый узел имеет одного потомка**

Второй случай тоже обходится без особых сложностей. Узел имеет только две связи: с родителем и со своим единственным потомком. Требуется «вырезать» узел из этой цепочки, соединив родителя с потомком напрямую. Для этого необходимо изменить соответствующую ссылку в родителе (leftChild или rightChild), чтобы она указывала на потомка удаляемого узла. Ситуация показана на рис. 9.

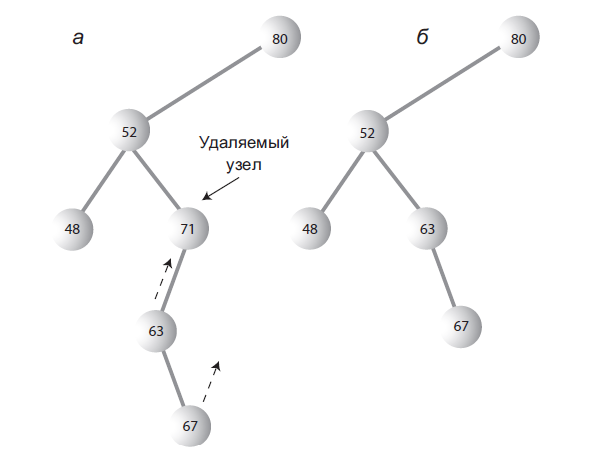


Рис. 9. Удаление узла с одним потомком: *а* — до удаления; *б* — после удаления

**Удаление узла с одним потомком**

Следующий код демонстрирует обработку ситуации с одним потомком. Возможны четыре варианта: потомок удаляемого узла может быть левым или правым, и в каждом из этих случаев удаляемый узел может быть левым или правым потомком своего родителя.

Также возможен особый случай: если удаляется корневой узел, не имеющий родителя, он просто заменяется соответствующим поддеревом. Приведенный ниже код следует за кодом удаления листовых узлов из предыдущего подраздела:

***// Продолжение delete()...***

***// Если нет правого потомка, узел заменяется левым поддеревом***

***else if(current.rightChild==null)***

***if(current == root)***

***root = current.leftChild;***

***else if(isLeftChild) // Левый потомок родителя***

***parent.leftChild = current.leftChild;***

***else // Правый потомок родителя***

***parent.rightChild = current.leftChild;***

***// Если нет левого потомка, узел заменяется правым поддеревом***

***else if(current.leftChild==null)***

***if(current == root)***

***root = current.rightChild;***

***else if(isLeftChild) // Левый потомок родителя***

***parent.leftChild = current.rightChild;***

***else // Правый потомок родителя***

***parent.rightChild = current.rightChild;***

***// Продолжение...***

Обратите внимание, насколько ссылки упрощают удаление целых поддеревьев. Для этого достаточно отсоединить старую ссылку от поддерева и создать новую ссылку на него в другом месте. Хотя поддерево может содержать большое количество узлов, не приходится удалять их по отдельности. Более того, узлы «перемещаются» только в том смысле, что концептуально они занимают новое положение по отношению к другим узлам. С точки зрения программы изменилась только ссылка на корень поддерева.

**Случай 3. Удаляемый узел имеет двух потомков**

Если удаляемый узел имеет двух потомков, нельзя просто заменить его одним из этих потомков (по крайней мере если потомок имеет собственных потомков). Почему? Взгляните на рис. 10 и представьте, как удаляемый узел 25 заменяется своим правым поддеревом с корнем 35. Какой узел должен быть левым потомком узла 35? Левый потомок удаленного узла 15 или левым потомком нового узла 30? В обоих случаях узел 30 окажется не на своем месте, но и просто выбросить его из дерева тоже нельзя.

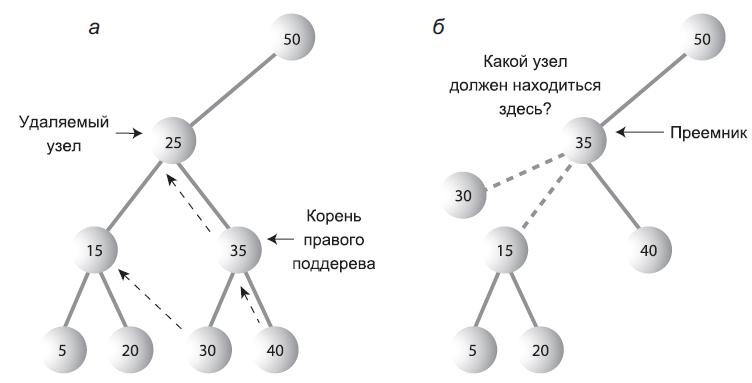
**

Рис. 10. Простая замена поддеревом невозможна: *а* — до удаления; *б* — после удаления

К счастью, существует полезный прием, который поможет справиться с проблемой. С другой стороны, даже при использовании этого приема приходится учитывать множество особых случаев. Вспомните, что мы работаем с деревом двоичного поиска, в котором узлы располагаются в порядке возрастания ключей. Для каждого узла узел со следующим по величине ключом называется его *преемником*. Так, на рис. 10, *а* узел 30 является преемником узла 25. Итак, обещанный прием: чтобы удалить узел с двумя потомками, *нужно заменить его преемником*. На рис. 11 изображена замена удаленного узла преемником. Обратите внимание на то, что порядок узлов при этом не нарушился.

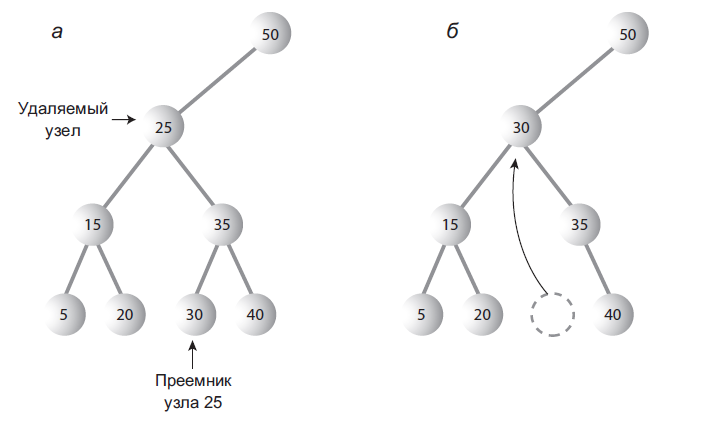


Рис. 11. Замена узла преемником: *а* — до удаления; *б* — после удаления

**Поиск преемника**

Как найти преемника узла? Человек сделает это очень быстро (по крайней мере для небольших деревьев) — ему достаточно взглянуть на дерево и найти число, следующее по величине за ключом удаляемого узла. На рис. 11 сразу видно, что преемником узла 25 является узел 30. В дереве нет другого числа, большего 25 и меньшего 35. Однако компьютер «взглянуть» не может; ему нужен алгоритм. Сначала программа переходит к правому потомку исходного узла, ключ которого должен быть больше ключа узла. Затем она переходит к левому потомку правого потомка (если он существует), к левому потомку левого потомка и т. д., следуя вниз по цепочке левых потомков. Последний левый потомок на этом пути является преемником исходного узла (рис. 12).

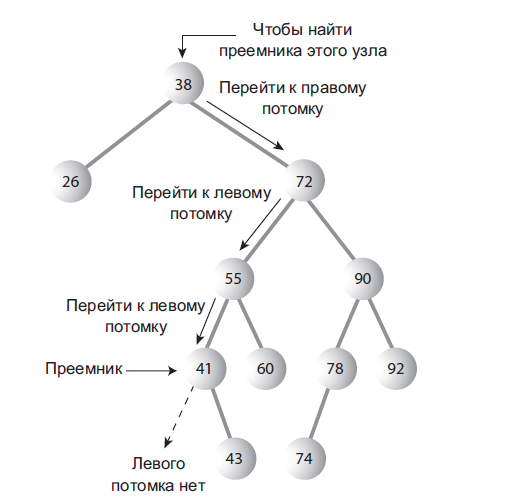


Рис. 12. Поиск преемника

Фактически мы ищем *наименьший узел* в наборе узлов, *больший исходного узла*. В поддереве правого потомка исходного узла все узлы больше исходного узла, что следует из самого определения дерева двоичного поиска. В этом дереве ищется наименьшее значение. Как говорилось выше, минимальный узел поддерева находится отслеживанием пути, состоящего из левых потомков. Таким образом, алгоритм находит минимальное значение, большее исходного узла, которое и является преемником удаляемого узла в соответствии с определением.

Если у правого потомка исходного узла нет левых потомков, то сам правый потомок становится преемником (рис. 13).

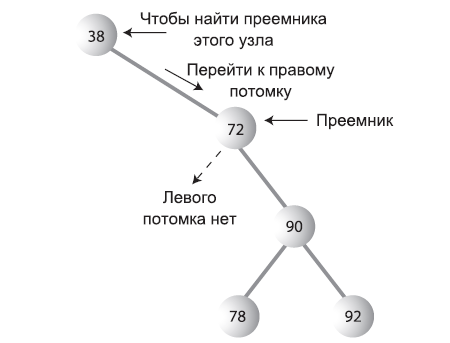


Рис. 13. Правый потомок становится преемником

**Реализация поиска преемника**

Ниже приведен фрагмент кода метода getSuccessor(), который возвращает преемника узла, переданного в аргументе delNode. (Метод подразумевает, что у delNode существует правый потомок, но это условие заведомо выполняется — предыдущая проверка определила, что удаляемый узел имеет двух потомков.)

***// Метод возвращает узел со следующим значением после delNode.***

***// Для этого он сначала переходит к правому потомку, а затем***

***// отслеживает цепочку левых потомков этого узла.***

***private node getSuccessor(node delNode)***

***{***

***Node successorParent = delNode;***

***Node successor = delNode;***

***Node current = delNode.rightChild; // Переход к правому потомку***

***while(current != null) // Пока остаются левые потомки***

***{***

***successorParent = successor;***

***successor = current;***

***current = current.leftChild; // Переход к левому потомку***

***}***

***// Если преемник не является***

***if(successor != delNode.rightChild) // правым потомком,***

***{ // создать связи между узлами***

***successorParent.leftChild = successor.rightChild;***

***successor.rightChild = delNode.rightChild;***

***}***

***return successor;***

***}***

Метод сначала переходит к правому потомку delNode, а затем в цикле while проходит по цепочке левых потомков этого правого потомка. При выходе из цикла while переменная successor содержит преемника delNode. При обнаружении преемника также может возникнуть необходимость в обращении к его родителю, поэтому в цикле while также отслеживается родитель текущего узла.

Метод getSuccessor() также выполняет две дополнительные операции, помимо поиска преемника. Однако чтобы понять их смысл, необходимо немного отойти и взглянуть на «общую картину».

Узел-преемник может занимать одну из двух возможных позиций относительно удаляемого узла current: он может быть его правым потомком или входить в цепочку левых потомков его правого потомка. Рассмотрим каждую из этих ситуаций.

**Преемник является правым потомком delNode**

Если successor является правым потомком current, ситуация немного упрощается, потому что мы можем просто переместить все поддерево, корнем которого является преемник, и вставить его на место удаленного узла. Эта операция выполняется всего за два шага:

1. Отсоединить current от поля rightChild (или leftChild) его родителя. Сохранить в поле ссылку на преемника.

2. Отсоединить левого потомка current от current и сохранить ссылку на него в поле leftChild объекта successor.

В коде delete() эти операции выполняются следующими командами:

1. parent.rightChild = successor;

2. successor.leftChild = current.leftChild;

На рис. 14 показано, как изменяются ссылки при выполнении этих двух команд.

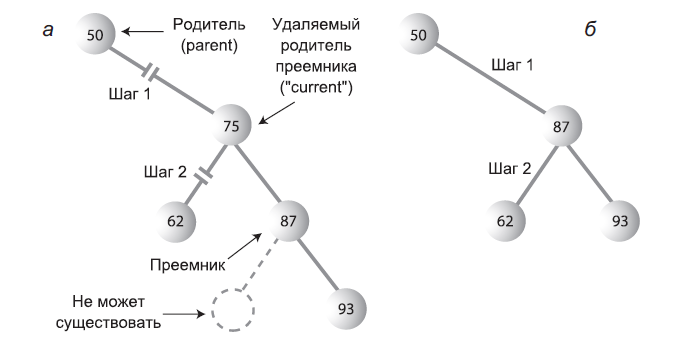
**

Рис. 14. Удаление в случае, когда преемник является правым потомком:

*а* — до удаления; *б* — поле удаления

Вот как выглядит код в контексте (продолжение цепочки else-if, приведенной ранее):

***// Продолжение delete()***

***else // Два потомка, узел заменяется преемником***

***{***

***// Поиск преемника для удаляемого узла (current)***

***Node successor = getSuccessor(current);***

***// Родитель current связывается с посредником***

***if(current == root)***

***Удаление узла 377***

***root = successor;***

***else if(isLeftChild)***

***parent.leftChild = successor;***

***else***

***parent.rightChild = successor;***

***// Преемник связывается с левым потомком current***

***successor.leftChild = current.leftChild;***

***} // Конец else для двух потомков***

***// (преемник не может иметь левого потомка)***

***return true;***

***} // Конец метода delete()***

Обратите внимание: мы (наконец-то) добрались до конца метода delete(). Проанализируем код этих двух операций:

* Шаг 1: если удаляемый узел current является корневым, то он не имеет родителя, поэтому root просто присваивается successor. В противном случае удаляемый узел может быть либо левым, либо правым потомком (на рис. 16 это правый потомок), поэтому в соответствующем поле его родителя сохраняется ссылка на successor. Когда delete() возвращает управление, а current выходит из области видимости, на узел, на который ссылается current, не остается ни одной ссылки, поэтому он будет уничтожен в ходе уборки мусора.
* Шаг 2: в поле левого потомка преемника сохраняется ссылка на левого потомка

current.

Что произойдет, если у преемника имеются свои потомки? Прежде всего, узел преемника заведомо не имеет левого потомка. Это утверждение истинно в любом случае — и если преемник является правым потомком удаляемого узла, и если он является одним из левых потомков правого потомка. Откуда это известно? Вспомните, что алгоритм поиска преемника сначала переходит к правому потомку, а затем перебирает всех его левых потомков. Перебор прекращается при достижении узла, не имеющего левого потомка, так что преемник не может иметь левых потомков в соответствии с алгоритмом. Если бы у него был левый потомок, то последний и стал бы преемником.

С другой стороны, ничто не мешает преемнику иметь правого потомка. Это не создает особых проблем, когда преемник является правым потомком удаляемого узла. При перемещении преемника его правое поддерево просто перемещается вместе с ним. С правым потомком удаляемого узла конфликтов тоже нет, потому что этим правым потомком является сам преемник.

**Преемник входит в число левых потомков правого потомка delNode**

Если преемник входит в число левых потомков правого потомка удаляемого узла, то удаление выполняется за четыре шага:

1. Сохранить ссылку на правого потомка преемника в поле leftChild родителя преемника.

2. Сохранить ссылку на правого потомка удаляемого узла в поле rightChild преемника.

3. Убрать current из поля rightChild его родителя и сохранить в этом поле ссылку на преемника successor.

4. Убрать ссылку на левого потомка current из объекта current и сохранить ее в поле leftChild объекта successor.

Шаги 1 и 2 выполняются методом getSuccessor(), а шаги 3 и 4 выполняются в delete(). На рис. 15 показано изменение ссылок в процессе их выполнения.

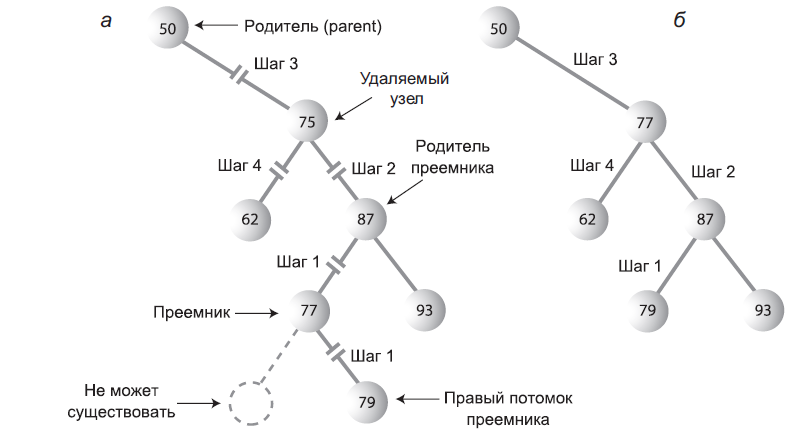


Рис. 15. Удаление в случае, когда преемник является левым потомком:

*а* — до удаления; *б* — после удаления

А вот как выглядит код реализации этих четырех шагов:

1. successorParent.leftChild = successor.rightChild;

2. successor.rightChild = delNode.rightChild;

3. parent.rightChild = successor;

4. successor.leftChild = current.leftChild;

(На шаге 3 также может использоваться левый потомок parent.) Числа на рис. 17 обозначают связи, изменяемые на каждом из четырех шагов. Шаг 1 фактически заменяет преемника его правым поддеревом. Шаг 2 оставляет правого потомка удаляемого узла на положенном месте (это происходит автоматически, когда преемник является правым потомком удаляемого узла). Шаги 1 и 2 выполняются в команде if, завершающей метод getSuccessor(). Еще раз приведем соответствующий фрагмент:

***// Если преемник не является***

***if(successor != delNode.rightChild) // правым потомком,***

***{ // создать связи между узлами***

***successorParent.leftChild = successor.rightChild;***

***successor.rightChild = delNode.rightChild;***

***}***

***return successor;***

***}***

Эти действия удобнее выполнять здесь, чем в delete(), потому что в getSuccessor() мы можем легко определить местонахождение родителя преемника в ходе перемещения по дереву в процессе поиска преемника.

Шаги 3 и 4 мы уже видели; они совпадают с шагами 1 и 2 в случае, когда преемник является правым потомком удаляемого узла, а соответствующий код находится в команде if в конце delete().

Итак, удаление узлов — весьма непростая операция. Она настолько сложна, что некоторые программисты предпочитают обходиться без нее. Они включают в класс node новое поле логического типа с именем вида isDeleted. Чтобы удалить узел, они просто присваивают этому полю значение true. Другие операции — такие, как find(), — прежде чем работать с узлом, проверяют это поле и убеждаются в том, что узел не помечен как удаленный. При таком подходе удаление узла не изменяет структуру дерева. Конечно, это также означает, что память может заполняться «удаленными» узлами.

Такой подход выглядит компромиссно, но он может оказаться подходящим при относительно небольшом количестве удалений из дерева. (Например, если данные бывших работников остаются в базе данных отдела кадров навсегда.)

**Задание на лабораторную работу.**

* + 1. Создать приложение WindowsForms для работы с бинарным дерево поиска, содержащим не менее 4 уровней, заполняя элементы дерева целыми числами.
    2. Вывести на экран дерево (все узлы дерева).
    3. Реализовать в дереве методы в соответствии с вариантом
    4. Создать программу для выполнения операций с деревом в соответствии с вариантом и отобразить выполнение этой программы на экране.

|  |  |  |
| --- | --- | --- |
| **Вариант** | **Методы** | **Операция с деревом** |
| 1 | 1, 3, 7 | Определить сумму всех чисел, находящихся в узлах дерева. |
| 2 | 2, 5, 8 | Определить число узлов в дереве, у которых есть только один потомок. |
| 3 | 4, 5, 8 | Удалить из дерева ветвь с элементом, имеющим заданное число |
| 3 | 1, 3, 7 | Найти среднее значение всех чисел в элементах дерева и узел, имеющий ближайшее к этому значение. |
| 5 | 2, 5, 6 | Определить максимальную глубину дерева, т. е. число узлов в самом длинном пути от корня дерева до листьев. |
| 6 | 1, 4, 7 | Удалить из левого поддерева дерева узел с максимальным значением числа и всех его потомков |
| 7 | 2, 5, 6 | Определить минимальную глубину дерева, т. е. число узлов в самом длинном пути от корня дерева до листьев. |
| 8 | 2, 8, 9 | Определить сумму чисел в правом поддереве. |
| 9 | 1, 4, 6 | Определить количество узлов в левом поддереве. |
| 10 | 1, 3, 6 | Определить количество потомков узлов на втором уровне |
| 11 | 2, 8, 9 | Подсчитать число листьев в дереве (Лист – это узел, из которого нет ссылок на другие узлы дерева). |
| 12 | 3, 5, 8 | Найти узел деревом со значением следующим по величине после минимального значения. |
| 13 | 1, 2, 5 | Определить число узлов в дереве, у которых есть два потомка. |
| 14 | 4, 5, 7 | Поменять местами элементы, содержащие максимальное и минимальное числа. |
| 15 | 1,4, 7 | Определить число уровней в дереве |

Обозначения методов дерева

1. Вставка нового узла с подсчетом количества операций сравнения.
2. Прямой обход дерева с выводом значений ключей.
3. Симметричный обход дерева выводом значений ключей.
4. Обратный обход дерева выводом значений ключей.
5. Поиск узла с заданным ключом с подсчетом количества операций сравнения
6. Удаление узла без потомков.
7. Удаление узла с одним потомком.
8. Поиск преемника удаляемого узла.
9. Удаление узла с двумя потомками.

**1.3. Содержание отчета по лабораторной работе**

- наименование лабораторной работы и ее цель;

- задание на лабораторную работу согласно варианту;

- листинг приложения для работы с двоичным деревом поиска;

- результаты работы программы

-результат оценки сложности алгоритма

Контрольные вопросы

1. Что такое бинарное дерево?
2. Что такое бинарное дерево поиска?
3. Какова последовательность перемещений по дереву при прямом обходе бинарного дерева?
4. Какова последовательность перемещений по дереву при обратном обходе бинарного дерева?
5. Какова последовательность перемещений по дереву при симметричном обходе бинарного дерева?
6. Какова сложность (в O-нотации) операций вставки и удаления в дереве?
7. Какова сложность алгоритма поиска в бинарном дереве поиска?