Министерство образования и науки Российской Федерации

Федеральное агентство по образованию

Федеральное государственное образовательное учреждение
высшего профессионального образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

М. Э. Абрамян

Бинарные деревья

Задачи, решения, указания

Ростов-на-Дону 2009

Печатается по решению
учебно-методической комиссии
факультета математики, механики и компьютерных наук ЮФУ
от 27 апреля 2009 г. (протокол № 8)

Р е ц е н з е н т:
к. ф.-м. н., доцент С. С. Михалкович

Аннотация

Учебное пособие «Бинарные деревья» состоит из четырех модулей. Модуль № 1 посвящен анализу содержимого дерева, модуль № 2 — формированию дерева с заданной структурой и преобразованию существующего дерева, в модуле № 3 рассматриваются особенности деревьев с обратной связью и деревьев поиска, а в модуле № 4 — особенности деревьев разбора выражений и деревьев общего вида. Наряду с базовыми сведениями о бинарных деревьях в пособии приводятся решения типовых задач. Кроме того, пособие содержит формулировки 100 учебных заданий, выполнение которых позволит закрепить изученный материал. Все задания снабжены указаниями.

Пособие предназначено для преподавателей программирования, старшеклассников и студентов.

Автор: М. Э. Абрамян.

Предисловие

Предлагаемое пособие посвящено теме «Бинарные деревья». Данная тема является естественным продолжением тем «Рекурсия» и «Динамические структуры данных», поскольку, с одной стороны, деревья являются динамическими структурами (подобно линейным структурам — стекам, очередям и спискам, — рассмотренным в [3]), а с другой стороны, при обработке деревьев, как правило, используются рекурсивные алгоритмы.

В соответствии с модульной технологией организации учебного материала пособие разбито на 4 модуля, которые включают как основное содержание, так и контрольные разделы (проектное задание и тесты рубежного контроля), позволяющие определить уровень усвоения материала. Модуль № 1 посвящен анализу содержимого дерева, модуль № 2 — формированию дерева с заданной структурой и преобразованию существующего дерева, в модуле № 3 рассматриваются особенности деревьев с обратной связью и деревьев поиска, а в модуле № 4 — особенности деревьев разбора выражений и деревьев общего вида. Наряду с базовыми сведениями о бинарных деревьях в пособии приводятся решения типовых задач. Кроме того, пособие содержит формулировки 100 учебных заданий, выполнение которых позволит закрепить изученный материал. Все задания снабжены указаниями. Заметим, что приведенный набор заданий можно рассматривать как дополнение к ранее опубликованному сборнику задач по программированию [1–3]. Наличие большого количества учебных задач позволило представить проектные задания к каждому модулю в виде набора из 24 вариантов, что дает возможность преподавателю снабдить каждого учащегося отдельным вариантом проектного задания.

В качестве языка программирования используется Паскаль, а точнее, его вариант, реализованный в системах Borland Delphi, Free Pascal Lazarus, Pascal ABC и PascalABC.NET (система Pascal ABC разработана доц. С. С. Михалковичем; он же является руководителем группы разработчиков системы PascalABC.NET).

Все задания, приведенные в настоящем методическом пособии, включены в электронный задачник по программированию Programming Taskbook. Использование электронного задачника существенно ускоряет процесс выполнения заданий, так как избавляет учащегося от дополнительных усилий по организации ввода/вывода и обеспечивает отображение исходных и результирующих данных в наглядном виде, что особенно удобно при обработке таких сложных структур, как деревья. Предоставляя учащемуся готовые исходные данные, задачник акцентирует его внимание на разработке и программной реализации

алгоритма решения задания, причем разнообразие исходных данных обеспечивает надежное *тестирование* предложенного алгоритма.

При описании решений типовых заданий используются возможности, предоставляемые электронным задачником Programming Taskbook (в частности, применяются специальные процедуры для ввода/вывода данных). Благодаря этим возможностям тексты программ с решениями удалось сделать более компактными и наглядными. Процедуры задачника Programming Taskbook описываются в приложении 1 (более подробные сведения о задачнике приводятся в книге [4]).

Приложение 2 содержит 35 контрольных вопросов по теме «Бинарные деревья». Эти вопросы допускают ответ в свободной форме (в отличие от тестов рубежного контроля) и не требуют использования компьютера (в отличие от проектных заданий).

Пособие подготовлено в рамках внутреннего гранта К-08-Т-У-29 Южного федерального университета «Разработка и методическая поддержка образовательной программы (магистратура и бакалавриат) по направлению “Информационные технологии”».

1. Модуль № 1. Анализ бинарного дерева

1.1. Комплексная цель

Ознакомить с основными понятиями, связанными с деревьями и бинарными деревьями. Освоить рекурсивные методы анализа бинарных деревьев.

1.2. Содержание модуля

1.2.1. Деревья: основные понятия

Дерево — это иерархическая структура данных, состоящая из элементов (*вершин*, или *узлов*), которые связаны между собой отношениями типа «*родительская вершина – дочерняя вершина*». Определить дерево с вершинами типа *T* можно следующим образом (см., например, [5], с. 247, [6], с. 33):

- это либо *пустое дерево*, не содержащее ни одной вершины;
- либо некоторая вершина типа *T*, соединенная *ветвями* с конечным числом отдельных деревьев с вершинами типа *T* (эти деревья называются *поддеревьями*).

Чаще всего дерево изображается в виде *графа* (см. рис. 1 и 2), вершинами которого являются вершины дерева, а ребрами — его ветви. Начальная вершина дерева, называемая *корнем*, изображается в верхней части графа, и считается, что она находится на *нулевом уровне*. Вершина *Y*, расположенная ниже вершины *X* и соединенная с ней ветвью, называется *непосредственным потомком* вершины *X*, или ее *дочерней вершиной* (а вершина *X*, соответственно, — *непосредственным предком* вершины *Y*, или ее *родительской вершиной*). Если вершина *X* находится на уровне *K*, то считается, что все ее непосредственные потомки находятся на уровне *K* + 1. На графе, изображающем дерево, все вершины одного уровня располагаются на одной горизонтали. Номер максимального уровня дерева называется *глубиной* (или *высотой*) дерева. Если вершина не имеет потомков, то она называется *терминальной вершиной*, или *листом*. Нетерминальная вершина называется *внутренней*. Число непосредственных потомков внутренней вершины дерева называется *степенью* этой вершины. Максимальная степень всех вершин дерева называется *степенью дерева*. Дерево называется *упорядоченным*, если все непосредственные потомки любой вершины упорядочены.

Пример дерева приведен на рис. 1. Это упорядоченное дерево степени 3 и глубины 2. Его корнем является вершина *A*, листьями — вершины *C*, *E*, *F*, *G*.

Особым видом деревьев являются *бинарные деревья*. Бинарное дерево с вершинами типа *T* можно определить следующим образом:

- это либо пустое дерево, не содержащее ни одной вершины;

- либо некоторая вершина типа *T*, соединенная ветвями с двумя бинарными деревьями с вершинами типа *T* (эти деревья называются *левым и правым поддеревом*).

Пример бинарного дерева приведен на рис. 2; в нем вершина со значением 0 является корнем, все вершины со значением 1 являются левыми дочерними вершинами, а все вершины со значением 2 — правыми.

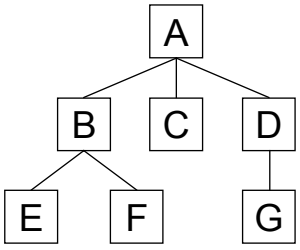


Рис. 1.

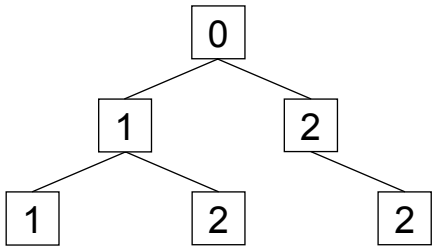


Рис. 2.

Отметим, что бинарные деревья нельзя считать просто упорядоченными деревьями степени 2. Различие между этими типами деревьев проявляется для вершин степени 1, то есть вершин, имеющих единственную дочернюю вершину. В «обычных» упорядоченных деревьях о подобной дочерней вершине нельзя сказать, является ли она левой или правой (такова вершина *G* на рис. 1), тогда как в бинарных деревьях единственная дочерняя вершина всегда будет либо левой, либо правой.

1.2.2. Анализ бинарного дерева: Tree2

В компьютерных программах деревья обычно описываются в виде динамических структур данных, каждый элемент которых является *записью*, содержащей информацию о некоторой вершине дерева и набор ссылок на непосредственных потомков данной вершины. Ссылки реализуются в виде *указателей*, а сами записи размещаются в динамической памяти. В объектно-ориентированных языках программирования со ссылочной объектной моделью вместо записей могут использоваться *объекты*, содержащие ссылки на другие объекты (в настоящем пособии такой способ представления деревьев не рассматривается).

Для хранения информации о бинарном дереве достаточно использовать набор связанных записей, каждая из которых содержит поле Data (в котором хранится *значение* соответствующей вершины дерева) и два поля-указателя Left и Right (которые связывают данную вершину дерева с ее дочерними вершинами).

При описании алгоритмов, связанных с бинарными деревьями, мы будем пользоваться языком Паскаль, предполагая, что программы выполняются в одной из следующих сред: Borland Delphi, Free Pascal Lazarus, Pascal ABC, PascalABC.NET. Кроме того, будем считать, что к Паскаль-программам подключен *электронный задачник Programming Taskbook*, реализованный в виде модуля PT4 и обеспечивающий автоматическую генерацию исходных данных, а также проверку результатов, полученных программой учащегося (см. приложение 1).

В задачнике Programming Taskbook уже определен тип TNode, предназначенный для хранения информации о вершине бинарного дерева, а также указатель PNode на данные этого типа: PNode = ^TNode. Подчеркнем, что типы TNode и PNode не являются стандартными типами языка Паскаль; они определены в задачнике Programming Taskbook и доступны в программе только при подключении к ней модуля PT4. В настоящем пункте на примере задания Tree2 рассматриваются особенности, связанные с использованием этих видов данных.

Программа-заготовка для задания Tree2, созданная с помощью утилиты PT4Load, входящей в состав задачника Programming Taskbook, будет иметь следующий вид:

```
program Tree2;
uses PT4;
begin
  Task('Tree2');
end.
```

После запуска этой программы на экране появится окно, подобное приведенному на рис. 3. В качестве исходных и результирующих данных в этом окне указываются не только числовые данные, но также деревья и указатели.

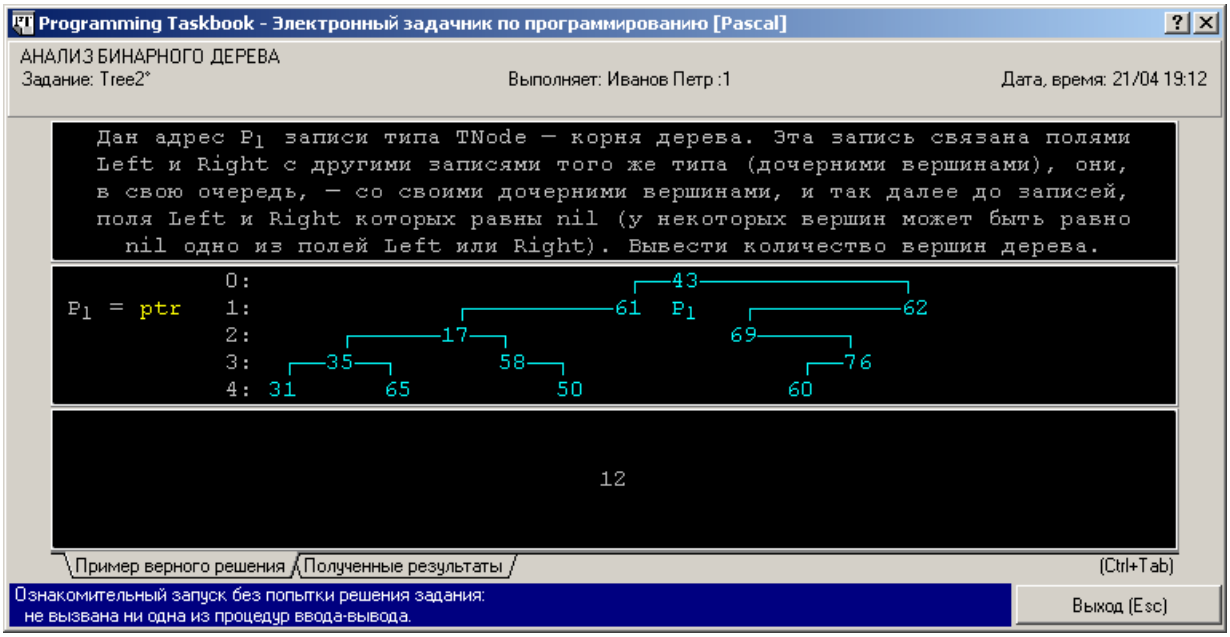


Рис. 3.

Начнем с описания того, как отображается на экране дерево. Для его вывода используются от двух до пяти экранных строк. На каждой строке изображаются вершины дерева, находящиеся на определенном уровне (номер уровня указывается слева от изображения дерева). Для каждой вершины выводится ее значение, то есть значение поля Data соответствующей записи типа TNode. Любая вершина соединяется линиями со своими дочерними вершинами, расположенными на следующем уровне дерева; левая дочерняя вершина изображается

слева от родительской вершины, а правая — справа. Отсутствие у вершины дочерних вершин означает, что ее поля Left и/или Right равны nil.

Вся информация о дереве отображается бирюзовым цветом. Тем самым подчеркивается, что дерево является *внешним элементом данных*, доступ к которому производится стандартными средствами Паскаля, без использования процедур ввода/вывода, определенных в задачнике.

Рассмотрим в качестве примера дерево, приведенное в разделе исходных данных на рис. 3. Корень этого дерева имеет значение 43, левая дочерняя вершина корня равна 61, правая дочерняя вершина равна 62, глубина дерева равна 4. Все листья дерева находятся на уровне 4, они имеют значения 31, 65, 50 и 60. Некоторые из внутренних вершин дерева имеют по две дочерние вершины (это корень и вершины со значениями 17 и 35), некоторые по одной: левой (вершины 61, 62 и 76) или правой (вершины 69 и 58).

Поскольку это дерево указано в разделе исходных данных, следовательно, после инициализации задания оно уже существует и размещается в некоторой области динамической памяти. Как получить доступ к этому дереву? Для доступа к данным, размещенным в динамической памяти, необходимо знать их *адрес*, поэтому в любом задании на обработку деревьев в набор исходных данных входят *указатели*, содержащие адреса этих деревьев (как правило, указывается адрес корня дерева). Имя указателя располагается под изображением той вершины, с которой он связан. В нашем случае с корнем дерева связан указатель P_1, который также содержится в наборе исходных данных. Описание этого указателя имеет вид

```
P1 = ptr
```

Здесь текст «P_1 =» является *комментарием* и выделяется светло-серым цветом, а текст «ptr» означает, что этот элемент исходных данных является *указателем*, и поэтому должен вводиться в программу с помощью специальной процедуры GetP, определенной в задачнике (для ввода исходных данных целого и строкового типа предусмотрены процедуры GetN и GetS — см. приложение 1).

Может возникнуть вопрос: почему вместо текста «ptr» не отображается настоящее значение указателя (то есть некоторый четырехбайтный адрес)? Это связано с тем, что даже выведя значение указателя на экран, мы не сможем определить, с какими данными связан этот указатель, поэтому подобная дополнительная информация будет для нас бесполезна.

Итак, слово «ptr» в разделе исходных или результирующих данных означает, что соответствующий элемент данных является указателем, причем *непустым* (для пустого указателя используется слово «nil»). Определить, с какой вершиной дерева связан непустой указатель, можно по экранной информации об этом дереве. Разумеется, при чтении указателя (процедурой GetP) программа учащегося получит «настоящий» адрес, с помощью которого она сможет обратиться к исходному дереву.

Аналогично, создав (или преобразовав) некоторое дерево, программа учащегося должна передать задачнику некоторый адрес, связанный с этим деревом (используя процедуру PutP). Зная этот адрес, задачник сможет проверить правильность созданного или измененного дерева. Заметим, что для вывода результирующих данных целого и строкового типа предусмотрены процедуры PutN и PutS (см. приложение 1).

Некоторые дополнительные обозначения, используемые при отображении деревьев, будут описаны позже, при рассмотрении других заданий группы Tree.

Вернемся к заданию Tree2. В нем не требуется ни создавать, ни преобразовывать исходное дерево; его необходимо лишь проанализировать, а именно определить количество его вершин. Для решения этого задания, как и для подавляющего большинства других заданий на обработку деревьев, следует воспользоваться вспомогательной *рекурсивной* подпрограммой (функцией или процедурой). Рекурсивная природа алгоритмов, связанных с обработкой деревьев (в частности, бинарных деревьев), объясняется тем, что сами *определения* деревьев общего вида и бинарных деревьев являются рекурсивными (см. п. 1.2.1). Дадим словесное описание функции NodeCount(*P*), подсчитывающей число вершин дерева с корнем, с которым связан указатель *P*: если указатель *P* равен nil, то следует вернуть значение 0; в противном случае следует вернуть значение выражения 1 + NodeCount(*P*^.Left) + NodeCount(*P*^.Right). Заметим, что в данном выражении первое слагаемое соответствует корню дерева, второе — его левому поддереву, а третье — его правому поддереву; при этом не требуется проверять, что указанные поддеревья существуют, так как при их отсутствии соответствующее слагаемое просто будет равно нулю.

Таким образом, решение задания будет иметь следующий вид:

```
program Tree2;
uses PT4;
function NodeCount(P: PNode): integer;
begin
  if P = nil then
    Result := 0
  else
    Result := 1 + NodeCount(P^.Left) + NodeCount(P^.Right);
end;
var
  P1: PNode;
begin
  Task('Tree2');
  GetP(P1);
  PutN(NodeCount(P1));
end.
```

Цепочка рекурсивных вызовов функции NodeCount завершается при достижении терминальной вершины (листа), у которой поля Left и Right равны nil.

Благодаря наличию функции NodeCount раздел операторов программы является очень кратким: в нем считывается адрес P_1 корня исходного дерева, после чего вызывается функция NodeCount(P_1), возвращаемое значение которой сразу выводится процедурой PutN.

1.2.3. Перебор вершин бинарного дерева: Tree12

Во многих ситуациях приходится организовывать перебор вершин бинарного дерева. Подобный перебор естественно реализовать в виде рекурсивной процедуры, которая обеспечивает обработку очередной вершины дерева и (посредством своего рекурсивного вызова) обработку левого и правого поддерева этой вершины. Порядок, в котором обрабатывается очередная вершина и два ее поддерева, и определяет порядок перебора вершин дерева. Хотя три элемента (вершина, ее левое поддерево, ее правое поддерево) можно упорядочить шестью способами, обычно используются только те способы, в которых левое поддерево обрабатывается до правого. Каждый из этих трех способов перебора вершин дерева имеет собственное название. Порядок перебора «левое поддерево – вершина – правое поддерево» называется *инфиксным*, вариант «вершина – левое поддерево – правое поддерево» — *префиксным*, а вариант «левое поддерево – правое поддерево – вершина» — *постфиксным*.

Задание Tree12 посвящено инфиксному порядку перебора вершин дерева: требуется вывести в указанном порядке значения вершин, то есть их поля Data. Как было отмечено выше, для реализации подобного перебора следует использовать рекурсивную процедуру, которая вначале обрабатывает левое поддерево текущей вершины, затем — саму текущую вершину, а затем — ее правое поддерево. В начале данной процедуры (как в и в начале функции, использованной при решении задания Tree2 в предыдущем пункте) следует проверять, не является ли текущая вершина пустой (равной nil), и если является, то немедленно выходить из процедуры.

Решение задания Tree12 принимает следующий вид:

```
program Tree12;
uses PT4;
procedure NodeOutput(P: PNode);
begin
  if P = nil then
    exit;
  NodeOutput(P^.Left);
  PutN(P^.Data);
  NodeOutput(P^.Right);
end;
var
  P1: PNode;
begin
  Task('Tree12');
```

```
  GetP(P1);
  NodeOutput(P1);
end.
```

На рис. 4 приводится вид окна задачника после успешного прохождения пяти тестов, необходимых для того, чтобы задание было зачтено как выполненное.

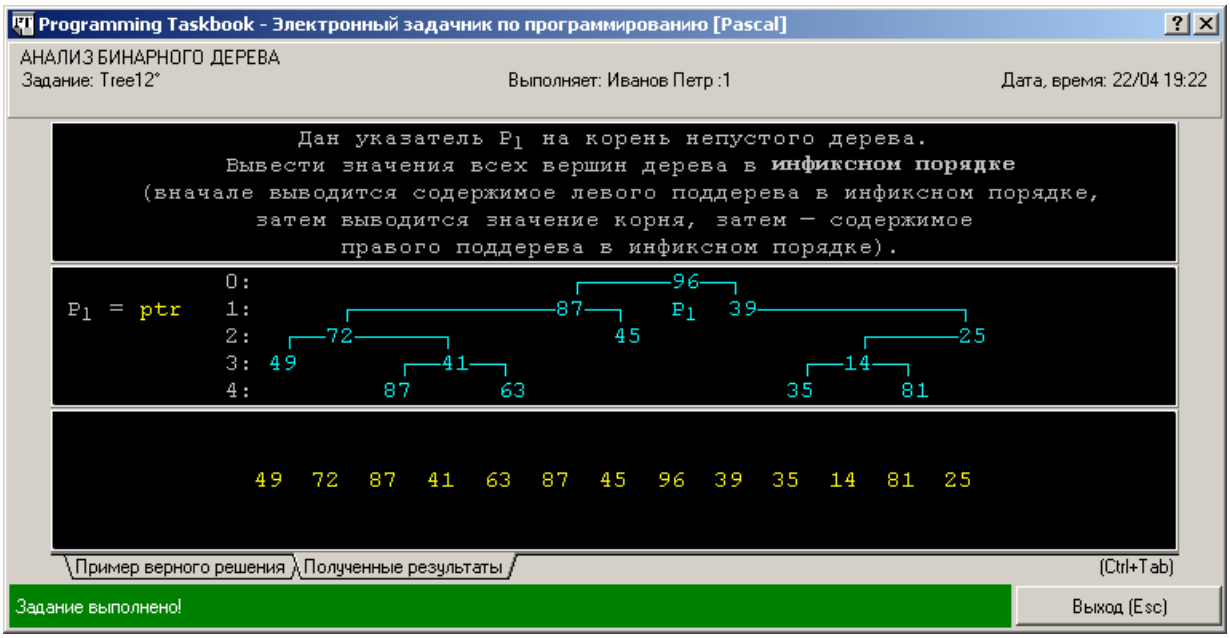


Рис. 4.

Заметим, что для вывода значений вершин в префиксном (задание Tree13) или постфиксном (задание Tree14) порядке в процедуре NodeOutput достаточно изменить порядок следования трех последних операторов. Например, для перебора в префиксном порядке операторы должны располагаться следующим образом:

```
PutN(P^.Data);
NodeOutput(P^.Left);
NodeOutput(P^.Right);
```

Аналогичные процедуры можно использовать и в других заданиях, связанных с перебором вершин бинарного дерева; следует лишь заменить в них оператор, определяющий *действие*, которое надо применить к каждой вершине (в нашем случае таким действием является вывод ее значения: PutN(*P*^.Data)).

1.3. Учебные задания и указания к ним

1.3.1. Формулировки заданий (Tree1–Tree24)

Tree1. Дан адрес P_1 записи типа TNode с полями Data (целого типа), Left и Right (типа PNode — указателя на TNode). Эта запись (*корень дерева*) связана полями Left и Right с записями того же типа (левой и правой *дочерней*

вершиной). Вывести значения полей Data корня, его левой и правой дочерних вершин, а также адреса левой и правой дочерних вершин в указанном порядке.

Tree2. Дан адрес P_1 записи типа TNode — корня дерева. Эта запись связана полями Left и Right с другими записями того же типа (дочерними вершинами), они, в свою очередь, — со своими дочерними вершинами, и так далее до записей, поля Left и Right которых равны nil (у некоторых вершин может быть равно nil одно из полей — Left или Right). Вывести количество вершин дерева.

Tree3. Дан указатель P_1 на корень непустого дерева и число K. Вывести количество вершин дерева, значение которых равно K.

Tree4. Дан указатель P_1 на корень непустого дерева. Вывести сумму значений всех вершин данного дерева.

Tree5. Дан указатель P_1 на корень непустого дерева. Вывести количество вершин дерева, являющихся левыми дочерними вершинами (корень дерева не учитывать).

Tree6. Дан указатель P_1 на корень непустого дерева. *Листом дерева* называется его вершина, не имеющая дочерних вершин. Вывести количество листьев для данного дерева.

Tree7. Дан указатель P_1 на корень непустого дерева. Вывести сумму значений всех листьев данного дерева.

Tree8. Дан указатель P_1 на корень дерева, содержащего по крайней мере две вершины. Вывести количество листьев дерева, являющихся правыми дочерними вершинами.

Tree9. Дан указатель P_1 на корень непустого дерева. Считается, что корень дерева находится на *нулевом уровне*, его дочерние вершины — на *первом уровне* и т. д. Вывести *глубину дерева*, то есть значение его максимального уровня (например, глубина дерева, состоящего только из корня, равна 0).

Tree10. Дан указатель P_1 на корень непустого дерева. Для каждого из уровней данного дерева, начиная с нулевого, вывести количество вершин, находящихся на этом уровне. Считать, что глубина дерева не превосходит 10.

Tree11. Дан указатель P_1 на корень непустого дерева. Для каждого из уровней данного дерева, начиная с нулевого, вывести сумму значений вершин, находящихся на этом уровне. Считать, что глубина дерева не превосходит 10.

Tree12. Дан указатель P_1 на корень непустого дерева. Вывести значения всех вершин дерева в *инфиксном порядке* (вначале выводится содержимое левого поддерева в инфиксном порядке, затем выводится значение корня, затем — содержимое правого поддерева в инфиксном порядке).

Tree13. Дан указатель P_1 на корень непустого дерева. Вывести значения всех вершин дерева в *префиксном порядке* (вначале выводится значение корня,

затем — содержимое левого поддерева в префиксном порядке, затем — содержимое правого поддерева в префиксном порядке).

Tree14. Дан указатель P_1 на корень непустого дерева. Вывести значения всех вершин дерева в *постфиксном порядке* (вначале выводится содержимое левого поддерева в постфиксном порядке, затем — содержимое правого поддерева в постфиксном порядке, затем — значение корня).

Tree15. Дан указатель P_1 на корень непустого дерева и число N (> 0), не превосходящее количество вершин в исходном дереве. Нумеруя вершины в инфиксном порядке (см. задание Tree12, нумерация ведется от 1), вывести значения всех вершин с порядковыми номерами от 1 до N.

Tree16. Дан указатель P_1 на корень непустого дерева и число N (> 0), не превосходящее количество вершин в исходном дереве. Нумеруя вершины в постфиксном порядке (см. задание Tree14, нумерация ведется от 1), вывести значения всех вершин с порядковыми номерами от N до максимального номера.

Tree17. Дан указатель P_1 на корень непустого дерева и два числа N_1, N_2 ($0 < N_1 < N_2$), которые не превосходят количество вершин в исходном дереве. Нумеруя вершины в префиксном порядке (см. задание Tree13, нумерация ведется от 1), вывести значения всех вершин с порядковыми номерами от N_1 до N_2.

Tree18. Дан указатель P_1 на корень непустого дерева и неотрицательное число L. Используя любой из описанных в заданиях Tree12–Tree14 способов обхода дерева, вывести значения всех вершин уровня L, а также их количество N (если дерево не содержит вершин уровня L, то вывести 0).

Tree19. Дан указатель P_1 на корень непустого дерева. Вывести максимальное из значений его вершин и количество вершин, имеющих это максимальное значение.

Tree20. Дан указатель P_1 на корень непустого дерева. Вывести минимальное из значений всех его вершин и количество листьев, имеющих это минимальное значение (данное количество может быть равно 0).

Tree21. Дан указатель P_1 на корень непустого дерева. Вывести минимальное из значений его вершин, являющихся листьями.

Tree22. Дан указатель P_1 на корень дерева, содержащего по крайней мере две вершины. Вывести максимальное из значений его *внутренних вершин* (то есть вершин, не являющихся листьями).

Tree23. Дан указатель P_1 на корень непустого дерева. Вывести указатель P_2 на первую вершину дерева с минимальным значением (вершины перебирать в префиксном порядке).

Tree24. Дан указатель P_1 на корень непустого дерева. Вывести указатель P_2 на последнюю вершину дерева с максимальным нечетным значением (верши-

ны перебирать в инфиксном порядке). Если дерево не содержит вершин с нечетными значениями, то вывести nil.

1.3.2. Указания

Tree1. Вводное задание к группе Tree, для решения которого достаточно вывести значения полей исходных записей в требуемом порядке. Организовывать рекурсивный перебор вершин не требуется.

Tree2–8. Решение Tree2 приводится в п. 1.2.2; прочие задания решаются с использованием аналогичных рекурсивных функций, в которые следует добавить дополнительные условия отбора. Например, в Tree3 можно использовать следующий вариант функции NodeCount:

```
function NodeCount(P: PNode; K: integer): integer;
begin
  Result := 0;
  if P = nil then exit;
  if P^.Data = K then
    Result := 1;
  Result := Result +
    NodeCount(P^.Left, K) + NodeCount(P^.Right, K);
end;
```

Tree9. В данном задании переменную *LMax*, определяющую глубину дерева, удобно описать вне рекурсивной процедуры. Рекурсивная процедура должна содержать вспомогательный параметр — номер уровня *L* для текущей вершины *P* — и корректировать внешнюю переменную *LMax*, если номер уровня *L* превысит прежнее значение *LMax*. Поскольку использовать глобальные переменные в подпрограммах не рекомендуется, следует заключить рекурсивную процедуру в «оболочку» нерекурсивной функции, которую можно назвать TreeDepth (depth — глубина); эта функция принимает в качестве параметра корень *Root* исходного дерева и возвращает глубину дерева:

```
function TreeDepth(Root: PNode): integer;
var
  LMax: integer;
procedure NodeLevel(P: PNode; L: integer);
begin
  if P = nil then exit;
  if L > LMax then
    LMax := L;
  NodeLevel(P^.Left, L + 1);
  NodeLevel(P^.Right, L + 1);
end;
begin
  LMax := 0;
```

```
  NodeLevel(Root, 0);
  Result := LMax;
end;
```

Tree10–11. Ср. с Tree9. В данном случае также удобно использовать нерекурсивную процедуру-оболочку для рекурсивной процедуры. В этой оболочке следует описать массив *Levels* типа array[0..10] of integer, инициализировать его элементы нулевыми значениями и выполнить стартовый запуск рекурсивной процедуры. Массив *Levels* надо заполнять в рекурсивной процедуре таким образом, чтобы в результате элемент *Levels*[*L*] содержал требуемую характеристику всех вершин дерева уровня *L* (а именно количество таких вершин в Tree10 или сумму значений вершин в Tree11). Передавать массив *Levels* во внешнюю программу не требуется; достаточно вывести его элементы непосредственно в процедуре-оболочке (выводятся только элементы с индексами от 0 до *LMax*, где *LMax* — глубина исходного дерева).

Tree12–17. Решение Tree12 и фрагмент решения Tree13 приводятся в п. 1.2.3; прочие задания решаются аналогично. В Tree15–17 следует использовать процедуру-оболочку для рекурсивной процедуры (см. указание к Tree9). В этой оболочке надо описать переменную-счетчик и инициализировать ее значением 0; в рекурсивной процедуре значение переменной-счетчика следует увеличивать на 1 перед обработкой очередной вершины. Данная переменная позволит отобрать те вершины дерева, которые требуется вывести на экран.

Tree18. Один из алгоритмов перебора элементов дерева приведен в п. 1.2.3 (см. решение Tree12). По поводу определения уровня для каждой вершины см. указание к Tree9.

Tree19–24. В этих заданиях требуется организовать перебор всех вершин дерева, причем порядок перебора важен только для заданий Tree23–24 (см. п. 1.2.3). Для хранения требуемого результата следует использовать переменную (целого типа для Tree19–22, типа PNode для Tree23–24), которая будет *внешней* по отношению к рекурсивной процедуре, обеспечивающей перебор вершин дерева. Удобно описать эту внешнюю переменную в функции-оболочке для рекурсивной процедуры (см. указание к Tree9).

1.4. Проектное задание

Выполните учебные задания группы Tree, указанные в вашем варианте проектного задания. Если вы не получили вариант проектного задания, то выполните задания из первого варианта.

ВАРИАНТ 1 Анализ дерева: 7, 16, 20	**ВАРИАНТ 2** Анализ дерева: 4, 17, 19

ВАРИАНТ 3 Анализ дерева: 5, 14, 21	**ВАРИАНТ 4** Анализ дерева: 3, 16, 22
ВАРИАНТ 5 Анализ дерева: 4, 17, 19	**ВАРИАНТ 6** Анализ дерева: 7, 14, 22
ВАРИАНТ 7 Анализ дерева: 6, 17, 23	**ВАРИАНТ 8** Анализ дерева: 5, 15, 24
ВАРИАНТ 9 Анализ дерева: 3, 14, 24	**ВАРИАНТ 10** Анализ дерева: 8, 15, 20
ВАРИАНТ 11 Анализ дерева: 8, 16, 23	**ВАРИАНТ 12** Анализ дерева: 6, 15, 21
ВАРИАНТ 13 Анализ дерева: 7, 14, 19	**ВАРИАНТ 14** Анализ дерева: 7, 16, 23
ВАРИАНТ 15 Анализ дерева: 3, 15, 20	**ВАРИАНТ 16** Анализ дерева: 8, 15, 21
ВАРИАНТ 17 Анализ дерева: 6, 16, 21	**ВАРИАНТ 18** Анализ дерева: 4, 16, 22
ВАРИАНТ 19 Анализ дерева: 8, 14, 22	**ВАРИАНТ 20** Анализ дерева: 4, 17, 23
ВАРИАНТ 21 Анализ дерева: 3, 17, 24	**ВАРИАНТ 22** Анализ дерева: 5, 17, 24
ВАРИАНТ 23 Анализ дерева: 5, 14, 19	**ВАРИАНТ 24** Анализ дерева: 6, 15, 20

1.5. Тест рубежного контроля

1. Укажите, сколько вершин — непосредственных потомков может иметь вершина произвольного дерева.			
(1)	Две	(2)	Не более двух
(3)	Не менее одной	(4)	Любое количество
2. Выберите правильное продолжение определения: «Внутренней вершиной называется вершина дерева, которая…			
(1)	не имеет потомков»	(2)	имеет ровно одного потомка»
(3)	имеет не менее одного потомка»	(4)	имеет ровно два непосредственных потомка»
3. Укажите, сколько вершин — непосредственных потомков может иметь вершина бинарного дерева.			
(1)	Две	(2)	Не более двух
(3)	Не менее одной	(4)	Любое количество
4. Закончите фразу: «Если вершина бинарного дерева имеет одного непосредственного потомка, то этот потомок…			
(1)	является левой дочерней вершиной»	(2)	является правой дочерней вершиной»
(3)	может быть как левой, так и правой дочерней вершиной»	(4)	не является ни левой, ни правой дочерней вершиной»
5. Перечислите поля записи TNode, которые используются при решении задач на анализ бинарного дерева.			
(1)	Data, Next, Prev	(2)	Left, Right
(3)	Data, Left, Right	(4)	Data, Left, Right, Next, Prev
6. Укажите вариант перебора вершин бинарного дерева, называемый инфиксным.			
(1)	Вершина – левое поддерево – правое поддерево	(2)	Левое поддерево – вершина – правое поддерево
(3)	Правое поддерево – вершина – левое поддерево	(4)	Левое поддерево – правое поддерево – вершина

2. Модуль № 2. Формирование и преобразование бинарного дерева

2.1. Комплексная цель

Ознакомить с приемами создания и преобразования бинарных деревьев.

2.2. Содержание модуля

2.2.1. Формирование бинарного дерева: Tree32

В качестве примера задания на формирование бинарного дерева рассмотрим задание Tree32, в котором требуется сформировать идеально сбалансированное дерево с N вершинами, значения которых даны. Определение *идеально сбалансированного дерева* дается в самой формулировке задания: это дерево, для каждой вершины которого количество вершин в ее левом и правом поддереве отличается не более чем на 1. Для того чтобы увидеть пример идеально сбалансированного дерева, достаточно запустить программу-заготовку, созданную для задания Tree32; при этом требуемое дерево будет изображено на вкладке «Пример верного решения» (см. рис. 5).

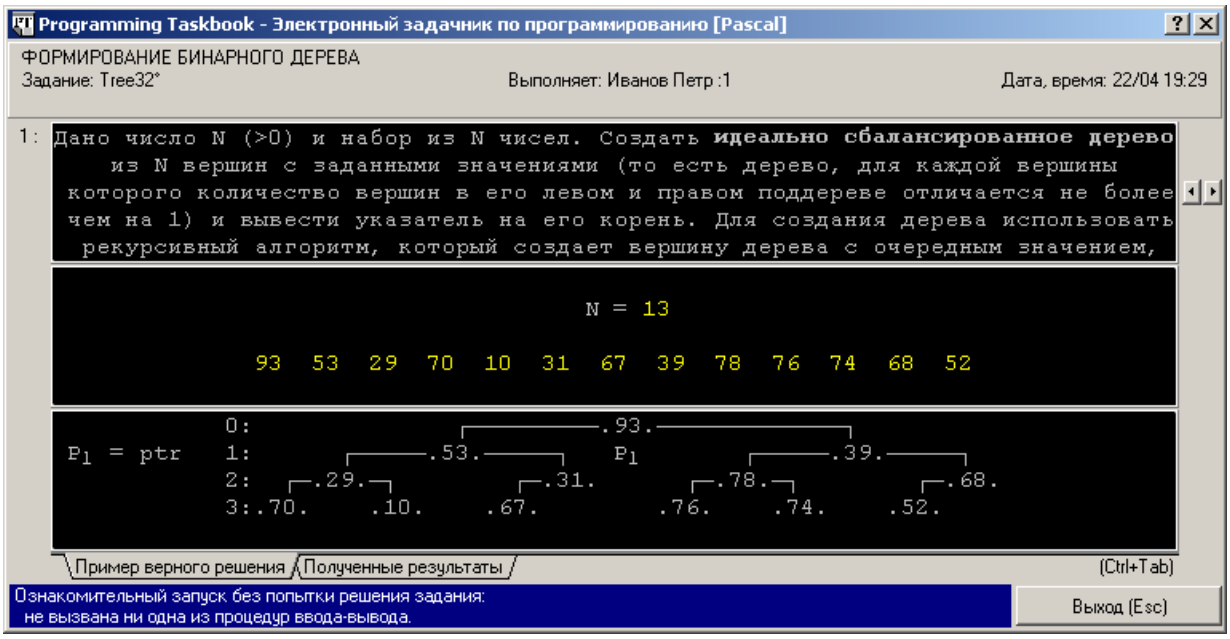


Рис. 5.

Изображение дерева, которое требуется создать для успешного выполнения задания, содержит новое обозначение, а именно точки, обрамляющие вершины дерева. Точки обозначают те вершины дерева, память для которых *должна быть выделена программой учащегося* (в отличие от тех вершин, кото-

рые размещаются в памяти самим задачником). В задании Tree32 программа учащегося должна выделить память для всех вершин результирующего дерева, поэтому в примере верного решения точки указываются около каждой вершины.

Ясно, что идеально сбалансированное дерево с указанным числом вершин можно сформировать неединственным образом. В примере, приведенном на рис. 5, левые потомки вершин 53 и 39 имеют по две дочерние вершины, тогда как правые — по одной (левой) дочерней вершине. В то же время, если бы правые потомки вершин 53 и 39 имели только правые дочерние вершины (или левые потомки имели по одной дочерней вершине, а правые — по две), то дерево все равно осталось бы идеально сбалансированным. Кроме того, значения созданным вершинам можно присваивать, перебирая их в различном порядке (см. п. 1.2.3). Для того чтобы избежать неоднозначности при построении идеально сбалансированного дерева, в формулировке задания приводится алгоритм его построения. Если считать, что функция Create(*N*) создает идеально сбалансированное дерево из *N* вершин и возвращает указатель на его корень, то, согласно алгоритму, приведенному в задании Tree32, эта функция должна действовать следующим образом: создать вершину дерева, присвоить ее полю Data очередное значение из набора исходных чисел, а затем сформировать ее левое и правое поддерево, выполнив свой рекурсивный вызов с параметрами $N \operatorname{div} 2$ и $N - 1 - N \operatorname{div} 2$ (при этом суммарное количество вершин будет равно *N*, причем количество вершин в левом и правом поддереве действительно будет отличаться не более чем на 1). Разумеется, в начале функции CreateNode следует обработать ситуацию, когда параметр *N* равен 0 (в этом случае функция не выполняет никаких действий и возвращает значение nil).

Итак, решение задания Tree32 будет выглядеть следующим образом:

```
program Tree32;
uses PT4;
function Create(N: integer): PNode;
begin
  if N = 0 then
  begin
    Result := nil;
    exit;
  end;
  New(Result);
  GetN(Result^.Data);
  Result^.Left := Create(N div 2);
  Result^.Right := Create(N - 1 - N div 2);
end;
var
  N: integer;
begin
```

```
  Task('Tree32');
  GetN(N);
  PutP(Create(N));
end.
```

Обратите внимание на то, что в функции CreateTree мы обошлись без вспомогательной локальной переменной типа PNode, так как при выделении памяти и заполнении полей записи можно использовать переменную *Result*.

Для того чтобы получить другие варианты сбалансированных бинарных деревьев, достаточно немного изменить процедуру CreateTree. Можно, например, поменять значения параметров при рекурсивных вызовах функции:

```
Result^.Left := Create(N - 1 - N div 2);
Result^.Right := Create(N div 2);
```

Полученное дерево также будет идеально сбалансированным, однако, из-за того, что для его построения был использован алгоритм, отличный от описанного в задании, решение будет считаться ошибочным (см. рис. 6). Обратите внимание на то, что для полученного дерева, в отличие от дерева, изображенного на рис. 5, правое поддерево может иметь больше вершин, чем левое. Кроме того, если вершина имеет только одного потомка, этим потомком является не левая, а правая дочерняя вершина.

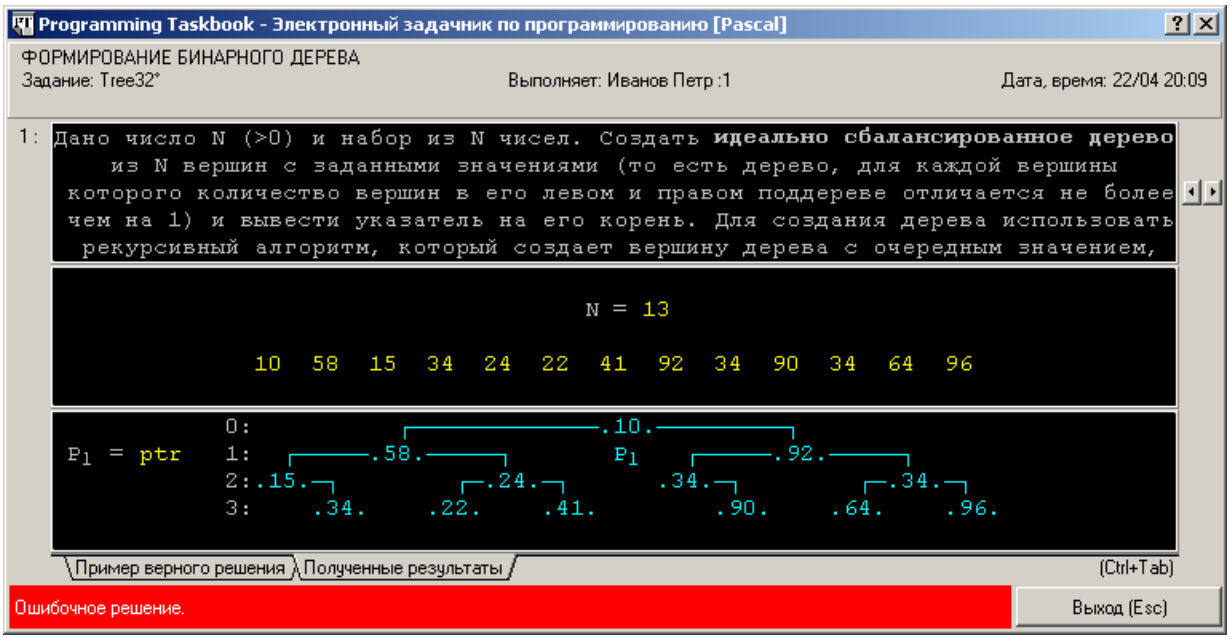


Рис. 6.

2.2.2. Преобразование бинарного дерева: Tree40

В процессе преобразования дерева может потребоваться добавить к нему новые вершины или удалить имеющиеся. Действия по добавлению новых вершин ничем по существу не отличаются от действий, связанных с формированием дерева (см. п. 2.2.1), поэтому в настоящем пункте мы рассмотрим задание,

связанное с удалением части вершин из исходного дерева, а именно задание Tree40, в котором требуется удалить все вершины, кроме корня.

Особенность заданий на удаление вершин заключается в том, что удаляемую вершину необходимо не только «отсоединить» от исходного дерева, но и полностью «уничтожить», то есть освободить память, занимаемую этой вершиной. Для того чтобы напомнить учащемуся о необходимости уничтожения некоторых вершин исходного дерева, значения этих вершин выделяются на экране бирюзовым цветом *меньшей яркости*.

Для отсоединения от корня его поддеревьев достаточно положить равными nil значения его полей Left и Right:

```
program Tree40;
uses PT4;
var
  P1: PNode;
begin
  Task('Tree40');
  GetP(P1);
  P1^.Left := nil;
  P1^.Right := nil;
end.
```

В программе не используются процедуры вывода. Это связано с тем, что в задании требуется изменить исходное дерево, указатель на корень которого известен. Поэтому для проверки правильности решения задачнику достаточно проанализировать структуру измененного дерева с тем же самым корнем. Таким образом, приведенная выше программа не содержит ошибок, связанных с вводом/выводом. Более того, измененное дерево будет совпадать с требуемым (иными словами, текст на вкладках «Полученные результаты» и «Пример верного решения» в окне задачника будет одинаковым). Однако приведенное решение является ошибочным, так как в нем не удаляются из памяти вершины, отсоединенные от исходного дерева. Поэтому при запуске программы на информационной панели окна задачника появится сообщение об ошибке «Не освобождена динамическая память», а в разделе исходных данных будут выделены красным цветом те вершины, которые требовалось удалить.

Для получения правильного решения необходимо для всех удаляемых вершин вызвать процедуру Dispose(*P*), освобождающую память, с которой связан указатель *P*. Перебор вершин, для которых требуется вызвать процедуру Dispose, следует, как обычно, реализовать с помощью рекурсивной процедуры. Эта процедура должна вызывать себя для разрушения левого и правого поддерева текущей вершины, после чего разрушать саму эту вершину. Итак, нам осталось дополнить предыдущий вариант программы следующим образом:

```
program Tree40;
uses PT4;
```

```
procedure Delete(P: PNode);
begin
  if P = nil then exit
  else
  begin
    Delete(P^.Left);
    Delete(P^.Right);
    Dispose(P);
  end;
end;
var
  P1: PNode;
begin
  Task('Tree40');
  GetP(P1);
  Delete(P1^.Left);
  Delete(P1^.Right);
  P1^.Left := nil;
  P1^.Right := nil;
end.
```

2.3. Учебные задания и указания к ним

2.3.1. Формулировки заданий (Tree25–Tree47)

Tree25. Дано число N (> 0) и набор из N чисел. Создать дерево из N вершин, в котором каждая вершина (кроме корня) является правой дочерней вершиной. Каждой создаваемой вершине присваивать очередное значение из исходного набора. Вывести указатель на корень созданного дерева.

Tree26. Дано число N (> 0) и набор из N чисел. Создать дерево из N вершин, в котором каждая внутренняя вершина имеет только одну дочернюю вершину, причем правые и левые дочерние вершины чередуются (корень имеет левую дочернюю вершину). Каждой создаваемой вершине присваивать очередное значение из исходного набора. Вывести указатель на корень созданного дерева.

Tree27. Дано число N (> 0) и набор из N чисел. Создать дерево из N вершин, в котором каждая внутренняя вершина имеет только одну дочернюю вершину, причем если значение вершины является нечетным, то она имеет левую дочернюю вершину, а если четным, то правую. Каждой создаваемой вершине присваивать очередное значение из исходного набора. Вывести указатель на корень созданного дерева.

Tree28. Дано четное число N (> 0) и набор из N чисел. 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. Вывести указатель на корень созданного дерева.

Tree29. Дано четное число N (> 0) и набор из N чисел. Создать дерево из N вершин со следующей структурой: если вершина дерева является внутренней, то в случае, если она имеет нечетное значение, ее левая дочерняя вершина должна быть листом, а в случае, если она имеет четное значение, листом должна быть ее правая вершина. Для каждой внутренней вершины вначале создавать дочернюю вершину-лист, а затем дочернюю внутреннюю вершину (если данная вершина существует); каждой создаваемой вершине присваивать очередное значение из исходного набора. Вывести указатель на корень созданного дерева.

Tree30. Дано число N (> 0). Создать дерево, корень которого имеет значение N, а вершины обладают следующими свойствами: вершина с четным значением K имеет левую дочернюю вершину со значением $K/2$ и не имеет правой дочерней вершины; вершина со значением 1 является листом; вершина с любым другим нечетным значением K имеет две дочерние вершины: левую со значением $K/2$ и правую со значением $K - K/2$ (символ «/» обозначает операцию деления нацело). Вывести указатель на корень созданного дерева.

Tree31. Даны положительные числа L, N ($N > L$) и набор из N чисел. Создать дерево глубины L, содержащее вершины со значениями из исходного набора. Вершины добавлять к дереву в префиксном порядке, используя алгоритм, который для каждой вершины уровня, не превышающего L, вначале создает саму вершину с очередным значением из исходного набора, затем ее левое поддерево соответствующей глубины, а затем ее правое поддерево. Если для заполнения дерева глубины L требуется менее N вершин, то оставшиеся числа из исходного набора не использовать. Вывести указатель на корень созданного дерева.

Tree32. Дано число N (> 0) и набор из N чисел. Создать *идеально сбалансированное дерево* из N вершин с заданными значениями (то есть дерево, для каждой вершины которого количество вершин в его левом и правом поддереве отличается не более чем на 1) и вывести указатель на его корень. Для создания дерева использовать рекурсивный алгоритм, который создает вершину дерева с очередным значением, после чего вызывается для создания левого поддерева с $N/2$ вершинами и правого поддерева с $N - 1 - N/2$ вершинами (символ «/» обозначает операцию деления нацело).

Tree33. Дано число N (> 0). Создать идеально сбалансированное дерево из N вершин и вывести указатель на его корень. Значение каждой вершины положить равным уровню этой вершины (например, корень дерева должен

иметь значение 0, его дочерние вершины — значение 1 и т. д.). При формировании дерева использовать алгоритм, описанный в задании Tree32.

Tree34. Дан указатель P_1 на корень непустого дерева. Создать копию данного дерева и вывести указатель P_2 на корень созданной копии.

Tree35. Дан указатель P_1 на корень непустого дерева. Удвоить значение каждой вершины дерева.

Tree36. Дан указатель P_1 на корень непустого дерева. Для каждой вершины дерева с четным значением уменьшить ее значение в два раза.

Tree37. Дан указатель P_1 на корень непустого дерева. Добавить 1 к значению каждого листа дерева и вычесть 1 из значения каждой внутренней вершины.

Tree38. Дан указатель P_1 на корень непустого дерева. Для каждой вершины дерева, имеющей две дочерние вершины, поменять местами значения дочерних вершин (то есть значения их полей Data).

Tree39. Дан указатель P_1 на корень непустого дерева. Для всех внутренних вершин дерева поменять местами их левые и правые дочерние вершины (то есть значения полей Left и Right).

Tree40. Дан указатель P_1 на корень непустого дерева. Удалить из дерева все вершины, кроме корня, и освободить память, которую занимали удаленные вершины (полям Left и Right корня следует присвоить значение nil).

Tree41. Дан указатель P_1 на корень дерева, содержащего по крайней мере две вершины. Удалить каждую вершину дерева, являющуюся листом; при этом соответствующее поле родительской вершины (Left или Right) следует положить равным nil. При удалении вершин освобождать память, которую они занимали.

Tree42. Дан указатель P_1 на корень непустого дерева. Удалить вершины дерева, имеющие значения, меньшие значения корня, вместе со всеми их дочерними вершинами. При удалении вершин освобождать память, которую они занимали.

Tree43. Дан указатель P_1 на корень непустого дерева. Для вершин дерева, имеющих две дочерние вершины, удалить одну из дочерних вершин: правую, если родительская вершина имеет четное значение, и левую в противном случае (вершины дерева перебирать в префиксном порядке, при удалении вершины удалять и всех ее потомков). Для удаленных вершин освобождать память, которую они занимали.

Tree44. Дан указатель P_1 на корень непустого дерева. Ко всем вершинам дерева, которые являются листьями, добавить по две дочерние вершины-листа: левую со значением 10 и правую со значением 11.

Tree45. Дан указатель P_1 на корень непустого дерева. Ко всем вершинам дерева, которые являются листьями, добавить по одной дочерней вершине-

листу; при этом к исходной вершине с нечетным значением добавляется левая дочерняя вершина, а к вершине с четным значением — правая. Значение каждой добавленной вершины положить равным значению ее родительской вершины.

Tree46. Дан указатель P_1 на корень непустого дерева. Ко всем вершинам дерева, которые содержат ровно по одной дочерней вершине, добавить еще одну дочернюю вершину-лист. Значение каждой добавленной вершины положить равным удвоенному значению ее родительской вершины.

Tree47. Дан указатель P_1 на корень непустого дерева. Не изменяя глубины L исходного дерева, дополнить его до *полного дерева*, то есть дерева, все листья которого находятся на уровне L. Значения всех добавленных вершин положить равными –1.

2.3.2. Указания

Tree25–31. В заданиях Tree25–29 требуется создать деревья, имеющие, по существу, линейную структуру, поэтому при формировании дерева достаточно последовательно создавать вершины в *цикле*, не прибегая к рекурсивным алгоритмам. В Tree30–31 следует использовать рекурсивную функцию, подобную функции CreateTree, приведенной в п. 2.2.1 при обсуждении задания Tree32 (функция возвращает указатель на созданную вершину). В Tree30 данная функция должна иметь параметр N, определяющий значение создаваемой вершины. В Tree31 функция должна иметь параметр L, определяющий уровень создаваемой вершины (для определения значения вершины достаточно прочесть очередное число из набора исходных данных); кроме того, необходимо вести подсчет уже созданных вершин, чтобы прекратить формирование дерева, когда будут прочитаны все исходные данные (для этого следует использовать внешнюю к рекурсивной функции переменную-счетчик — см. указание к Tree12–17).

Tree32–33. Решение Tree32 приводится в п. 2.2.1; задание Tree33 решается аналогично, с применением того же алгоритма формирования идеально сбалансированного дерева.

Tree34. Опишите и используйте рекурсивную функцию CopyTree, подобную функции CreateTree, приведенной в п. 2.2.1. Функция должна иметь параметр P — указатель на вершину исходного дерева — и возвращать указатель на соответствующую вершину в созданном дереве-копии. Если параметр P равен nil, то функция возвращает nil, иначе функция создает новую вершину, заносит в нее значение P^.Data и полагает ее поля Left и Right равными Copy(P^.Left) и Copy(P^.Right) соответственно.

Tree35–39. Задания на преобразование исходного дерева, при котором не требуется ни удалять, ни добавлять вершины. Для выполнения этих заданий достаточно реализовать рекурсивную процедуру с параметром P — указа-

телем на текущую вершину дерева. Данная процедура обрабатывает вершину дерева *P*, после чего (в случае, если *P* <> nil) выполняет два своих вызова с параметрами *P*^.Left и *P*^.Right (ср. с фрагментом решения Tree13, приведенным в конце п. 1.2.3).

Tree40–43. Задания на удаление части вершин из исходного дерева. Решение Tree40 приводится в п. 2.2.2; прочие задания решаются аналогично.

Tree44–47. Задания на добавление к существующему дереву новых вершин. В Tree44–46 достаточно организовать перебор существующих вершин, в ходе которого необходимо выявлять вершины с требуемыми свойствами (например, вершины-листья в Tree44–45) и создавать для них новые дочерние вершины. Важно организовать перебор таким образом, чтобы обрабатывались только «старые» вершины исходного дерева. Решение Tree47 состоит из двух этапов: на первом этапе следует определить глубину исходного дерева *LMax* (см. Tree9), на втором этапе — организовать перебор вершин дерева и добавление дочерних вершин к тем вершинам, уровень которых имеет значение, меньшее *LMax* (значение уровня вершины следует передавать в качестве дополнительного параметра *L* рекурсивной функции, выполняющей перебор существующих и создание новых вершин, — ср. с процедурой NodeLevel, приведенной в указании к Tree9).

2.4. Проектное задание

Выполните учебные задания группы Tree, указанные в вашем варианте проектного задания. Если вы не получили вариант проектного задания, то выполните задания из первого варианта.

ВАРИАНТ 1 (1) Формирование дерева: 30, 34 (2) Преобразование дерева: 35, 41, 45	**ВАРИАНТ 2** (1) Формирование дерева: 29, 33 (2) Преобразование дерева: 39, 42, 44
ВАРИАНТ 3 (1) Формирование дерева: 30, 31 (2) Преобразование дерева: 38, 42, 46	**ВАРИАНТ 4** (1) Формирование дерева: 27, 34 (2) Преобразование дерева: 38, 42, 47
ВАРИАНТ 5 (1) Формирование дерева: 26, 31 (2) Преобразование дерева: 39, 41, 44	**ВАРИАНТ 6** (1) Формирование дерева: 27, 34 (2) Преобразование дерева: 36, 41, 46
ВАРИАНТ 7 (1) Формирование дерева: 26, 34 (2) Преобразование дерева: 37, 42, 47	**ВАРИАНТ 8** (1) Формирование дерева: 25, 31 (2) Преобразование дерева: 39, 41, 45

ВАРИАНТ 9 (1) Формирование дерева: 28, 33 (2) Преобразование дерева: 37, 43, 45	**ВАРИАНТ 10** (1) Формирование дерева: 28, 33 (2) Преобразование дерева: 39, 43, 47
ВАРИАНТ 11 (1) Формирование дерева: 29, 33 (2) Преобразование дерева: 36, 43, 46	**ВАРИАНТ 12** (1) Формирование дерева: 25, 31 (2) Преобразование дерева: 35, 43, 44
ВАРИАНТ 13 (1) Формирование дерева: 28, 34 (2) Преобразование дерева: 39, 42, 44	**ВАРИАНТ 14** (1) Формирование дерева: 26, 31 (2) Преобразование дерева: 39, 41, 47
ВАРИАНТ 15 (1) Формирование дерева: 26, 31 (2) Преобразование дерева: 37, 43, 46	**ВАРИАНТ 16** (1) Формирование дерева: 29, 33 (2) Преобразование дерева: 37, 41, 47
ВАРИАНТ 17 (1) Формирование дерева: 27, 33 (2) Преобразование дерева: 39, 43, 44	**ВАРИАНТ 18** (1) Формирование дерева: 25, 34 (2) Преобразование дерева: 36, 41, 45
ВАРИАНТ 19 (1) Формирование дерева: 30, 33 (2) Преобразование дерева: 35, 42, 46	**ВАРИАНТ 20** (1) Формирование дерева: 28, 31 (2) Преобразование дерева: 35, 43, 45
ВАРИАНТ 21 (1) Формирование дерева: 29, 33 (2) Преобразование дерева: 38, 42, 44	**ВАРИАНТ 22** (1) Формирование дерева: 30, 34 (2) Преобразование дерева: 39, 42, 45
ВАРИАНТ 23 (1) Формирование дерева: 25, 31 (2) Преобразование дерева: 36, 41, 46	**ВАРИАНТ 24** (1) Формирование дерева: 27, 34 (2) Преобразование дерева: 38, 43, 47

2.5. Тест рубежного контроля

1. Укажите правильное продолжение определения: «Идеально сбалансированным бинарным деревом называется дерево, в котором…

(1)	каждая нетерминальная вершина имеет ровно два непосредственных потомка»	(2)	для каждой вершины количество вершин в ее левом и правом поддереве отличается не более чем на 1»
(3)	для каждой вершины количество вершин в ее левом и правом поддереве совпадает»	(4)	каждая вершина имеет не более одного непосредственного потомка»
2. Укажите, какие действия требуются для того, чтобы удалить из бинарного дерева все вершины, кроме корня Root.			
(1)	Положить Root^.Left и Root^.Right равными nil	(2)	Вызвать процедуру Dispose для Root^.Left и Root^.Right
(3)	Положить Root^.Left и Root^.Right равными nil, после чего вызвать процедуру Dispose для Root^.Left и Root^.Right	(4)	Вызвать процедуру Dispose для Root^.Left и Root^.Right, после чего положить Root^.Left и Root^.Right равными nil
3. Укажите верное утверждение.			
(1)	При создании непустого дерева всегда используется рекурсия и может использоваться процедура New	(2)	При создании непустого дерева всегда используется рекурсия и всегда используется процедура New
(3)	При создании непустого дерева может использоваться рекурсия и всегда используется процедура New	(4)	При создании непустого дерева может использоваться рекурсия и может использоваться процедура New
4. Укажите правильное продолжение определения: «Полным бинарным деревом называется дерево глубины L, в котором…			
(1)	число вершин равно 2^L»	(2)	число листьев равно L»
(3)	число вершин равно $2L+1$»	(4)	число листьев равно 2^L»
5. Укажите, какие действия требуются для перемены местами левого и правого потомка вершины P.			
(1)	Поменять значения полей P^.Left и P^.Right	(2)	Поменять значения полей P^.Left^.Data и P^.Right^.Data
(3)	Поменять значения полей P^.Left^.Data и P^.Right^.Data, после чего поменять значения полей P^.Left и P^.Right	(4)	Поменять значения полей P^.Left и P^.Right, после чего поменять значения полей P^.Left^.Data и P^.Right^.Data

6. Укажите правильный вариант алгоритма создания идеально сбалансированного дерева с N вершинами.			
(1)	Алгоритм создает корень дерева, после чего рекурсивно вызывается для создания левого и правого поддерева с N div 2 вершинами	(2)	Алгоритм создает корень дерева, после чего рекурсивно вызывается для создания левого и правого поддерева с $N - 1 - N$ div 2 вершинами
(3)	Алгоритм создает корень дерева, после чего рекурсивно вызывается для создания левого поддерева с N div 2 вершинами и правого поддерева с $N - 1 - N$ div 2 вершинами	(4)	Алгоритм создает корень дерева, после чего рекурсивно вызывается для создания левого поддерева с N div 2 вершинами и правого поддерева с $N - N$ div 2 вершинами

3. Модуль № 3. Бинарные деревья с обратной связью и бинарные деревья поиска

3.1. Комплексная цель

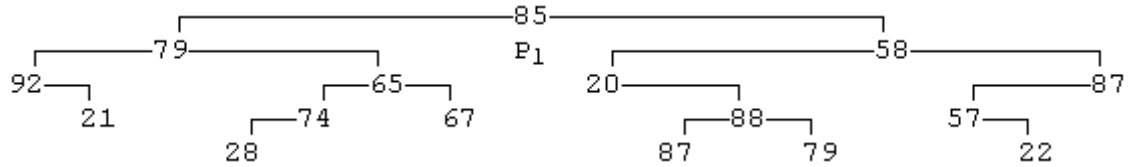
Ознакомить с методами обработки деревьев с обратной связью. Освоить алгоритмы, в которых применяются бинарные деревья поиска, в частности, алгоритм сортировки деревом.

3.2. Содержание модуля

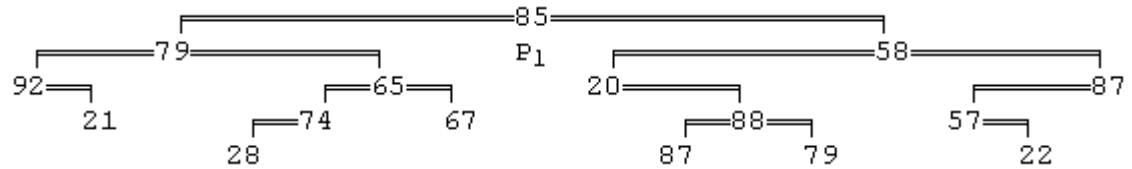
3.2.1. Бинарные деревья с обратной связью: Tree49

Рассмотренные ранее реализации бинарных деревьев позволяют легко переходить от родительских вершин к их дочерним вершинам, но не допускают обратного перехода. В то же время, для некоторых задач, связанных с обработкой деревьев, возможность обратного перехода от потомков к их предку позволяет получить более простое решение. Ясно, что для обеспечения возможности обратного перехода каждую вершину дерева надо снабдить еще одним полем связи, в котором должна храниться ссылка на ее родительскую вершину. Это поле связи естественно назвать Parent. Поскольку корень дерева предка не имеет, его поле Parent должно быть равно nil.

Деревья, вершины которых содержат информацию о своих родителях, будем называть *деревьями с обратной связью*. Особенности работы с подобными деревьями рассмотрим на примере задания Tree49, в котором требуется преобразовать «обычное» бинарное дерево в дерево с обратной связью. Запустив программу-заготовку, созданную для этого задания, мы увидим в области исходных данных изображение бинарного дерева, подобное рассмотренным в предыдущих пунктах, например:



В области результатов будет изображено дерево с обратной связью, вершины которого связаны не одинарными, а двойными линиями:



Для преобразования исходного дерева в дерево с обратной связью необходимо задать правильные значения для полей Parent всех вершин дерева, перебирая эти вершины с помощью подходящей рекурсивной процедуры. В эту процедуру удобно передавать в качестве параметров не только указатель *P* на текущую вершину, но и указатель *Par* на предка этой вершины:

```
program Tree49;
uses PT4;
procedure SetParent(P, Par: PNode);
begin
  if P = nil then exit;
  P^.Parent := Par;
  SetParent(P^.Left, P);
  SetParent(P^.Right, P);
end;
var
  P1: PNode;
begin
  Task('Tree49');
  GetP(P1);
  SetParent(P1, nil);
end.
```

В этой программе, как и в программе, приведенной в п. 2.2.2, не используются процедуры вывода. Обратите внимание на то, что при стартовом вызове рекурсивной процедуры SetParent в качестве второго параметра указывается nil.

Заметим, что все остальные задания на обработку деревьев с обратной связью не требуют действий, подобных приведенным в решении задания Tree49, так как исходные деревья в этих заданиях уже содержат правильные значения полей Parent для всех вершин.

Особое обозначение для двойной связи может оказаться полезным при анализе ошибочного решения. Так, если в изображении дерева с обратной связью имеется вершина, соединенная со своей родительской вершиной не двойной, а одинарной линией, значит у этой вершины поле Parent содержит ошибочное значение (например, равно nil).

Специальное обозначение предусмотрено также для ситуации, когда корень дерева с обратной связью имеет значение, отличное от nil. Эту ситуацию можно промоделировать с помощью приведенной выше программы, если изменить стартовый вызов процедуры SetParent следующим образом:

```
SetParent(P1, P1);
```

В результате подобного изменения поле Parent корня дерева будет указывать на этот же самый корень, что является ошибочным. При запуске измененной программы окно задачника примет вид, приведенный на рис. 7.

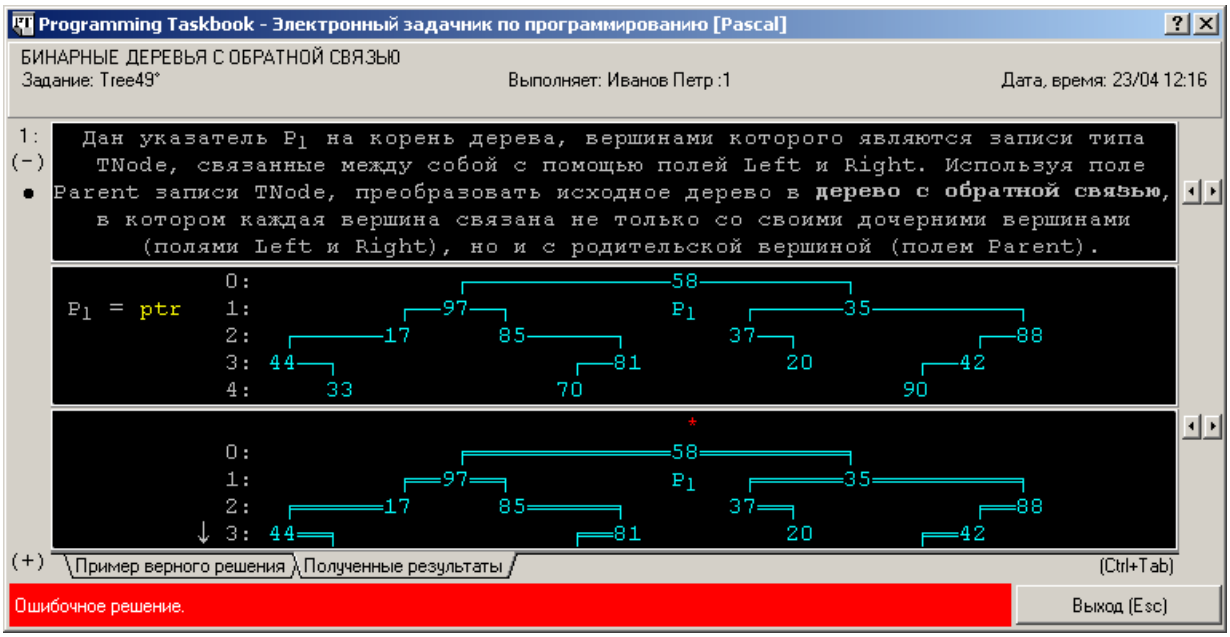


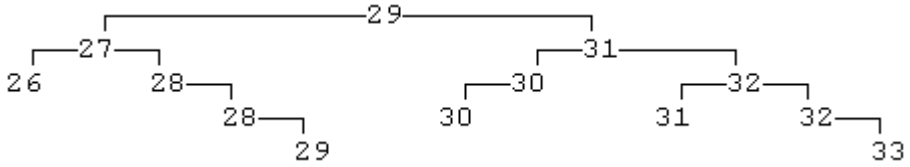
Рис. 7.

Признаком ошибочного значения поля Parent для корня полученного дерева является красная звездочка, отображаемая выше этого корня.

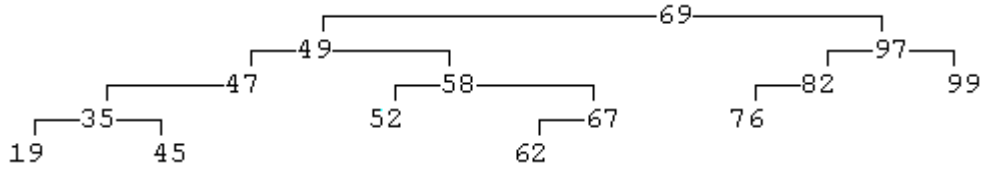
В приведенном окне содержится еще один элемент, который ранее не встречался: это стрелка ↓, расположенная рядом с номером уровня 3 в разделе результатов. Наличие подобной стрелки означает, что в дереве имеются уровни, которые в данный момент не отображаются на экране. В нашем случае это уровень 4, который сместился за нижнюю границу раздела результатов из-за того, что первая строка раздела была отведена для вывода звездочки. Для отображения на экране этого уровня достаточно «прокрутить» изображение дерева (действия для прокрутки дерева аналогичны действиям для прокрутки текстовых файлов: можно использовать клавиши со стрелками, клавиши [Home], [End], [PgUp] и [PgDn], а также кнопки, которые появляются на правом поле окна задачника рядом с изображением дерева, если это изображение допускает прокрутку). При прокрутке дерева вниз в окне задачника скрывается верхняя часть изображения дерева; в этом случае рядом с номером первого уровня, изображенного в окне, выводится стрелка ↑, являющаяся признаком того, что для дерева доступна прокрутка вверх. Не следует думать, что прокрутка требуется только для деревьев, созданных с ошибками. В некоторых заданиях количество уровней исходных или результирующих деревьев может превосходить число строк, отведенных для отображения этих деревьев в окне задачника; для подобных деревьев также становится доступной прокрутка.

3.2.2. Бинарные деревья поиска, сортировка деревом: Tree65

Бинарное дерево называется *деревом поиска*, если значение каждой его вершины не меньше значений вершин ее левого поддерева и не больше значений вершин ее правого поддерева. Приведем пример дерева поиска:



Если в дереве поиска отсутствуют вершины с одинаковыми значениями, будем называть его *деревом поиска без повторяющихся элементов*. Ниже приводится пример дерева поиска без повторяющихся элементов:



Деревья поиска обладают важным свойством: при обходе их вершин в инфиксном порядке значения вершин образуют неубывающую последовательность (в случае дерева поиска без повторяющихся элементов последовательность будет возрастающей). Данное свойство можно использовать в качестве определения дерева поиска (как это сделано в формулировках заданий Tree57 и Tree58). Заметим, что для перебора в инфиксном порядке вершин дерева, изображенного в окне задачника, достаточно пройти по изображениям вершин *слева направо* (горизонтальный уровень, на котором находится вершина, принимать во внимание не следует). Например, при переборе в инфиксном порядке вершин дерева, приведенного выше в качестве дерева поиска без повторяющихся элементов, мы получим следующую последовательность чисел: 19, 35, 45, 47, 49, …, 76, 82, 97, 99.

Название деревьев поиска отражает тот факт, что поиск в них вершин с определенным значением можно выполнить быстрее, чем в обычных бинарных деревьях (в которых для этого требуется просмотреть все вершины). Особенно быстро такой поиск можно выполнить для деревьев поиска без повторяющихся элементов (см. задание Tree59). В среднем (для «хорошо сбалансированного» дерева) достаточно проанализировать не более чем $\log_2 N$ вершин, где N — общее число вершин в дереве поиска. Столь же быстро работает и алгоритм вставки новой вершины в дерево поиска (см. задания Tree61–Tree64).

На этих особенностях деревьев поиска основан способ сортировки числовых последовательностей (*сортировка деревом*). На первом этапе подобной сортировки строится дерево поиска, в которое помещаются все элементы исходной последовательности («в среднем» число операций для построения такого дерева пропорционально $N \log_2 N$, где N — количество элементов сортируемой последовательности). На втором этапе организуется перебор вершин построенного дерева в инфиксном порядке, в результате которого мы получаем отсортированную последовательность исходных чисел (данный этап требует порядка N операций). После завершения сортировки необходимо разрушить созданное дерево поиска (этот этап также требует порядка N операций). Таким образом, «в среднем» число операций в алгоритме сортировки деревом пропор-

ционально $N \log_2 N$, что свидетельствует о высокой эффективности данного алгоритма (заметим, что количество операций для *алгоритма быстрой сортировки* QuickSort имеет такой же порядок).

Реализовать описанный выше алгоритм сортировки деревом требуется в задании Tree65. Единственное отличие от описанной выше схемы состоит в том, что после формирования отсортированной последовательности чисел не требуется разрушать полученное дерево поиска, так как это дерево является одним из элементов результирующих данных. Впрочем, реализовать, при необходимости, дополнительный этап разрушения дерева не составляет труда (см. решение задания Tree40 в п. 2.2.2).

При решении задания Tree65 следует определиться с выбором алгоритма добавления новой вершины в дерево поиска. Имеется несколько алгоритмов, каждый из которых позволяет создать дерево поиска, вершины которого будут содержать значения из исходного набора чисел, причем созданные деревья поиска будут отличаться друг от друга. Для того чтобы созданное дерево поиска в точности соответствовало тому, которое приводится в примере правильного решения, надо использовать алгоритм, описанный в задании Tree61. Этот алгоритм определяет действия, необходимые для добавления новой вершины со значением *K* в дерево поиска с корнем *P*, и состоит в следующем: если указатель *P* равен nil, то надо создать вершину-лист со значением *K* и присвоить указателю *P* адрес созданного листа, если же *P* не равен nil (то есть исходное дерево не пусто), то в случае, если значение корня больше, чем *K*, надо выполнить данный алгоритм для поля Left вершины *P*, иначе выполнить алгоритм для поля Right вершины *P*. Параметр *P*, передаваемый в процедуру, которая реализует этот алгоритм (назовем ее AddNode), должен быть входным и выходным параметром, так как его значение может измениться при выполнении данной процедуры.

Второй этап сортировки состоит в инфиксном переборе вершин созданного дерева и проблем не представляет (см. решение задания Tree12 в п. 1.2.3; описанную в нем процедуру NodeOutput можно без изменений перенести в программу, выполняющую задание Tree65). В результате получаем следующее решение задания Tree65:

```
program Tree65;
uses PT4;
procedure AddNode(var P: PNode; K: integer);
begin
  if P = nil then
  begin
    New(P);
    P^.Data := K;
    P^.Left := nil;
    P^.Right := nil;
  end
```

```
  else
    if K < P^.Data then
      AddNode(P^.Left, K)
    else
      AddNode(P^.Right, K);
end;
procedure NodeOutput(P: PNode);
begin
  if P = nil then exit;
  NodeOutput(P^.Left);
  PutN(P^.Data);
  NodeOutput(P^.Right);
end;
var
  N, K, I: integer;
  P1: PNode;
begin
  Task('Tree65');
  GetN(N);
  P1 := nil;
  for I := 1 to N do
  begin
    GetN(K);
    AddNode(P1, K);
  end;
  PutP(P1);
  NodeOutput(P1);
end.
```

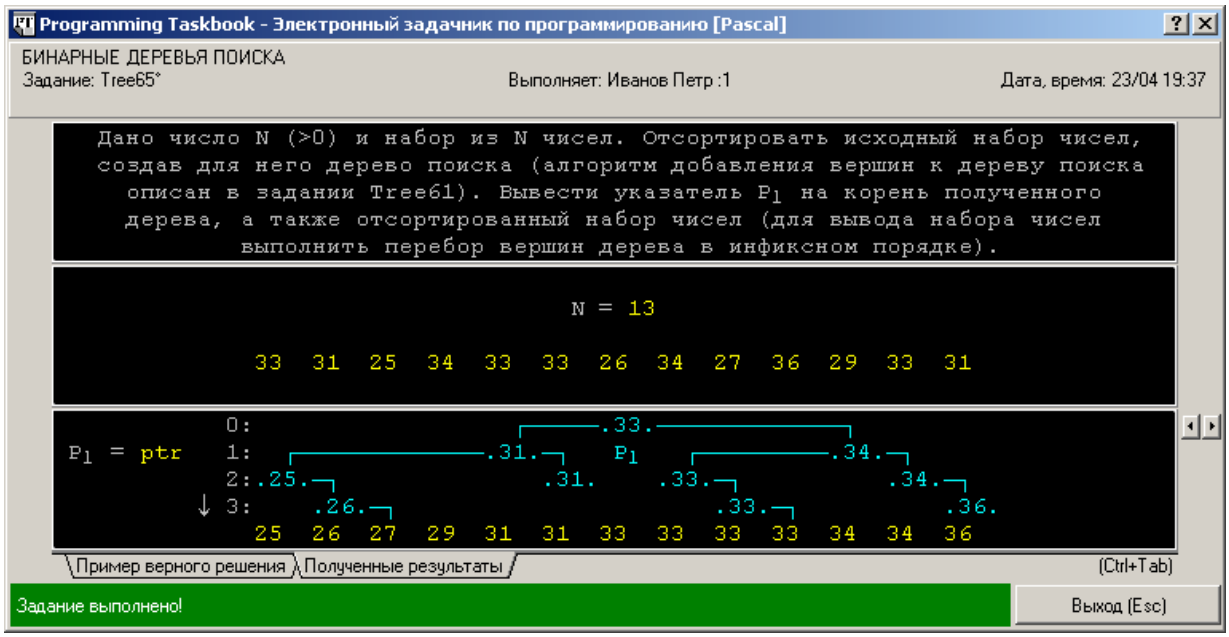


Рис. 8.

Приведем вид окна задачника при успешном выполнении задания Tree65 (см. рис. 8). Обратите внимание на то, что в разделе результатов для созданного дерева поиска доступна прокрутка.

3.3. Учебные задания и указания к ним

3.3.1. Формулировки заданий (Tree48–Tree71)

Tree48. Дан адрес P_1 вершины дерева — записи типа TNode, содержащей поля Data (целого типа), Left, Right и Parent (типа PNode — указателя на TNode). Поля Left и Right указывают на дочерние вершины, а поле Parent — на родительскую вершину данной вершины (если вершина является корнем дерева, то ее поле Parent равно nil). Для данной вершины вывести указатели P_L, P_R и P_0 на ее левую и правую дочерние вершины и родительскую вершину, а также указатель P_2 на ее *сестру*, то есть другую вершину дерева, имеющую в качестве родительской вершину с адресом P_0. Если некоторые из перечисленных вершин не существуют, то вывести для них значение nil.

Tree49. Дан указатель P_1 на корень дерева, вершинами которого являются записи типа TNode, связанные между собой с помощью полей Left и Right. Используя поле Parent записи TNode, преобразовать исходное дерево в *дерево с обратной связью*, в котором каждая вершина связана не только со своими дочерними вершинами (полями Left и Right), но и с родительской вершиной (полем Parent). Поле Parent корня дерева положить равным nil.

Tree50. Дан указатель P_1 на одну из вершин дерева с обратной связью. Вывести указатель P_2 на корень исходного дерева.

Tree51. Даны указатели P_1, P_2, P_3 на три вершины дерева с обратной связью. Для каждой из данных вершин вывести ее уровень (корень дерева имеет уровень 0).

Tree52. Даны указатели P_1 и P_2 на две различные вершины дерева с обратной связью. Вывести *степень родства* вершины P_1 по отношению к вершине P_2 (степень родства равна –1, если вершина P_2 не находится в цепочке предков для вершины P_1; в противном случае степень родства равна $L_1 - L_2$, где L_1 и L_2 — уровни вершин P_1 и P_2 соответственно).

Tree53. Даны указатели P_1 и P_2 на две различные вершины дерева с обратной связью. Вывести указатель P_0 на вершину дерева, являющуюся ближайшим общим предком вершин P_1 и P_2.

Tree54. Дан указатель P_1 на одну из вершин дерева с обратной связью. Создать копию данного дерева и вывести указатель P_2 на корень созданной копии.

Tree55. Дан указатель P_1 на вершину дерева с обратной связью, которая не является корнем. Если вершина P_1 имеет сестру, то удалить эту сестру вместе со всеми ее потомками, освободив занимаемую ими память; если вершина P_1 не имеет сестры, то создать сестру и всех ее потомков в виде копии под-

дерева с корнем P_1. Вывести указатель P_0 на родительскую вершину вершины P_1.

Tree56. Даны положительные числа L, N ($N > L$) и набор из N чисел. Создать дерево глубины L с обратной связью, содержащее вершины со значениями из исходного набора. Вершины добавлять к дереву в префиксном порядке, используя алгоритм, который для каждой вершины уровня, не превышающего L, вначале создает саму вершину с очередным значением из исходного набора, затем ее левое поддерево соответствующей глубины, а затем ее правое поддерево. Если для заполнения дерева глубины L требуется менее N вершин, то оставшиеся числа из исходного набора не использовать. Вывести указатель на корень созданного дерева.

Tree57. Дан указатель P_1 на корень непустого дерева. Если данное дерево является *деревом поиска*, то есть если при обходе его вершин в инфиксном порядке их значения образуют неубывающую последовательность, то вывести nil; в противном случае вывести адрес первой вершины (в инфиксном порядке), нарушающей требуемую закономерность.

Tree58. Дан указатель P_1 на корень непустого дерева. Если данное дерево является *деревом поиска без повторяющихся элементов*, то есть если при обходе его вершин в инфиксном порядке их значения образуют возрастающую последовательность, то вывести nil; в противном случае вывести адрес первой вершины (в инфиксном порядке), нарушающей требуемую закономерность.

Tree59. Дан указатель P_1 на корень непустого дерева поиска без повторяющихся элементов и число K. Определить, содержит ли исходное дерево вершину со значением K. Если содержит, то вывести указатель P_2 на эту вершину, в противном случае вывести nil. Вывести также количество N вершин, которые потребовалось проанализировать для выполнения задания.

Tree60. Дан указатель P_1 на корень непустого дерева поиска и число K. Вывести количество C вершин исходного дерева, имеющих значение K, а также количество N вершин, которые потребовалось проанализировать для выполнения задания.

Tree61. Дан указатель P_1 на корень дерева поиска (если дерево является пустым, то P_1 = nil). Также дано число K. Добавить к исходному дереву поиска новую вершину со значением K таким образом, чтобы полученное дерево осталось деревом поиска, и вывести указатель P_2 на корень полученного дерева. Использовать следующий рекурсивный алгоритм для дерева с корнем P: если P = nil, то создать лист со значением K и присвоить указателю P адрес созданного листа; если корень P существует, то в случае, если его значение больше K, выполнить алгоритм для поля Left корня P, иначе выполнить алгоритм для его поля Right.

Tree62. Дан указатель P_1 на корень дерева поиска без повторяющихся элементов (если дерево является пустым, то P_1 = nil). Также дано число K. Добавить к исходному дереву поиска новую вершину со значением K таким образом, чтобы полученное дерево осталось деревом поиска без повторяющихся элементов, и вывести указатель P_2 на корень полученного дерева. Если исходное дерево уже содержит вершину со значением K, то оставить дерево без изменений. Использовать следующий рекурсивный алгоритм для дерева с корнем P: если P = nil, то создать лист со значением K и присвоить указателю P адрес созданного листа; если корень P существует, то в случае, если его значение больше K, выполнить алгоритм для поля Left корня P, а в случае, если его значение меньше K, выполнить алгоритм для его поля Right.

Tree63. Дано число N (> 0) и набор из N чисел, а также указатель P_1 на корень дерева поиска (если дерево является пустым, то P_1 = nil). Добавить к исходному дереву поиска N новых вершин со значениями из исходного набора таким образом, чтобы полученное дерево осталось деревом поиска, и вывести указатель P_2 на корень полученного дерева. Для добавления новых вершин использовать алгоритм, описанный в задании Tree61.

Tree64. Дано число N (> 0) и набор из N чисел, а также указатель P_1 на корень дерева поиска без повторяющихся элементов (если дерево является пустым, то P_1 = nil). Добавить к исходному дереву поиска новые вершины со значениями из исходного набора таким образом, чтобы полученное дерево осталось деревом поиска без повторяющихся элементов, и вывести указатель P_2 на корень полученного дерева. Для добавления новых вершин использовать алгоритм, описанный в задании Tree62.

Tree65. Дано число N (> 0) и набор из N чисел. Отсортировать исходный набор чисел, создав для него дерево поиска (алгоритм добавления вершин к дереву поиска описан в задании Tree61). Вывести указатель P_1 на корень полученного дерева, а также отсортированный набор чисел (для вывода набора чисел выполнить перебор вершин дерева в инфиксном порядке).

Tree66. Дано число N (> 0) и набор из N чисел. Получить отсортированный набор исходных чисел без повторений, создав для исходного набора дерево поиска без повторяющихся элементов (алгоритм добавления вершин к подобному дереву описан в задании Tree62). Вывести указатель P_1 на корень полученного дерева, а также отсортированный набор чисел без повторений (для вывода набора чисел выполнить перебор вершин дерева в инфиксном порядке).

Tree67. Даны два указателя: P_1 на корень непустого дерева поиска и P_2 на одну из вершин этого дерева, имеющих не более одной дочерней вершины. Удалить из исходного дерева вершину с адресом P_2 так, чтобы полученное дерево осталось деревом поиска (если удаляемая вершина P_2 имеет дочер-

нюю вершину, то эту дочернюю вершину необходимо связать с родительской вершиной вершины P_2). Вывести указатель P_3 на корень полученного дерева или nil, если в результате удаления вершины P_2 дерево стало пустым.

Tree68. Даны два указателя: P_1 на корень непустого дерева поиска и P_2 на одну из вершин этого дерева, имеющих две дочерние вершины. Удалить из исходного дерева вершину P_2 так, чтобы полученное дерево осталось деревом поиска. Удаление выполнять следующим образом: в левом поддереве вершины P_2 найти вершину P с наибольшим значением, присвоить это наибольшее значение вершине P_2, после чего удалить вершину P, действуя, как в задании Tree67 (поскольку вершина P будет иметь не более одной дочерней вершины).

Tree69. Даны два указателя: P_1 на корень непустого дерева поиска и P_2 на одну из вершин этого дерева, имеющих две дочерние вершины. Удалить из исходного дерева вершину P_2 так, чтобы полученное дерево осталось деревом поиска. Удаление выполнять следующим образом: в правом поддереве вершины P_2 найти вершину P с наименьшим значением, присвоить это наименьшее значение вершине P_2, после чего удалить вершину P, действуя, как в задании Tree67 (поскольку вершина P будет иметь не более одной дочерней вершины).

Tree70. Дан указатель P_1 на одну из вершин непустого дерева поиска с обратной связью. Удалить из исходного дерева вершину P_1 таким образом, чтобы полученное дерево осталось деревом поиска с обратной связью, и вывести указатель P_2 на корень полученного дерева или nil, если в результате удаления дерево стало пустым. Если вершина P_1 имеет две дочерние вершины, то для ее удаления использовать алгоритм, описанный в задании Tree68.

Tree71. Дан указатель P_1 на одну из вершин непустого дерева поиска с обратной связью. Удалить из исходного дерева вершину P_1 таким образом, чтобы полученное дерево осталось деревом поиска с обратной связью, и вывести указатель P_2 на корень полученного дерева или nil, если в результате удаления дерево стало пустым. Если вершина P_1 имеет две дочерние вершины, то для ее удаления использовать алгоритм, описанный в задании Tree69.

3.3.2. Указания

Tree48. Вводное задание к группе заданий, посвященной бинарным деревьям с обратной связью. Для решения задания достаточно вывести значения полей исходных записей в требуемом порядке. Организовывать рекурсивный перебор вершин не требуется.

Tree49. Решение данного задания приводится в п. 3.2.1.

Tree50–53. Задания на анализ бинарного дерева с обратной связью, не требующие использования рекурсивных алгоритмов. В Tree50–51 достаточно организовать цикл, позволяющий последовательно перебрать всех предков вершины с адресом *P*, начиная с ее непосредственного предка (пока значение *P*^.Parent не равно nil, выполняется оператор присваивания *P* := *P*^.Parent). В Tree51 надо использовать вспомогательную переменную-счетчик. В Tree52 следует перебирать предков вершины P_1, пока не будет обнаружена вершина P_2 (требуется также предусмотреть обработку ситуации, когда вершина P_2 не находится в цепочке предков для вершины P_1). Задание Tree53 решается в три этапа: на первом этапе надо определить уровни исходных вершин P_1 и P_2 (ср. с Tree51), на втором этапе следует перейти от вершины с бóльшим уровнем к ее предку, уровень которого совпадает с уровнем другой вершины. На третьем этапе надо сравнивать адреса полученных вершин (находящихся на одном уровне): если адреса совпадают, значит обнаружен ближайший общий предок исходных вершин, в противном случае надо подняться на уровень вверх для каждой вершины и повторить сравнение адресов (на некотором шаге адреса вершин обязательно совпадут, так как у любых двух вершин есть по крайней мере один общий предок — корень дерева).

Tree54. Вначале надо перейти к корню исходного дерева (ср. с Tree50), затем следует воспользоваться рекурсивной функцией, аналогичной функции CopyTree, описанной в указании к Tree34. Для того чтобы для любой создаваемой вершины можно было задать ее поле Parent, в рекурсивной функции следует предусмотреть дополнительный параметр *Par*, в котором передается адрес родителя создаваемой вершины (см. решение Tree49 в п. 3.2.1, использующее рекурсивную процедуру с аналогичным параметром).

Tree55. Задание, в котором требуется преобразовать исходное дерево с обратной связью, причем преобразование может заключаться как в добавлении новых вершин (точнее, в создании нового поддерева — ср. с Tree54), так и в удалении существующих (см. решение Tree40, приведенное в п. 2.2.2). После завершения преобразования дерева следует перейти к его корню (ср. с Tree50).

Tree56. Ср. с Tree31. Рекурсивная функция CreateTree, предназначенная для создания очередной вершины дерева (см. указание к Tree25–31), должна в данном случае иметь дополнительный параметр *Par*, в котором передается адрес родителя создаваемой вершины (ср. с решением Tree49 в п. 3.2.1, использующим рекурсивную процедуру с аналогичным параметром).

Tree57–58. Особенности бинарных деревьев поиска рассматриваются в п. 3.2.2. Для того чтобы проверить, является ли бинарное дерево деревом поиска, следует организовать перебор его вершин в инфиксном порядке

(см. решение Tree12 в п. 1.2.3), в ходе которого сравнивать значение текущей вершины и предшествующей ей вершины. Значение предшествующей вершины удобно передавать в качестве дополнительного параметра рекурсивной процедуры, выполняющей перебор вершин. При обнаружении вершины, нарушающей требуемую закономерность, следует прервать последовательность рекурсивных вызовов; для этого надо использовать внешнюю по отношению к рекурсивной процедуре переменную *WrongNode* типа PNode. Перед началом перебора вершин переменная *WrongNode* полагается равной nil, при обнаружении ошибочной вершины в нее заносится адрес этой вершины. В начало рекурсивной процедуры следует добавить условный оператор, обеспечивающий немедленный выход из процедуры в случае, если ошибочная вершина уже найдена:

```
if WrongNode <> nil then exit;
```

Tree59–60. Как было отмечено в п. 3.2.2, для нахождения в дереве поиска вершины с требуемым значением нет необходимости просматривать все вершины. Особенно просто поиск организуется для деревьев без повторяющихся элементов; приведем описание соответствующей рекурсивной функции, возвращающей вершину с требуемым значением *K* или nil, если дерево не содержит вершины с этим значением:

```
function FindNode(P: PNode; K: integer): PNode;
begin
  if P = nil then
    Result := nil
  else
    if P^.Data = K then
      Result := P
    else
      if P^.Data > K then
        Result := FindNode(P^.Left, K)
      else
        Result := FindNode(P^.Right, K);
end;
```

При стартовом вызове этой функции в качестве *P* указывается адрес корня дерева поиска. При выполнении Tree59 в приведенную функцию необходимо добавить фрагмент, позволяющий определить число вершин, которые были проанализированы в ходе поиска требуемой вершины (для подсчета просмотренных вершин надо использовать переменную-счетчик, являющуюся внешней по отношению к рекурсивной функции). Аналогичное дополнение необходимо сделать и при выполнении Tree60; кроме того, в данном случае надо изменить функцию так, чтобы она возвращала общее количество найденных вершин с требуемым значением (следует также учесть, что если в дереве поиска с повторяющимися элементами для теку-

щей вершины P имеет место равенство P^.Data = K, то поиск следует продолжать как в левом, так и в правом поддереве вершины P).

Tree61–64. Задания, связанные с добавлением к дереву поиска новых вершин. Процедура AddNode, реализующая алгоритм добавления новой вершины к дереву поиска с повторяющимися элементами, приводится в п. 3.2.2. В случае деревьев поиска без повторяющихся элементов в эту процедуру следует внести небольшое изменение, чтобы не допустить добавления вершины, если дерево уже содержит вершину с указанным значением (см. алгоритм добавления вершины, приведенный в формулировке задания Tree62).

Tree65–66. Решение Tree65 приводится в п. 3.2.2; задание Tree66 решается аналогично.

Tree67. В начале выполнения данного задания удобно сохранить во вспомогательном указателе P_3 адрес существующего потомка для удаляемой вершины P_2 (если вершина P_2 не имеет потомков, то значение P_3 будет равно nil):

```
P3 := P2^.Left;
if P3 = nil then
  P3 := P2^.Right;
```

Особую ситуацию $P_2 = P_1$ (удаляемая вершина совпадает с корнем дерева) следует обрабатывать отдельно; заметим, что только в этом случае изменяется корень исходного дерева. В ситуации, когда удаляемая вершина не совпадает с корнем, надо найти непосредственного предка вершины P_2 и изменить его поле связи (Left или Right), равное P_2, на P_3. Как и во всех заданиях, связанных с удалением вершин (см. п. 2.2.2), необходимо также вызвать процедуру Dispose для удаляемой вершины P_2.

Tree68–69. Задания, демонстрирующие два способа удаления вершины из дерева поиска в случае, если удаляемая вершина имеет два непосредственных потомка (алгоритм удаления описывается в формулировках заданий). При выполнении этих заданий не требуется использовать рекурсивные подпрограммы. Для поиска вершины с наибольшим значением в левом поддереве (Tree68) достаточно выполнить перебор в цикле *правых* дочерних вершин этого поддерева: пока значение P^.Right не равно nil, выполняется присваивание P := P^.Right. Аналогично, для поиска вершины с наименьшим значением в правом поддереве (Tree69) достаточно организовать перебор *левых* дочерних вершин (P := P^.Left). Заметим, что в ходе этого перебора желательно определить не только требуемую вершину с наименьшим/наибольшим значением, но и ее родительскую вершину, поскольку родительская вершина также используется в алгоритме удаления (см. Tree67 и указание к нему).

Tree70–71. См. описание особенностей деревьев с обратной связью (п. 3.2.1) и указания к Tree67 и Tree68–69. Благодаря наличию обратной связи алгоритм удаления вершин упрощается, так как отпадает необходимость в поиске непосредственного предка для удаляемой вершины (достаточно использовать ее поле Parent).

3.4. Проектное задание

Выполните учебные задания группы Tree, указанные в вашем варианте проектного задания. Если вы не получили вариант проектного задания, то выполните задания из первого варианта.

ВАРИАНТ 1 (1) Деревья с обратной связью: 50, 54 (2) Деревья поиска: 59, 64, 69	**ВАРИАНТ 2** (1) Деревья с обратной связью: 53, 56 (2) Деревья поиска: 58, 61, 67
ВАРИАНТ 3 (1) Деревья с обратной связью: 52, 54 (2) Деревья поиска: 60, 62, 66	**ВАРИАНТ 4** (1) Деревья с обратной связью: 53, 56 (2) Деревья поиска: 58, 61, 71
ВАРИАНТ 5 (1) Деревья с обратной связью: 52, 55 (2) Деревья поиска: 58, 61, 71	**ВАРИАНТ 6** (1) Деревья с обратной связью: 50, 55 (2) Деревья поиска: 57, 64, 70
ВАРИАНТ 7 (1) Деревья с обратной связью: 53, 54 (2) Деревья поиска: 60, 62, 70	**ВАРИАНТ 8** (1) Деревья с обратной связью: 51, 55 (2) Деревья поиска: 59, 62, 69
ВАРИАНТ 9 (1) Деревья с обратной связью: 51, 56 (2) Деревья поиска: 57, 64, 66	**ВАРИАНТ 10** (1) Деревья с обратной связью: 52, 56 (2) Деревья поиска: 59, 63, 68
ВАРИАНТ 11 (1) Деревья с обратной связью: 51, 55 (2) Деревья поиска: 57, 63, 68	**ВАРИАНТ 12** (1) Деревья с обратной связью: 50, 54 (2) Деревья поиска: 60, 63, 67
ВАРИАНТ 13 (1) Деревья с обратной связью: 50, 54 (2) Деревья поиска: 60, 61, 67	**ВАРИАНТ 14** (1) Деревья с обратной связью: 52, 56 (2) Деревья поиска: 57, 62, 69

ВАРИАНТ 15 (1) Деревья с обратной связью: 51, 55 (2) Деревья поиска: 60, 61, 66	**ВАРИАНТ 16** (1) Деревья с обратной связью: 53, 54 (2) Деревья поиска: 58, 62, 71
ВАРИАНТ 17 (1) Деревья с обратной связью: 50, 56 (2) Деревья поиска: 58, 62, 70	**ВАРИАНТ 18** (1) Деревья с обратной связью: 53, 55 (2) Деревья поиска: 59, 63, 70
ВАРИАНТ 19 (1) Деревья с обратной связью: 50, 56 (2) Деревья поиска: 58, 61, 67	**ВАРИАНТ 20** (1) Деревья с обратной связью: 53, 54 (2) Деревья поиска: 57, 64, 69
ВАРИАНТ 21 (1) Деревья с обратной связью: 52, 54 (2) Деревья поиска: 59, 63, 68	**ВАРИАНТ 22** (1) Деревья с обратной связью: 52, 55 (2) Деревья поиска: 59, 63, 71
ВАРИАНТ 23 (1) Деревья с обратной связью: 51, 55 (2) Деревья поиска: 57, 64, 66	**ВАРИАНТ 24** (1) Деревья с обратной связью: 51, 56 (2) Деревья поиска: 60, 64, 68

3.5. Тест рубежного контроля

1. Укажите, какое дополнительное поле записи TNode используется в деревьях с обратной связью и позволяет перейти от дочерней вершине к ее непосредственному предку.			
(1)	Back	(2)	Prev
(3)	Parent	(4)	Pred
2. Каким образом в окне задачника отмечается, что указатель на предка для корня дерева с обратной связью не равен nil?			
(1)	Значение корня дерева выводится красным цветом	(2)	Над значением корня дерева выводится красная звездочка
(3)	Значение корня дерева обрамляется точками	(4)	Под значением корня дерева выводится красная звездочка

3. Укажите *неправильное* продолжение для следующего начала фразы: «Если бинарное дерево является деревом поиска, то…			
(1)	значение каждой его вершины не больше значений вершин ее правого поддерева»	(2)	максимальное значение в левом поддереве любой вершины не превосходит значение этой вершины»
(3)	при обходе его вершин в инфиксном порядке значения вершин образуют неубывающую последовательность»	(4)	при обходе его вершин в префиксном порядке значения вершин образуют невозрастающую последовательность»
4. Укажите, какое максимальное количество вершин «хорошо сбалансированного» дерева поиска без повторяющихся элементов надо проанализировать для того, чтобы выполнить в нем поиск вершины с требуемым значением (предполагается, что дерево содержит N вершин, а значения указанных выражений округляются до ближайшего целого).			
(1)	$N^{1/2}$	(2)	$\log_{10} N$
(3)	$\log_2 N$	(4)	$N / 2$
5. Укажите правильное продолжение описания алгоритма для добавления новой вершины со значением K в дерево поиска без повторяющихся элементов с корнем R: «Если R = nil, то создать лист со значением K и присвоить указателю R адрес этого листа; если корень R существует, то…			
(1)	в случае, если его значение больше K, выполнить алгоритм для поля Left корня R, иначе выполнить алгоритм для его поля Right»	(2)	в случае, если его значение меньше K, выполнить алгоритм для поля Left корня R, иначе выполнить алгоритм для его поля Right»
(3)	в случае, если его значение больше K, выполнить алгоритм для поля Left корня R, а в случае, если его значение меньше K, выполнить алгоритм для его поля Right»	(4)	в случае, если его значение меньше K, выполнить алгоритм для поля Left корня R, а в случае, если его значение больше K, выполнить алгоритм для его поля Right»
6. Укажите, какому значению пропорционально число операций в алгоритме сортировки деревом, примененном к набору из N элементов.			
(1)	N^2	(2)	$N \log_2 N$
(3)	$N \log_{10} N$	(4)	$N^{3/2}$

4. Модуль № 4. Бинарные деревья разбора выражений и деревья общего вида

4.1. Комплексная цель

Ознакомить с методами разбора выражений, основанными на представлении их в виде деревьев. Изучить методы обработки деревьев общего вида при их представлении в виде бинарных деревьев, интерпретируемых специальным образом.

4.2. Содержание модуля

4.2.1. Бинарные деревья разбора выражений: Tree74, Tree75

Задания Tree72–Tree85, приведенные в п. 4.3.1, можно разбить на две группы. В заданиях первой группы требуется по строке-описанию некоторого выражения сформировать *дерево разбора* этого выражения (Tree72, Tree74, Tree76–Tree78, Tree83). Вторая группа содержит «обратные» задания: в них дается дерево разбора выражения, по которому требуется либо вычислить данное выражение (Tree79, Tree84), либо сформировать его строковое описание (Tree73, Tree75, Tree80–82, Tree85). В данном пункте приводятся примеры решения заданий из обеих групп.

Задания, посвященные разбору выражений, дополняют соответствующий раздел группы Recur (см. [3, 4]). Напомним приведенные в [4] рекомендации, которым желательно следовать при организации разбора выражений. Будем считать, что исходное выражение хранится в строке *S*. Для последовательного перебора элементов выражения удобно предусмотреть специальную функцию. Поскольку во всех видах выражений, рассматриваемых в заданиях из п. 4.3.1, элементами выражения являются отдельные *символы*, назовем эту функцию NextChar. Данная функция использует внешнюю по отношению к ней переменную-счетчик *K*, которая содержит номер очередного символа строки *S*. При каждом вызове функции NextChar она увеличивает счетчик *K* на 1 и возвращает символ *S*[*K*] или особый символ #0 (символ с кодом 0), если значение *K* превышает длину строки *S*:

```
function NextChar: char;
begin
  Inc(K);
  if K <= Length(S) then
    Result := S[K]
  else
```

```
    Result := #0;
end;
```

В некоторых ситуациях при разборе выражения требуется сделать «шаг назад» для того, чтобы только что прочитанный символ строки *S* был повторно прочитан и окончательно обработан на одном из последующих этапов разбора выражения. Ясно, что для этого достаточно уменьшить на 1 значение счетчика *K*, однако для повышения наглядности целесообразно оформить подобное действие в виде вспомогательной процедуры BackChar:

```
procedure BackChar;
begin
  Dec(K);
end;
```

Символы обрабатываемого выражения, являющиеся цифрами, обычно требуется преобразовать в соответствующие числовые значения. Для этого проще всего воспользоваться выражением Ord(*C*) – Ord(′0′), где *C* — символ, изображающий цифру (здесь используется тот факт, что символы «0»–«9» располагаются в кодовой таблице символов последовательно).

Переменную-счетчик *K*, вспомогательные подпрограммы NextChar и BackChar, а также рекурсивную функцию, осуществляющую разбор выражения и формирование дерева, удобно заключить в «оболочку» нерекурсивной функции, которая выполняет инициализирующие действия (в нашем случае полагает счетчик *K* равным нулю), осуществляет стартовый запуск рекурсивной функции и возвращает указатель на созданное дерево разбора. Исходное выражение передается в данную функцию в качестве строкового параметра *S*.

Воспользуемся всеми перечисленными рекомендациями для решения задания Tree74, в котором требуется сформировать бинарное дерево на основе его строкового описания, имеющего следующий формат:

```
<дерево>   ::= <вершина> |
               <вершина>(<левое поддерево>,<правое поддерево>) |
               <вершина>(<левое поддерево>) |
               <вершина>(,<правое поддерево>)
<вершина> ::= <цифра>
```

Варианты исходных строк и изображения соответствующих бинарных деревьев можно просмотреть, выполняя ознакомительные запуски программы с заготовкой для данного задания (см. рис. 9):

```
program Tree74;
uses PT4;
begin
  Task('Tree74');
end.
```

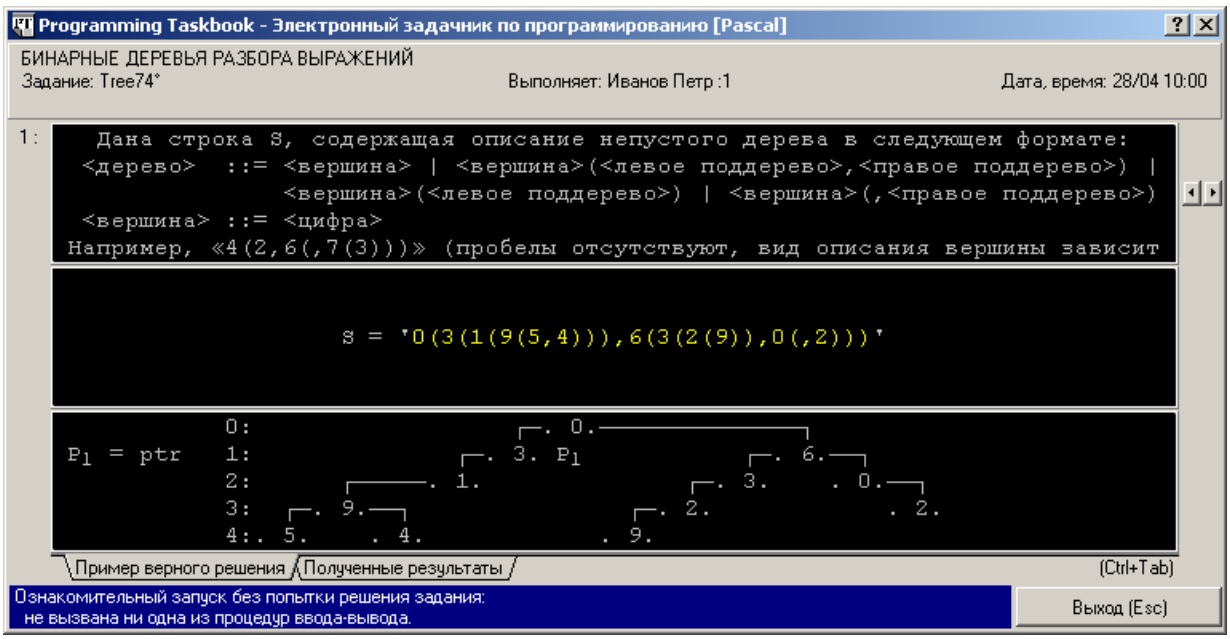


Рис. 9.

Напомним, что точки, обрамляющие все вершины результирующего дерева, означают, что эти вершины должны быть созданы в программе учащегося.

Следует отметить, что выражение, используемое в задании Tree74, является одним из наиболее сложных в данной группе заданий, поскольку каждая вершина в нем может описываться четырьмя различными способами (в зависимости от количества и вида ее дочерних вершин). Перечислим эти способы, предполагая, что значение вершины равно 5:

1) вершина является листом, то есть не имеет дочерних вершин: `5`;
2) вершина имеет только левое поддерево: `5(`*<левое поддерево>*`)`;
3) вершина имеет только правое поддерево: `5(,`*<правое поддерево>*`)`;
4) вершина имеет оба дочерних поддерева:
   `5(`*<левое поддерево>*`,`*<правое поддерево>*`)`.

Мы видим, что для любой вершины вначале указывается ее значение (равное значению поля Data записи TNode). Информация о ее дочерних вершинах определяется по наличию или отсутствию скобок и символа «,» (запятая). Ясно, что если реализовать функцию CreateNode, создающую очередную вершину дерева и возвращающую указатель на созданную вершину, то для формирования поддеревьев созданной вершины достаточно вызывать эту функцию рекурсивно (и присваивать возвращенные ею значения полям Left и/или Right созданной вершины). Значение создаваемой вершины удобно передавать в функцию CreateNode в качестве символьного параметра *C*, содержащего изображение соответствующей цифры (напомним, что по условию задания значение каждой вершины дерева определяется одной цифрой). Приведем описание функции CreateNode, считая, что ранее в программе уже описаны вспомогательные подпрограммы NextChar и BackChar:

```
function CreateNode(C: char): PNode;
begin
  New(Result);
  Result^.Data := Ord(C) - Ord('0');
  Result^.Left := nil;
  Result^.Right := nil;
  if NextChar <> '(' then                          {1}
  begin
    BackChar;
    exit;
  end;
  C := NextChar;                                   {2}
  if C <> ',' then
  begin
    Result^.Left := CreateNode(C);                 {3}
    C := NextChar;                                 {4}
  end;
  if C = ',' then                                  {5}
  begin
    Result^.Right := CreateNode(NextChar);
    NextChar;
  end;
end;
```

Ввиду сложности данной функции прокомментируем каждый из ее фрагментов.

После выделения памяти процедурой New для новой вершины-записи в ее поле Data заносится значение вершины (при этом символ-цифра *C* преобразуется в соответствующее ему числовое значение), а поля-указатели Left и Right инициализируются значениями nil.

Затем из исходной строки с помощью функции NextChar извлекается очередной символ, позволяющий определить, содержит ли созданная вершина дочерние вершины (см. оператор, помеченный комментарием {1}). Если этот символ не равен открывающей скобке «(», значит вершина является листом, и ее создание завершено. В этом случае перед выходом из функции необходимо вернуть последний прочитанный символ обратно, используя процедуру BackChar, так как этот символ не относится к той вершине, которая была только что создана (возвращенный символ будет обработан на следующих этапах разбора исходной строки). Заметим, что если исходная строка состоит из единственной цифры (например, «5»), то при попытке чтения очередного символа мы выйдем за пределы строки. Поскольку эта ситуация предусмотрена в функции NextChar, ошибки не произойдет (будет возвращен символ с кодом 0, после анализа которого функция завершит работу; в результате будет создано дерево, состоящее из единственной вершины).

Если очередной символ равен открывающей скобке, то это означает, что созданная вершина имеет дочерние вершины. Чтобы выяснить, какие именно дочерние вершины она имеет, необходимо проанализировать следующий символ строки. Этот символ считывается в переменную *C* (см. оператор, помеченный комментарием {2}). Если символ не равен «,», значит вершина содержит левое поддерево, и для его создания следует выполнить рекурсивный вызов функции CreateNode, передав ей в качестве параметра символ *C* (см. оператор {3}). После формирования левого поддерева необходимо прочесть символ, расположенный в исходной строке после описания этого дерева (см. оператор {4}); этим символом может быть либо запятая, либо закрывающая скобка.

На последний условный оператор данной функции (помеченный комментарием {5}) мы можем перейти либо сразу после выполнения оператора {2} (если в результате выполнения оператора {2} переменная *C* получит значение «,»), либо после выполнения действий, связанных с созданием левого поддерева (см. операторы {3} и {4}; в этом случае переменная *C* может содержать одно из двух значений: «,» или «)»). Если *C* равна «)», значит обработка текущей вершины завершена. Если *C* равна «,», значит обрабатываемая вершина содержит правое поддерево, которое необходимо создать, выполнив очередной рекурсивный вызов функции CreateNode, после чего прочесть из исходной строки еще один символ — закрывающую скобку (для этого достаточно вызвать функцию NextChar как процедуру, в виде отдельного оператора, поскольку ее возвращаемое значение в дальнейшем не используется).

Осталось привести текст функции-оболочки для описанных ранее вспомогательных подпрограмм (назовем эту функцию T74 по имени задания, которое она позволяет решить), а также раздел операторов основной программы:

```
program Tree74;
uses PT4;
function T74(S: string): PNode;
var
  K: integer;
<здесь должны располагаться описания подпрограмм NextChar,
BackChar и CreateNode (в указанном порядке)>
begin
  K := 0;
  Result := CreateNode(NextChar);
end;
var
  S: string;
begin
  Task('Tree74');
  GetS(S);
  PutP(T74(S));
end.
```

Обратимся к заданию Tree75, в котором требуется выполнить «обратное» действие: по исходному бинарному дереву сформировать его строковое описание в указанном формате. Задания подобного типа сводятся к перебору вершин дерева в определенном порядке, в ходе которого формируется требуемая строка. Поэтому решение задания Tree75 будет проще, чем решение ранее рассмотренного задания Tree74. В частности, в нем не надо описывать оболочку для рекурсивной функции, поскольку рекурсивная функция (которую можно назвать TreeToStr) не использует внешние переменные:

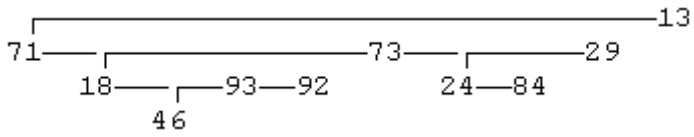
```
program Tree75;
uses PT4;
function TreeToStr(P: PNode): string;
begin
  Result := Chr(Ord('0') + P^.Data);
  if (P^.Left = nil) and (P^.Right = nil) then exit;
  Result := Result + '(';
  if P^.Left <> nil then
    Result := Result + TreeToStr(P^.Left);
  if P^.Right <> nil then
    Result := Result + ',' + TreeToStr(P^.Right);
  Result := Result + ')';
end;
var
  P1: PNode;
begin
  Task('Tree75');
  GetP(P1);
  PutS(TreeToStr(P1));
end.
```

В функции TreeToStr приходится преобразовывать числовое значение *P*^.Data (лежащее в диапазоне 0–9) в его строковое представление; для этого достаточно воспользоваться выражением Chr(Ord('0') + *P*^.Data). Прочие фрагменты решения дополнительных комментариев не требуют.

4.2.2. Деревья общего вида: Tree86

С помощью связанных записей типа TNode можно моделировать не только бинарные деревья, но и произвольные упорядоченные деревья, вершины которых имеют любое число непосредственных потомков (такие деревья называются *деревьями общего вида*, или *деревьями с произвольным ветвлением*). Необходимо лишь изменить смысл ссылок, содержащихся в полях Left и Right. В дереве общего вида поле Left любой внутренней вершины *P* содержит указатель на первую дочернюю вершину (то есть левую дочернюю вершину, если, как обычно, считать, что дочерние вершины располагаются в порядке слева направо), а поле Right — указатель на следующую (то есть правую) *сестру* верши-

ны *P*, то есть на вершину, имеющую того же родителя, что и вершина *P* (в английском языке для вершин, имеющих общего родителя, используется слово sibling, означающее как родного брата, так и родную сестру). Ниже приводится пример дерева общего вида, которое реализовано с помощью связанных записей типа TNode (аналогичным образом деревья общего вида изображаются в окне задачника):



Корень этого дерева (со значением 13) имеет три дочерние вершины (71, 73 и 29), причем вершина 71 не имеет потомков, вершина 73 имеет три непосредственных потомка (18, 93 и 92), а вершина 29 — два (24 и 84). На последнем уровне располагается вершина 46, являющаяся единственной дочерней вершиной вершины 93. Обратите внимание на то, что поле Right *корня* дерева общего вида всегда равно nil, так как корень дерева сестер не имеет.

Рассмотрим задание Tree86 — первое из заданий, связанных с деревьями общего вида. В этом задании дано бинарное дерево и требуется создать дерево общего вида, структура которого соответствует структуре исходного бинарного дерева. Заметим, что при переходе от бинарного дерева к дереву общего вида часть информации о структуре бинарного дерева теряется, поскольку в случае, если некоторая вершина дерева общего вида имеет только одного непосредственного потомка, нельзя определить, каким был этот потомок в исходном бинарном дереве — левым или правым.

Приведем вид окна задачника при ознакомительном запуске задания Tree86 (см. рис. 10).

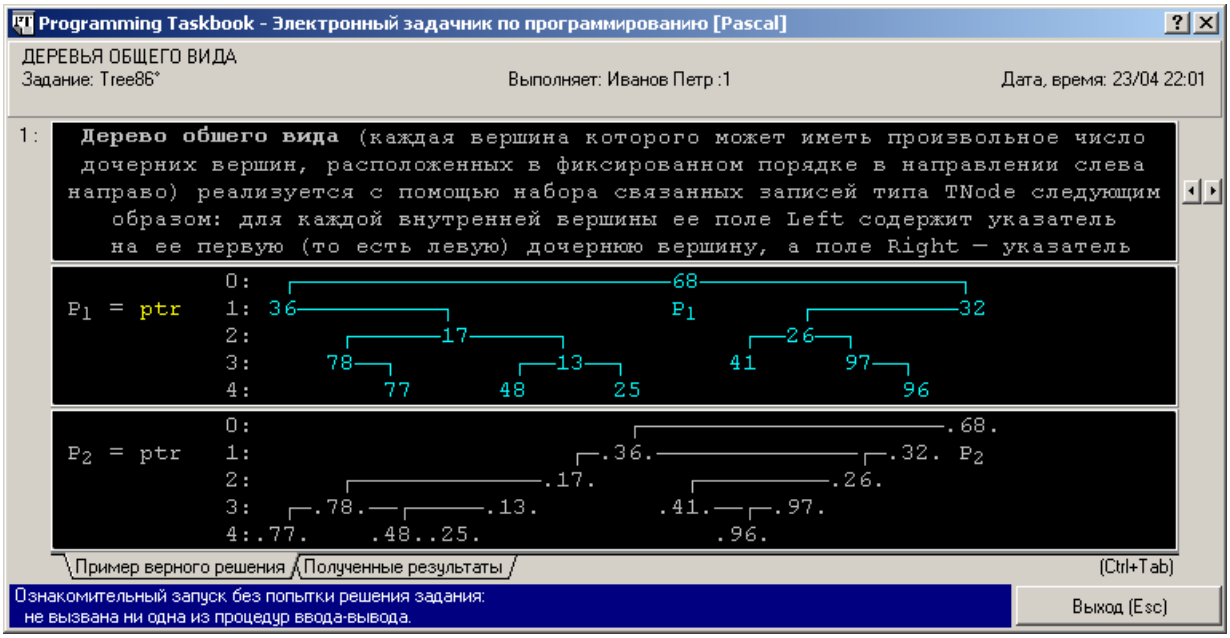


Рис. 10.

Обратите внимание на то, как выглядит одно и то же дерево в двух различных представлениях: вариант, соответствующий обычному бинарному дереву, приводится в разделе исходных данных, а вариант, соответствующий дереву общего вида, — в разделе результатов.

При формировании нового дерева будем использовать рекурсивную функцию CreateNode(*P*). Параметр *P* содержит указатель на вершину исходного дерева, копия которой создается при вызове функции. Возвращаемым значением функции является указатель на созданную вершину (как обычно, если *P* = nil, то функция не выполняет никаких действий и возвращает nil). Для создания дочерних вершин выполняется рекурсивный вызов этой функции. Заметим, что цепочка дочерних вершин может быть пустой (если вершина *P* является листом), содержать один элемент (если вершина *P* имеет только одного непосредственного потомка) или два элемента. Перед формированием цепочки дочерних вершин удобно занести адреса дочерних вершин вершины *P* во вспомогательные переменные P_1 и P_2. При этом в случае, если вершина *P* имеет только одного потомка (неважно, левого или правого), адрес этого потомка заносится в переменную P_1, а переменная P_2 остается равной nil. Благодаря использованию переменных P_1 и P_2 фрагмент кода, отвечающий за формирование списка дочерних вершин, удается сделать более кратким. Приведем текст программы, решающей задание Tree86.

```
program Tree86;
uses PT4;
function CreateNode(P: PNode): PNode;
var
  P1, P2: PNode;
begin
  if P = nil then
  begin
    Result := nil;
    exit;
  end;
  New(Result);
  Result^.Data := P^.Data;
  Result^.Right := nil;
  P1 := P^.Left;
  P2 := P^.Right;
  if P1 = nil then
  begin
    P1 := P2;
    P2 := nil;
  end;
  { формирование списка дочерних вершин }
  Result^.Left := CreateNode(P1);
```

```
  if P1 <> nil then
    Result^.Left^.Right := CreateNode(P2);
end;
var
  P1: PNode;
begin
  Task('Tree86');
  GetP(P1);
  PutP(CreateNode(P1));
end.
```

Завершая обсуждение деревьев общего вида, отметим, что фрагмент дерева, содержащий все дочерние вершины некоторой вершины, можно рассматривать как *односвязный список*, элементы которого связаны между собой с помощью поля Right (у последнего элемента списка поле Right равно nil). Каждый элемент подобного списка может содержать «подсписок» своих дочерних элементов; адрес начала этого подсписка хранится в поле Left данного элемента. Поэтому в алгоритмах, связанных с обработкой вершин деревьев общего вида, для перебора непосредственных потомков некоторой вершины удобно использовать цикл (как при переборе элементов списка), в то время как для обработки каждой дочерней вершины следует, как обычно, использовать рекурсию. Приведем общую схему подобных алгоритмов, реализовав ее в виде процедуры Tree:

```
procedure Tree(P: PNode);
var
  P0: PNode;
begin
  if P = nil then exit;
  <обработка вершины P>
  P0 := P^.Left;
  while P0 <> nil do
  begin
    Tree(P0);
    P0 := P0^.Right;
  end;
end;
```

В некоторых ситуациях вместо процедуры удобно использовать функцию; кроме того, если обработка вершины зависит от результатов обработки ее потомков, соответствующий фрагмент алгоритма следует поместить *после* цикла while. Разумеется, если обработка дерева состоит в добавлении или удалении его вершин или изменении порядка их следования, то описанная выше схема требует соответствующей модификации.

4.3. Учебные задания и указания к ним

4.3.1. Формулировки заданий (Tree72–Tree100)

Tree72. Дана строка *S*, содержащая описание непустого дерева в следующем формате:

<дерево> ::= <пусто> |
<вершина>(<левое поддерево>,<правое поддерево>)

<вершина> ::= <цифра>

Например, «4(2(,),6(,7(3(,),)))» (пробелы отсутствуют). Создать дерево по описанию, приведенному в *S*, и вывести указатель на его корень.

Tree73. Дан указатель P_1 на корень непустого дерева. Вывести строку с описанием исходного дерева в формате, приведенном в задании Tree72.

Tree74. Дана строка *S*, содержащая описание непустого дерева в следующем формате:

<дерево> ::= <вершина> |
<вершина>(<левое поддерево>,<правое поддерево>) |
<вершина>(<левое поддерево>) |
<вершина>(,<правое поддерево>)

<вершина> ::= <цифра>

Например, «4(2,6(,7(3)))» (пробелы отсутствуют, вид описания вершины зависит от того, имеет ли вершина непустое левое и/или правое поддерево). Создать дерево по описанию, приведенному в *S*, и вывести указатель на его корень.

Tree75. Дан указатель P_1 на корень непустого дерева. Вывести строку с описанием исходного дерева в формате, приведенном в задании Tree74.

Tree76. Дана строка *S*, содержащая описание числового выражения в следующем формате:

<выражение> ::= <цифра> | (<выражение><знак><выражение>)

<знак> ::= + | – | *

Пробелы в строке отсутствуют. Создать дерево, соответствующее исходному выражению (*дерево разбора выражения*): каждая внутренняя вершина дерева должна соответствовать одной из трех возможных арифметических операций и иметь значение –1 для операции сложения, –2 для операции вычитания и –3 для операции умножения; левое и правое дочерние поддеревья любой внутренней вершины-операции должны соответствовать выражениям слева и справа от знака операции; листьями полученного дерева должны быть выражения-цифры. Вывести указатель на корень созданного дерева.

Tree77. Дана строка *S*, содержащая описание числового выражения в следующем формате (так называемый *префиксный бесскобочный формат* записи числового выражения):

<выражение> ::= <цифра> | <знак> <выражение> <выражение>

<знак> ::= + | – | *

Выражения отделяются друг от друга и от знаков операций ровно одним пробелом. Создать дерево разбора выражения и вывести указатель на его корень. Структура дерева разбора выражения описана в задании Tree76; для каждой вершины-операции ее левое поддерево должно соответствовать первому операнду данной операции, а правое поддерево — второму.

Tree78. Дана строка *S*, содержащая описание числового выражения в следующем формате (так называемый *постфиксный бесскобочный формат* записи числового выражения):

<выражение> ::= <цифра> | <выражение> <выражение> <знак>

<знак> ::= + | – | *

Выражения отделяются друг от друга и от знаков операций ровно одним пробелом. Создать дерево разбора выражения и вывести указатель на его корень. Структура дерева разбора выражения описана в задании Tree76; для каждой вершины-операции ее левое поддерево должно соответствовать первому операнду данной операции, а правое поддерево — второму.

Tree79. Дан указатель P_1 на корень непустого дерева разбора выражения (структура дерева описана в задании Tree76). Вывести значение выражения, определяемого исходным деревом.

Tree80. Дан указатель P_1 на корень непустого дерева разбора выражения (структура дерева описана в задании Tree76). Вывести строковое представление соответствующего выражения в формате, приведенном в том же задании:

<выражение> ::= <цифра> | (<выражение><знак><выражение>)

<знак> ::= + | – | *

Tree81. Дан указатель P_1 на корень непустого дерева разбора выражения. Вывести строковое представление соответствующего выражения в префиксном бесскобочном формате (см. задание Tree77).

Tree82. Дан указатель P_1 на корень непустого дерева разбора выражения. Вывести строковое представление соответствующего выражения в постфиксном бесскобочном формате (см. задание Tree78).

Tree83. Дана строка *S*, содержащая описание числового выражения в следующем формате (функция M возвращает максимальное из двух выражений, а m — минимальное):

<выражение> ::= <цифра> | M(<выражение>,<выражение>) |
m(<выражение>,<выражение>)

Пробелы в строке отсутствуют. Создать дерево разбора исходного выражения: каждая внутренняя вершина дерева должна соответствовать одной из двух возможных функций и иметь значение –1 для функции M и –2 для функции m; для каждой вершины-функции ее левое поддерево должно соответствовать первому аргументу функции, а правое поддерево — второму; листьями полученного дерева должны быть выражения-цифры. Вывести указатель на корень созданного дерева.

Tree84. Дан указатель P_1 на корень непустого дерева разбора выражения (структура дерева описана в задании Tree83). Вывести значение выражения, определяемого исходным деревом.

Tree85. Дан указатель P_1 на корень непустого дерева разбора выражения (структура дерева описана в задании Tree83). Вывести строковое представление соответствующего выражения в формате, приведенном в том же задании:

<выражение> ::= <цифра> | M(<выражение>,<выражение>) |
m(<выражение>,<выражение>)

Tree86. *Дерево общего вида* (каждая вершина которого может иметь произвольное число дочерних вершин, расположенных в фиксированном порядке в направлении слева направо) реализуется с помощью набора связанных записей типа TNode следующим образом: для каждой внутренней вершины ее поле Left содержит указатель на ее первую (то есть левую) дочернюю вершину, а поле Right — указатель на ее правую *сестру*, то есть вершину, имеющую в дереве общего вида того же родителя. Поле Right корня дерева общего вида всегда равно nil, так как корень сестер не имеет. Дан указатель P_1 на корень непустого бинарного дерева. Создать дерево общего вида, соответствующее исходному бинарному дереву, и вывести указатель P_2 на его корень.

Tree87. Дан указатель P_1 на корень непустого дерева общего вида. Любая вершина исходного дерева имеет не более двух дочерних вершин. Создать бинарное дерево, соответствующее исходному дереву общего вида, и вывести указатель P_2 на его корень. Считать, что первая дочерняя вершина любой вершины дерева общего вида соответствует левой дочерней вершине в бинарном дереве.

Tree88. Дан указатель P_1 на корень непустого дерева общего вида. Вывести *глубину* данного дерева, то есть значение его максимального уровня, считая, что все вершины-сестры находятся на одном уровне, а корень дерева расположен на уровне 0.

Tree89. Дан указатель P_1 на корень непустого дерева общего вида. Для каждого из уровней данного дерева, начиная с уровня 0, вывести количество вершин, находящихся на этом уровне. Считать, что глубина дерева общего вида не превосходит 10.

Tree90. Дан указатель P_1 на корень непустого дерева общего вида. Для каждого из уровней данного дерева, начиная с уровня 0, вывести сумму значений вершин, находящихся на этом уровне. Считать, что глубина дерева общего вида не превосходит 10.

Tree91. Дан указатель P_1 на корень непустого дерева общего вида. Также дано неотрицательное число L. Перебирая дочерние вершины в заданном порядке (то есть слева направо), вывести значения всех вершин уровня L и их количество N. Если дерево не содержит вершин уровня L, то вывести 0.

Tree92. Дан указатель P_1 на корень непустого дерева общего вида. Вывести значения всех вершин дерева в *инфиксном порядке*: вначале выводится содержимое первого (левого) поддерева в инфиксном порядке, затем выводится значение корня, а затем — содержимое остальных поддеревьев в инфиксном порядке (поддеревья перебираются слева направо).

Tree93. Дан указатель P_1 на корень непустого дерева общего вида. Вывести значения всех вершин дерева в *постфиксном порядке*: вначале выводится содержимое каждого поддерева в постфиксном порядке (поддеревья перебираются слева направо), а затем — значение корня.

Tree94. Дан указатель P_1 на корень непустого дерева общего вида. Также дано неотрицательное число N. Вывести количество вершин исходного дерева, имеющих ровно N дочерних вершин. Если требуемые вершины отсутствуют, то вывести 0.

Tree95. Дан указатель P_1 на корень непустого дерева общего вида. Вывести указатель P_2 на первую вершину дерева, имеющую наибольшее количество дочерних вершин. Вершины перебирать в инфиксном порядке (см. задание Tree92).

Tree96. Дан указатель P_1 на корень непустого дерева общего вида. Вывести указатель P_2 на последнюю вершину дерева с наибольшей суммой значений дочерних вершин. Вершины перебирать в постфиксном порядке (см. задание Tree93).

Tree97. Дан указатель P_1 на корень непустого дерева общего вида. В каждом наборе вершин-сестер заменить все значения вершин (то есть значения полей Data) на максимальное из их исходных значений.

Tree98. Дан указатель P_1 на корень непустого дерева общего вида. В каждом наборе вершин-сестер изменить порядок следования их значений на противоположный, то есть поменять местами значения поля Data первой (левой) и последней (правой) сестры, второй и предпоследней сестры, и т. д.

Tree99. Дана строка *S*, содержащая описание непустого дерева общего вида в следующем формате:

<дерево> ::= <вершина> |
<вершина>(<список поддеревьев>)

<список поддеревьев> ::= <дерево> |
<дерево>,<список поддеревьев>

<вершина> ::= <цифра>

Например, «3(2,7(6,4,5),8(4(2,3),5(1)))» (пробелы отсутствуют, вершины-сестры перечисляются в порядке слева направо). Создать дерево общего вида по описанию, приведенному в *S*, и вывести указатель на его корень.

Tree100. Дан указатель P_1 на корень непустого дерева общего вида. Вывести строку с описанием исходного дерева в формате, приведенном в задании Tree99.

4.3.2. Указания

Tree72–75. Задания, связанные с формированием бинарного дерева по его строковому описанию (Tree72 и Tree74), и выполнением обратного действия: формированием строкового описания исходного дерева (Tree73 и Tree75). Решения Tree74 и Tree75 приводятся в п. 4.2.1; задания Tree72 и Tree73 решаются аналогично.

Tree76–85. Задания, связанные с разбором различных вариантов арифметических выражений. В Tree76–78 и Tree83 требуется по заданному арифметическому выражению сформировать дерево разбора выражения; эти задания решаются аналогично Tree74 (см. п. 4.2.1). В Tree79 и Tree84 требуется вычислить значение выражения, заданного деревом разбора, а в Tree80–82 и Tree85 — восстановить запись арифметического выражения в указанном формате по его дереву разбора. Эти задания решаются с помощью перебора вершин исходного дерева в требуемом порядке (ср. с решением Tree75 в п. 4.2.1).

Tree86–87. Решение Tree86 приводится в п. 4.2.2; «обратное» задание Tree87 решается с использованием рекурсивной функции, аналогичной функции CreateNode(*P*) из решения Tree86; следует лишь учитывать, что для получения адреса левого потомка непустой вершины с адресом *P* в дереве общего вида надо, как обычно, обратиться к его полю Left, а для получения адреса его следующего (то есть правого) потомка следует использовать выражение вида *P*^.Left^.Right, причем предварительно надо проверить, что указатель *P*^.Left не равен nil (равенство *P*^.Left = nil означает, что вершина *P* не имеет ни левого, ни правого потомка).

Tree88–91. Задания на анализ уровней дерева общего вида. Ср. Tree88 с Tree9, Tree89–90 с Tree10–11, Tree91 с Tree18.

Tree92–93. Задания на обход дерева общего вида в определенном порядке (ср. с аналогичными заданиями Tree12–14 для бинарного дерева). При реализации рекурсивной процедуры обхода используйте в качестве образца процедуру Tree из п. 4.2.2, модифицировав ее требуемым образом (заметим, что приведенный в п. 4.2.2 вариант процедуры Tree обеспечивает перебор вершин в префиксном порядке).

Tree94–98. Задания, связанные с перебором вершин-сестер для дерева общего вида. Используйте в качестве образца для организации подобного перебора рекурсивную процедуру Tree из п. 4.2.2. В Tree97–98 требуется в ходе этого перебора изменять значения для каждого набора вершин-сестер, поэтому в данных заданиях просмотр всех элементов связного списка сестер придется проводить дважды: первый раз для получения и обработки их значений (например, в Tree97 — для нахождения максимального значения), второй раз — для требуемого изменения значений вершин-сестер.

Tree99–100. Задания, связанные с формированием дерева общего вида по его строковому описанию (Tree99), и выполнением обратного действия: формированием строкового описания исходного дерева (Tree100). Ср. Tree99 с Tree74, Tree100 с Tree75 (решения Tree74 и Tree75 приводятся в п. 4.2.1). Примеры функции, обеспечивающей формирование дерева общего вида, и процедуры, выполняющей перебор вершин дерева общего вида, приводятся в п. 4.2.2 (функция CreateNode и процедура Tree соответственно).

4.4. Проектное задание

Выполните учебные задания группы Tree, указанные в вашем варианте проектного задания. Если вы не получили вариант проектного задания, то выполните задания из первого варианта.

ВАРИАНТ 1 (1) Разбор выражений: 78, 85, 84 (2) Деревья общего вида: 91, 93, 95, 87	**ВАРИАНТ 2** (1) Разбор выражений: 99, 74, 84 (2) Деревья общего вида: 91, 92, 94, 97
ВАРИАНТ 3 (1) Разбор выражений: 83, 81, 84 (2) Деревья общего вида: 90, 92, 96, 87	**ВАРИАНТ 4** (1) Разбор выражений: 76, 85, 79 (2) Деревья общего вида: 88, 92, 95, 98
ВАРИАНТ 5 (1) Разбор выражений: 76, 74, 84 (2) Деревья общего вида: 88, 93, 96, 87	**ВАРИАНТ 6** (1) Разбор выражений: 99, 100, 79 (2) Деревья общего вида: 88, 93, 96, 98

ВАРИАНТ 7 (1) Разбор выражений: 78, 82, 79 (2) Деревья общего вида: 89, 93, 94, 98	**ВАРИАНТ 8** (1) Разбор выражений: 77, 100, 84 (2) Деревья общего вида: 89, 92, 96, 87
ВАРИАНТ 9 (1) Разбор выражений: 72, 81, 84 (2) Деревья общего вида: 91, 93, 95, 97	**ВАРИАНТ 10** (1) Разбор выражений: 77, 82, 79 (2) Деревья общего вида: 90, 93, 95, 97
ВАРИАНТ 11 (1) Разбор выражений: 83, 80, 79 (2) Деревья общего вида: 89, 92, 94, 97	**ВАРИАНТ 12** (1) Разбор выражений: 72, 80, 79 (2) Деревья общего вида: 90, 92, 94, 98
ВАРИАНТ 13 (1) Разбор выражений: 72, 81, 84 (2) Деревья общего вида: 89, 92, 96, 98	**ВАРИАНТ 14** (1) Разбор выражений: 99, 81, 79 (2) Деревья общего вида: 91, 92, 95, 97
ВАРИАНТ 15 (1) Разбор выражений: 76, 82, 79 (2) Деревья общего вида: 88, 92, 95, 97	**ВАРИАНТ 16** (1) Разбор выражений: 77, 85, 84 (2) Деревья общего вида: 91, 93, 96, 87
ВАРИАНТ 17 (1) Разбор выражений: 77, 100, 79 (2) Деревья общего вида: 90, 92, 94, 87	**ВАРИАНТ 18** (1) Разбор выражений: 72, 74, 84 (2) Деревья общего вида: 89, 93, 95, 97
ВАРИАНТ 19 (1) Разбор выражений: 76, 85, 84 (2) Деревья общего вида: 90, 93, 95, 97	**ВАРИАНТ 20** (1) Разбор выражений: 78, 74, 84 (2) Деревья общего вида: 91, 92, 94, 98
ВАРИАНТ 21 (1) Разбор выражений: 83, 80, 79 (2) Деревья общего вида: 88, 93, 96, 98	**ВАРИАНТ 22** (1) Разбор выражений: 83, 80, 79 (2) Деревья общего вида: 89, 93, 96, 98
ВАРИАНТ 23 (1) Разбор выражений: 78, 82, 79 (2) Деревья общего вида: 88, 92, 94, 87	**ВАРИАНТ 24** (1) Разбор выражений: 99, 100, 84 (2) Деревья общего вида: 90, 93, 94, 87

4.5. Тест рубежного контроля

1. Укажите выражение, позволяющее получить число по его символьному представлению *C* (символ *C* может принимать значения от ′0′ до ′9′).			
(1)	Ord(*C*)	(2)	Ord(*C* – ′0′)
(3)	Ord(*C*) – Ord(′0′)	(4)	Ord(′0′) – Ord(*C*)
2. Дано числовое выражение 4 + 4 * 6 – 8 (как обычно, операция умножения * имеет более высокий приоритет, чем операции сложения и вычитания). Укажите, какой из приведенных вариантов соответствует записи этого выражения в префиксном бесскобочном формате.			
(1)	+ * 4 4 – 6 8	(2)	– + 4 * 4 6 8
(3)	+ * – 4 4 6 8	(4)	– * + 4 4 6 8
3. Дано числовое выражение 4 + 4 * (6 – 8). Укажите, какой из приведенных вариантов соответствует записи этого выражения в постфиксном бесскобочном формате.			
(1)	4 4 6 8 – * +	(2)	4 4 + 6 * 8 –
(3)	4 4 – 6 * 8 +	(4)	4 4 6 8 + * –
4. Укажите стандартный способ, используемый для представления дерева общего вида с помощью бинарного дерева (этот способ описан в пособии).			
(1)	Поле Right любой внутренней вершины *P* содержит указатель на первую дочернюю вершину, а поле Left — указатель на следующую сестру вершины *P*	(2)	Поле Left любой внутренней вершины *P* содержит указатель на первую дочернюю вершину, а поле Right — указатель на следующую сестру вершины *P*
(3)	Поле Right любой внутренней вершины *P* содержит указатель на последнюю дочернюю вершину, а поле Left — указатель на предыдущую сестру вершины *P*	(4)	Поле Left любой внутренней вершины *P* содержит указатель на последнюю дочернюю вершину, а поле Right — указатель на предыдущую сестру вершины *P*
5. Дано бинарное дерево, с помощью которого моделируется дерево общего вида. Требуется вывести вершины дерева общего вида в префиксном порядке. Укажите порядок, в котором надо обходить вершины исходного бинарного дерева.			
(1)	Инфиксный	(2)	Префиксный
(3)	Постфиксный	(4)	Порядок «вершина – правое поддерево – левое поддерево»

6. Дано бинарное дерево, с помощью которого моделируется дерево общего вида. Требуется вывести вершины дерева общего вида в постфиксном порядке. Укажите порядок, в котором надо обходить вершины исходного бинарного дерева.			
(1)	Инфиксный	(2)	Префиксный
(3)	Постфиксный	(4)	Порядок «правое поддерево – левое поддерево – вершина»

Приложение 1
Процедуры задачника Programming Taskbook

В настоящем приложении дается описание вспомогательных процедур, используемых при выполнении учебных заданий с применением задачника Programming Taskbook. Для того чтобы эти процедуры стали доступны в программе, необходимо подключить к ней с помощью оператора uses модуль PT4. Описаны только те процедуры ввода-вывода, которые требуются для выполнения заданий группы Tree. Описание всех процедур ввода-вывода, предусмотренных в задачнике Programming Taskbook, приводится, например, в [4]. Следует заметить, что при выполнении заданий с применением задачника Programming Taskbook в средах Pascal ABC и PascalABC.NET для ввода-вывода данных можно использовать стандартные процедуры read и write.

```
procedure Task(Name: string);
```

Процедура инициализирует задание с именем *Name*. Она должна вызываться в начале программы, выполняющей это задание (до вызова процедур ввода-вывода Get–Put). Если в программе, подключившей модуль PT4, не указана процедура Task, то при запуске программы будет выведено окно с сообщением «Не вызвана процедура Task с именем задания».

Имя задания *Name* должно включать имя темы и порядковый номер в пределах темы (например, «Tree2»). Регистр букв в имени темы может быть произвольным. Если указана неверная тема задания, то программа выведет сообщение об ошибке. Если указан недопустимый номер задания, то программа выведет сообщение, в котором будет указан диапазон допустимых номеров для данной темы.

Если после имени задания в параметре *Name* указан символ «?» (например, «Tree2?»), то программа будет работать в *демонстрационном режиме*, имеющем следующие особенности:

- даже если программа содержит решение задания, это решение не анализируется и информация в файл результатов не заносится;
- после отображения на экране окна задачника в разделе результатов сразу будет выбрана вкладка «Пример верного решения»;
- при одном запуске программы можно просмотреть несколько вариантов исходных и контрольных данных; для смены набора данных требуется нажать кнопку «Новые данные» или клавишу пробела;
- при одном запуске программы можно последовательно просмотреть все задания данной группы; для перехода к заданию с бóльшим номером требуется нажать кнопку «Следующее задание» или клавишу [Enter], а

для перехода к заданию с меньшим номером требуется нажать кнопку «Предыдущее задание» или клавишу [Backspace]. Задания перебираются циклически.

С помощью процедуры Task можно также генерировать тексты формулировок учебных заданий и дополнительные пояснения к заданиям в виде html-страницы. Для создания подобной страницы и ее немедленного отображения на экране (в html-браузере, используемом по умолчанию для данного компьютера) достаточно вызвать процедуру Task, указав в качестве ее параметра имя группы заданий или имя конкретного задания, дополненное символом «#», например, «Tree#» или «Tree2#». При указании имени группы генерируется текст всех заданий, включенных в эту группу. Процедуру Task с параметром, оканчивающимся символом «#», можно вызывать несколько раз, указывая различные имена групп или конкретных заданий; в результате созданная html-страница будет содержать тексты всех заданий, указанных при различных запусках процедуры Task (в том же порядке).

Если при каком-либо вызове будет указано неверное имя группы или неверный номер задания в пределах группы, то программа выведет сообщение об ошибке, и html-страница создана не будет. При успешной генерации html-страницы она сохраняется в файле со стандартным именем PT4Tasks.html в рабочем каталоге учащегося.

Если при первом вызове процедуры Task в параметре не указывается символ «#», то все последующие вызовы процедуры Task игнорируются. Если при первом вызове процедуры Task в параметре указывается символ «#», то игнорируются все ее последующие вызовы, не содержащие этот символ.

```
procedure GetN(var X: integer);
procedure GetP(var X: PNode);
procedure GetS(var X: string);
```

Процедуры обеспечивают ввод исходных данных в программу, выполняющую учебное задание. Они должны вызываться *после* вызова процедуры Task; в случае их вызова до вызова процедуры Task при запуске программы будет выведено сообщение об ошибке «В начале программы не вызвана процедура Task с именем задания».

Используемая процедура ввода должна соответствовать типу очередного элемента исходных данных; в противном случае выводится сообщение об ошибке «Неверно указан тип при вводе исходных данных» (такое сообщение будет выведено, например, если очередной элемент данных является указателем, а для его ввода используется процедура GetN).

При попытке ввести больше исходных данных, чем это предусмотрено в задании, выводится сообщение об ошибке «Попытка ввести лишние исходные данные». Если исходные данные, необходимые для решения задания, введены не полностью, то выводится сообщение «Введены не все требуемые исходные данные».

```
procedure PutN(X: integer);
procedure PutP(X: PNode);
procedure PutS(X: string);
```

Процедуры обеспечивают вывод на экран результирующих данных, найденных программой, и сравнение их с контрольными данными (то есть с правильным решением). Как и процедуры группы Get, эти процедуры должны вызываться после вызова процедуры Task; в противном случае при запуске программы будет выведено сообщение об ошибке «В начале программы не вызвана процедура Task с именем задания».

В отличие от процедур группы Get, в качестве параметров процедур группы Put можно указывать не только переменные, но и выражения (в частности, константы соответствующего типа). Используемая процедура должна соответствовать типу очередного элемента результирующих данных; в противном случае выводится сообщение об ошибке «Неверно указан тип при выводе результатов».

Как и в случае процедур группы Get, при вызовах процедур группы Put программа осуществляет контроль за соответствием количества требуемых и выведенных результирующих данных. Если программа выведет недостаточное или избыточное количество результирующих данных, то после проверки этих данных появится сообщение «Выведены не все результирующие данные» или, соответственно, «Попытка вывести лишние результирующие данные».

```
procedure Dispose(var P: PNode);
```

Данная процедура переопределяет стандартную процедуру Dispose для обеспечения контроля за действиями программы учащегося по освобождению динамической памяти. В отличие от стандартной процедуры Dispose, описанной в модуле System, данная процедура позволяет использовать в качестве параметра только указатели типа PNode.

Приложение 2
Контрольные вопросы по теме «Бинарные деревья»

1. Дайте определение дерева.
2. Дайте определение корня дерева, его листа и внутренней вершины.
3. Дайте определение уровня вершины и глубины дерева.
4. Дайте определение степени вершины и степени дерева.
5. Дайте определение бинарного дерева.
6. В чем состоит отличие бинарного дерева от упорядоченного дерева степени 2?
7. Опишите структуру, позволяющую хранить информацию о бинарном дереве, если с каждой его вершиной связывается некоторое целое число.
8. Приведите рекурсивный алгоритм, обеспечивающий нахождение количества вершин для данного бинарного дерева.
9. Опишите различные способы перебора вершин бинарного дерева и приведите названия для трех наиболее распространенных способов.
10. Дайте определение идеально сбалансированного дерева.
11. Опишите рекурсивный алгоритм построения идеально сбалансированного дерева с N вершинами.
12. Опишите рекурсивный алгоритм создания копии данного бинарного дерева.
13. Опишите рекурсивный алгоритм освобождения динамической памяти, выделенной для данного бинарного дерева.
14. Дайте определение полного дерева и опишите алгоритм дополнения данного бинарного дерева до полного дерева.
15. Дайте определение бинарного дерева с обратной связью.
16. Опишите алгоритм нахождения корня бинарного дерева с обратной связью, если дан указатель на одну из его вершин.
17. Опишите алгоритм нахождения ближайшего общего предка для двух различных вершин бинарного дерева с обратной связью.
18. Дайте определение бинарного дерева поиска, основанное на сравнении значений любой его вершины и значений вершин в ее левом и правом поддереве.
19. Дайте определение бинарного дерева поиска, основанное на инфиксном переборе вершин.
20. Дайте два варианта определения бинарного дерева поиска без повторяющихся элементов.
21. Опишите рекурсивный алгоритм поиска всех вершин с требуемым значением в бинарном дереве поиска.

22. Опишите рекурсивный алгоритм поиска вершины с требуемым значением в бинарном дереве поиска без повторяющихся элементов.
23. Опишите рекурсивный алгоритм добавления новой вершины с указанным значением в бинарное дерево поиска.
24. Опишите рекурсивный алгоритм добавления новой вершины с указанным значением в бинарное дерево поиска без повторяющихся элементов.
25. Опишите алгоритм сортировки деревом и оцените его быстродействие.
26. Опишите два варианта алгоритма удаления из бинарного дерева поиска вершины с двумя непосредственными потомками, в результате которого дерево останется деревом поиска.
27. Приведите примеры строковых описаний бинарных деревьев.
28. Дайте определение префиксного и постфиксного бесскобочного формата арифметического выражения.
29. Дайте определение дерева разбора арифметического выражения.
30. Опишите рекурсивный алгоритм вычисления значения арифметического выражения по его дереву разбора.
31. Опишите способ представления дерева общего вида, использующий структуру бинарного дерева.
32. Опишите рекурсивный алгоритм определения глубины дерева общего вида, представленного в виде бинарного дерева.
33. Дайте определение вершин-сестер и опишите итерационный алгоритм перебора всех вершин-сестер, являющихся непосредственными потомками данной вершины дерева общего вида.
34. Дано бинарное дерево, с помощью которого моделируется дерево общего вида. Опишите префиксный способ обхода дерева общего вида и определите, какому способу обхода «обычного» бинарного дерева он соответствует.
35. Дано бинарное дерево, с помощью которого моделируется дерево общего вида. Опишите постфиксный способ обхода дерева общего вида и определите, какому способу обхода «обычного» бинарного дерева он соответствует.

Литература

1. *Абрамян М. Э.* 1000 задач по программированию. Часть I: Скалярные типы данных, управляющие операторы, процедуры и функции. — Ростов н/Д.: УПЛ РГУ, 2004. — 43 с. (http://open-edu.sfedu.ru/files/files/pubs/progr-x1.zip)
2. *Абрамян М. Э.* 1000 задач по программированию. Часть II: Минимумы и максимумы, одномерные и двумерные массивы, символы и строки, двоичные файлы. — Ростов н/Д.: УПЛ РГУ, 2004. — 42 с. (http://open-edu.sfedu.ru/files/files/pubs/progr-x2.zip)
3. *Абрамян М. Э.* 1000 задач по программированию. Часть III: Текстовые файлы, составные типы данных в процедурах и функциях, рекурсия, указатели и динамические структуры. — Ростов н/Д.: УПЛ РГУ, 2004. — 43 с. (http://open-edu.sfedu.ru/files/files/pubs/progr-x3.zip)
4. *Абрамян М. Э.* Практикум по программированию на языке Паскаль. 6-е изд., перераб. и доп. — Ростов н/Д.: ЦВВР, 2008. — 227 с.
5. *Вирт Н.* Алгоритмы и структуры данных. — М.: Мир, 1989. — 360 с.
6. *Михалкович С. С.* Основы программирования: Динамические массивы. Списки. Ассоциативные массивы. Деревья. Хеш-таблицы. — Ростов н/Д.: УПЛ ЮФУ, 2007. — 48 с.

Содержание