Камчатский государственный технический университет

А.Ю. Алексеев, С.А. Ивановский, Д.В. Куликов

ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

Учебно-методическое пособие для студентов специальностей 220400 "Программное обеспечение вычислительной техники и автоматизированных систем" и 210100 «Управление и информатика в технических системах»

Петропавловск-Камчатский
2004

УДК 681.3
ББК 32.973
А47

Рецензенты:

В.Б. Макулов,
кандидат технических наук, доцент (ВНЦ ГОИ им. С.И. Вавилова)

кафедра цифровой вычислительной техники и информатики
Санкт-Петербургского государственного университета телекоммуникаций

Алексеев А.Ю., Ивановский С.А., Куликов Д.В.

А47 Динамические структуры данных: Учебно-методическое пособие. – Петропавловск-Камчатский: КамчатГТУ, 2004. – 68 с.

Содержит основные сведения, приемы и упражнения по программированию динамических структур данных: списков, стеков, очередей и деревьев. Ориентировано на использование при проведении лабораторных и выполнении курсовых работ в учебных курсах по программированию и структурам данных.

Предназначено для студентов специальностей 220400 "Программное обеспечение вычислительной техники и автоматизированных систем" и 210100 «Управление и информатика в технических системах».

Рекомендовано к изданию решением учебно-методического совета КамчатГТУ (протокол № 6 от 20 февраля 2004 г.).

В авторской редакции
Технический редактор Бабух Е.Е.
Набор текста Ивановский С.А.
Верстка Ивановский С.А., Бабух Е.Е.
Оригинал-макет Бабух Е.Е.

Лицензия ИД № 02187 от 30.06.00 г. Подписано в печать 12.03.2004 г.
Формат 61*86/16. Печать офсетная. Гарнитура Times New Roman
Авт. л. 4,78. Уч.-изд. л. 4,85. Усл. печ. л. 4,37
Тираж 60 экз. Заказ № 218

Отпечатано полиграфическим участком РИО КамчатГТУ
683003, г. Петропавловск-Камчатский, ул. Ключевская, 35

Введение

При обучении программированию особую трудность вызывает работа с динамическими структурами данных. Как правило, такие структуры данных являются "самодельными", и программист сам должен управлять их размещением в памяти машины: порождать, модифицировать и уничтожать их в процессе выполнения программы. Все эти операции в конечном итоге реализуются на встроенных в вычислительную машину (ВМ) структурах данных (на базе модели памяти данной ВМ). При этом встроенные структуры данных отражают специфику организации ВМ (как совокупности программно-аппаратных средств), в то время как "самодельные" структуры данных отражают специфику решаемой задачи. Разрешение этого естественного противоречия может основываться на абстрагировании от деталей реализации динамических структур данных, на определении этих структур данных через базовые операции над ними с указанием их характерных свойств.

В рамках такого подхода в учебном пособии рассматриваются типовые динамические структуры данных, используемые при решении многих программистских задач: списки, стеки и очереди, деревья и леса. Для каждой структуры даются основные определения, формальная спецификация, представление и реализация на языке Паскаль (в версии Турбо-Паскаль). Затем следует набор упражнений для программирования.

При формулировке заданий для упражнений отражена лишь алгоритмическая суть задачи. Подразумевается, что должна быть написана программа, содержащая основной алгоритм (вычислительную часть) и модуль или модули (в смысле языка Турбо-Паскаль) для работы с используемыми структурами данных. Кроме того, программа должна выполнять тестирующие функции по отношению к своей вычислительной части: должен быть предусмотрен удобный при тестировании ввод исходных данных, вывод (визуализация) промежуточных данных и результатов в форме, удобной для анализа работы основного алгоритма. В случае динамических структур данных выполнение этих не вполне формальных требований может составлять определенную проблему, которую следует решать при консультации с преподавателем**.**

1. СПИСКИ

1.1. Линейный однонаправленный список

Линейный список представляет собой способ организации последовательности однотипных элементов, при котором, каждый элемент, кроме первого, имеет одного предшественника (предыдущий элемент) и каждый элемент, кроме последнего, имеет одного преемника (следующий элемент). Доступ к каждому элементу списка можно получить, последовательно продвигаясь по списку от элемента к элементу. Если это продвижение осуществляется только в одном направлении, то говорят о *линейном однонаправленном* списке (Л1–списке). Если можно перемещаться по списку, как в направлении его конца, так и к его началу, то говорят о *линейном двунаправленном* списке (Л2–списке). Схематично линейный список можно представлять, например, так, как это изображено на рис. 1.1.

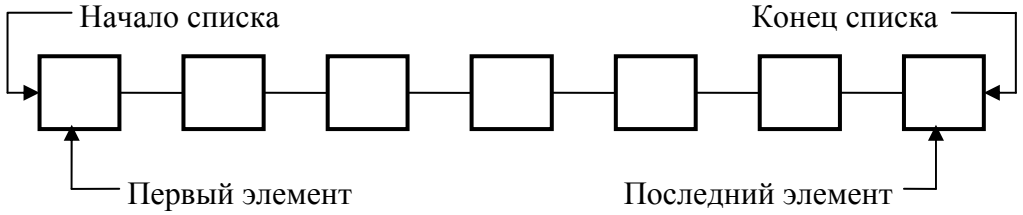


Рис. 1.1. Схематичное (модельное) представление списка

Операции добавления, удаления или замены элементов списка могут производиться в любом месте списка по мере продвижения по нему.

Простейший способ реализации линейного списка основан на его представлении одномерным массивом так, что соседние элементы списка записаны в соседние элементы массива (так называемая *непрерывная* реализация на базе вектора). Очевидным недостатком такой реализации является необходимость сдвига элементов массива при выполнении операций вставки и удаления элемента списка. Иначе говоря, операции удаления и вставки при такой реализации являются *массовыми*, т.е. требующими выполнения элементарных операций в количестве, в среднем пропорциональном числу элементов списка.

От этого недостатка свободно так называемое цепное (или ссылочное) представление Л1–списка. Здесь каждый элемент списка представляется звеном в цепи, состоящим из информационного поля *Info*, содержащего собственно элемент списка, и поля *Next*, куда записывается ссылка на следующий элемент.

Схематично отдельное звено списка изображено на рис. 1.2,а. Пустую ссылку обычно обозначают символом **nil** и изображают, например, как на рис. 1.2,б. Тогда последний элемент списка будет изображен так, как показано на рис. 1.2,в. В качестве примера на рис. 1.3 изображен в этих обозначениях список из элементов ***a,b,c,d***.

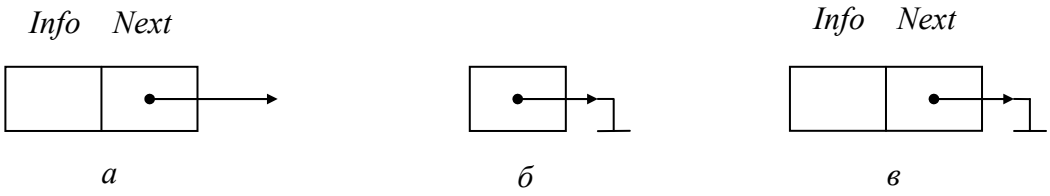


Рис. 1.2. Изображения: а - отдельного звена списка;
б - пустой ссылки; в - последнего звена в списке

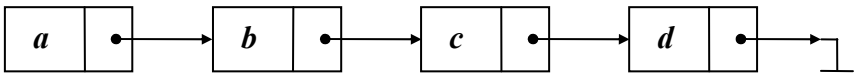


Рис. 1.3. Пример изображения Л1-списка

На Паскале такое цепное представление списка может быть реализовано, например [xx], с использованием ссылочных переменных. Соответствующие типы данных можно описать в виде

```
type
El = ... ;                    { базовый тип для элементов списка }
Link = ^Node;                 { ссылка на звено в цепи            }
Node = record                 { звено  цепи  ( списка ) :         }
       Info : El;             { содержимое (элемент списка)       }
       Next : Link ;          { ссылка на следующее звено         }
       end  {Node} ;
```

Длина списка в этом случае ограничивается только размерами доступной динамической памяти, в которой размещаются элементы списка (звенья цепи), связанные ссылками.

Для работы со списком удобно описать дополнительную переменную типа *Link* (например, ***var*** *s* : *Link*) и использовать ее как указатель на начало списка (рис. 1.4). Если список пуст, то *s* = **nil**.

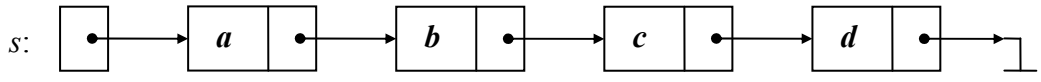


Рис. 1.4. Пример изображения Л1-списка с указателем на начало

Другим возможным способом реализации цепного представления Л1–списка на Паскале является ссылочная реализация на базе вектора. В качестве ссылок в этом случае используются номера элементов вектора, предназначенных для хранения звеньев цепи (элементов списка). Как ссылочная реализация Л1–списка в динамической памяти, так и реализация на базе вектора подробно рассмотрены далее в 1.3, 1.4.

1.2. Л1–список как абстрактный тип данных

Рассмотрим Л1–список как *абстрактный тип данных* (АТД), определяя его через класс базовых операций, которые (и только они) выполняются над Л1–списком. Все остальные действия над Л1–списком реализуются на основе этих операций.

Перед тем, как задать функциональную спецификацию Л1–списка, полезно рассмотреть его неформальную модель. Будем считать, что состояние списка задается не только перечислением набора элементов, но и дополнительно указанием одного из них в качестве текущего (очередного). Относительно этого элемента будет определяться семантика базовых операций. Заданием текущего элемента список разделяется на две части: от начала до текущего элемента (пройденная часть) и от текущего элемента (включая его) до конца списка (рис. 1.5). К элементам пройденной части можно получить доступ, только начиная просмотр списка сначала. В непройденной части доступны все элементы поочередно, начиная с текущего элемента.

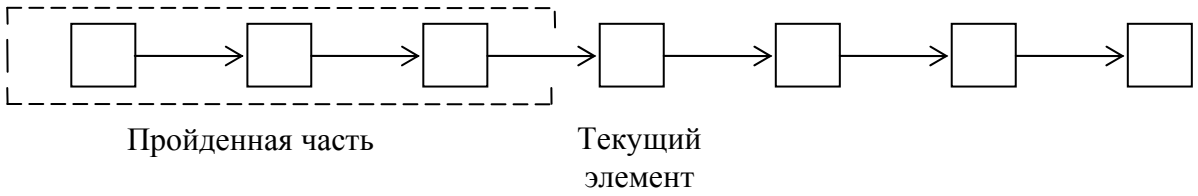


Рис. 1.5. Модель Л1-списка

Определим Л1-список, задав набор базовых операций с помощью функциональной спецификации, приведенной на рис.1.6. Используем обозначения: *L_list* – тип линейного списка, *El* – тип элементов списка, а знаком × обозначим декартово произведение множеств.

Пусть $w = [\, x_1, x_2, ..., x_n \,]$ – последовательность элементов типа *El*. Для формального описания семантики базовых операций используем две вспомогательные функции над непустыми последовательностями: $First(w) = x_1$ – первый элемент последовательности, $Rest(\, w \,) = [\, x_2, x_3, ..., x_n \,]$ – остаток последовательности. Операция *Prefix* добавляет элемент в начало последовательности: $Prefix\ (\, x_0, w \,) = = [\, x_0, x_1, x_2, ..., x_n \,]$, $Prefix(\, First(\, w \,), Rest(\, w \,)) = w$, а операция *Postfix* дописывает элемент в конец последовательности: $Postfix(\, w, x) = [\, x_1, x_2, ..., x_n, x \,]$. Операцию *Postfix* будем также записывать в инфиксной форме, используя знак * для обозначения этой операции: $[\, x_1, x_2, ..., x_n \,] * x = [\, x_1, x_2, ..., x_n, x \,]$. Пустую последовательность [ ] обозначим символом Δ. Тогда $Postfix(\Delta, x) = \Delta * x = Prefix(\, x, \Delta \,) = [\, x \,]$, $First(\, [\, x \,] \,) = x$, $Rest(\, [\, x \,] \,) = \Delta$.

1.*Create*:	$\rightarrow L\_list$	Начать работу(создать пустой список)
2.*Null*:	$L\_list \rightarrow Boolean$	Список пуст?
3.*Emply*:	$L\_list \rightarrow L\_list$	Сделать список пустым
4.*GoBOL*:	$L\_list \rightarrow L\_list$	Встать в начало списка (сделать текущим первый элемент, тогда пройденная часть списка пуста)
5.*EOList*:	$L\_list \rightarrow$ Boolean	Конец списка (непройденная часть списка пуста)?
6.*GoNext*:	$L\_list \rightarrow L\_list$	Перейти к следующему элементу; если текущим был последний, то новое состояние списка – *EOList*; отказ: в состояниях *Null*, *EOList*
7.*GetEl*:	$L\_list \rightarrow El$	Получить текущий элемент; отказ: в состояниях *Null*, *EOList*
8.*Insert*:	$L\_list \times El \rightarrow L\_list$	Добавить элемент перед текущим (в состоянии *EOList* добавить элемент в конец списка),сделать текущим новый элемент
9.*Replace*:	$L\_list \times El \rightarrow$ L_list	Заменить текущий элемент; отказ: в состояниях *Null*, *EOList*
10.*Delete*:	$L\_list \rightarrow L\_list$	Удалить текущий элемент, следующий сделать текущим; отказ: в состояниях *Null*, *EOList*
11.*Destroy*:	*L-list* $\rightarrow$	Закончить работу

Рис.1.6. Функциональная спецификация Л1-списка

Пусть *Left*(*s*) – последовательность, соответствующая пройденной части списка *s*, *Right*(*s*) – непройденной его части, начиная с текущего элемента, а *Seq*(*s*) – всему списку в целом. Состояние *BOList* (начало списка) будем фиксировать, когда текущим элементом является первый элемент списка, т.е. *Left*(*s*) = Δ, *Right*(*s*) = *Seq*(*s*). Состояние *EOList* (конец списка) фиксируется, когда *Left*(*s*) = *Seq*(*s*) и *Right*(*s*) = Δ. Текущий элемент при этом не определен, а операция *Insert* имеет особый смысл (добавить элемент в конец списка).

Семантика операций 1,2,3,5 ясна из рис.1.6. Для остальных базовых операций зададим их семантику в виде свойств (троек Хоара) {*Pred*} *Op* {*Post*}, где *Pred* – предусловие, *Op* – операция, *Post* – постусловие. Используем обозначения: *s*, s_0 – списки типа *L_list*; *L*, *R*, *R*1 – последовательности; *e* – переменная типа *El*.

*Op*4: { $s_0 = s$ }
GoBOL(*s*)
{ *Left*(*s*) = Δ **&** *Right*(*s*) = *Seq*(s_0) } ;

*Op*6: { **not** *Null*(*s*) **&** **not** *EOList*(*s*) **&** *L* = *Left*(*s*) **&** *R* = *Right*(*s*) }
GoNext(*s*)
{ *Left*(*s*) = *L*First*(*R*) **&** *Right*(*s*) = *Rest*(*R*) } ;

*Op*7: { $s_0 = s$ **&** **not** *Null*(*s*) **&** **not** *EOList*(*s*) **&** *R* = *Right*(*s*) }
GetEl(*s*, *e*)
{ *e* = *First*(*R*) **&** $s = s_0$ };
*Op*8: { *L* = *Left*(*s*) **&** *R* = *Right*(*s*) }
Insert(*s*, *e*)
{ *Left*(*s*) = *L* **&** *Right*(*s*) = *Prefix*(*e*, *R*) };
*Op*9: { **not** *Null*(*s*) **&** **not** *EOList*(*s*) **&** *L* = *Left*(*s*) **&** *R* = *Right*(*s*) **&** *R*1 = *Rest*(*R*) }
Replace(*s*, *e*)
{ *Left*(*s*) = *L* & *Right*(*s*) = *Prefix*(*e*, *R*1) };
*Op*10: { **not** *Null*(*s*) **&** **not** *EOList*(*s*) **&** *L* = *Left*(*s*) **&** *R* = *Right*(*s*) **&** *R*1 = *Rest*(*R*) }
Delete(*s*)
{ *Left*(*s*) = *L* **&** *Right*(*s*) = *R*1 }.

После выполнения операции *Destroy*(*s*) память, отведенная под список *s*, освобождается, а значение переменной *s* становится неопределенным.

После того, как задана функциональная спецификация, перейдем к рассмотрению особенностей реализации линейного списка на основе различных базовых структур данных.

1.3. Ссылочная реализация Л1–списка в динамической памяти

Основная идея реализации Л1–списка ясна из рис. 1.7, *а÷г.* Список представляется агрегатом из собственно цепочки звеньев списка и дополнительной записи (формуляра списка) из трех полей ссылок на первый, текущий и предшествующий текущему элементы. Содержимое списка и его текущее состояние полностью определяются этими (и только этими) переменными. Переменной, идентифицирующей список, является указатель на формуляр списка (переменная s на рис.1.7,*а÷г*).

Память под формуляр списка выделяется при выполнении операции *Create*. Выполнение операции *Insert* приводит к размещению очередного элемента списка в памяти, освободить которую можно, например, удалив этот элемент с помощью операции *Delete*. При выполнении операции *Empty* освобождается вся память, занятая под элементы списка, но сохраняется формуляр списка. Выполнение операции *Destroy* равносильно выполнению операции *Empty* и, кроме того, приводит к освобождению памяти, отведенной под формуляр списка; значение идентифицирующей список переменной становится при этом неопределенным.

Особые состояния списка: *Null* – пустой список, *BOList* – начало списка и *EOList* – конец списка, определяются как

Null $\leftrightarrow$ (*Head* = **nil**) **&** (*PredCur* = **nil**) **&** (*Cur* = **nil**);
BOList **&** **not** *Null* $\leftrightarrow$ (*Head* = *Cur* $\neq$ **nil**) **&** (*PredCur* = **nil**);
EOList **&** **not** *Null* $\leftrightarrow$ (*Head* $\neq$ **nil**) **&** (*PredCur* $\neq$ **nil**) **&** (*Cur* = **nil**) **&**
(*PredCur*^.*link* = **nil**)

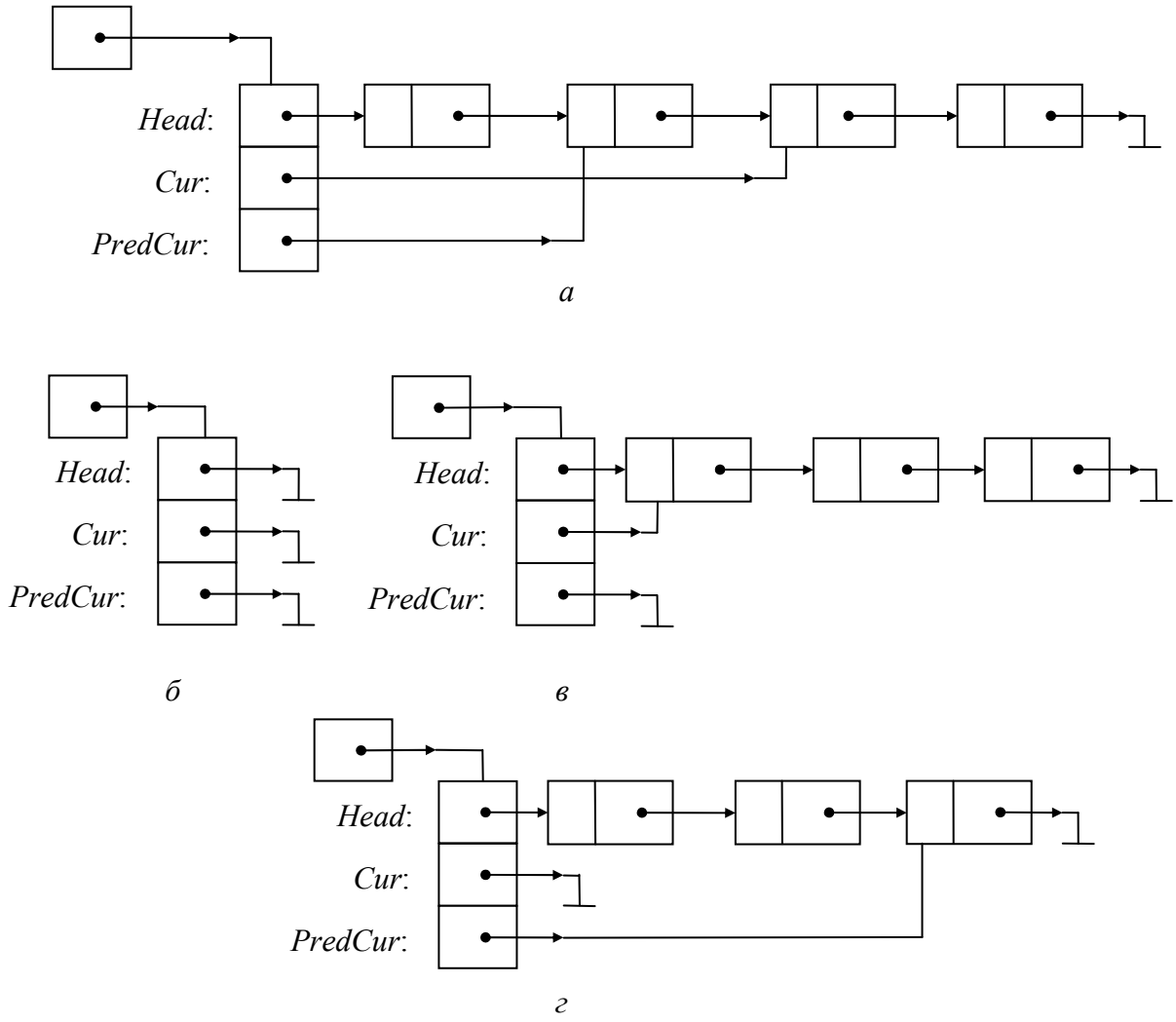


Рис. 1.7. Представление Л1-списка при ссылочной реализации в динамической памяти: а - список из четырех элементов; б - пустой список; в - состояние BOList списка из трех элементов; г - состояние EOList списка из трех элементов

На Турбо-Паскале Л1-список может быть реализован в виде модуля *L*1*List*, макет которого представлен на рис. 1.8. В макете полностью приведена "видимая" часть (секция **interface**) модуля, в которой описаны необходимые типы данных и приведены заголовки процедур и функций в соответствии с функциональной спецификацией Л1–списка. Описанный в модуле тип данных *L*1*_list* соответствует типу данных *L_list* функциональной спецификации.

В качестве самостоятельного задания предлагается написать секцию **implementation** модуля *L*1*List*, в которой должны быть сосредоточены блоки процедур и функций.

Для того, чтобы воспользоваться модулем *L*1*List*, необходимо иметь модуль *Global* с описанием типа *El* и главную программу или модуль, где производится собственно обработка списка. Макеты этих программных единиц (модуля *Global* и главной программы) представлены на рис. 1.9.

```
Unit L1List;
    {   Реализация L1–списка на Турбо-Паскале          }
Interface
Uses Global;    { модуль, в котором описан тип элементов списка El  }
{================================================================= }
 {Ссылочное представление в динамической памяти }
 type        { El – базовый тип для  элементов списка (описан в Global)}
      Link = ^ Node;          {ссылка на звено в цепи       }
RepList = record              {формуляр ("представитель") списка}
Head, Cur, PredCur : Link;
end  { RepList };
= record            { звено цепи ( списка ):              }
      Info : El;        { - содержимое (элемент списка)   }
      Next : Link ;   { - ссылка на следующее звено       }
end  { Node };
      L1_list = ^ RepList;  {Л1–список}
{================================================================}
{1}function Create : L1_list;                  {Начать работу (создать пустой список
)}
{2}function Null ( l : L1_list ) : Boolean;          {Список пуст ?}
{3}procedure Empty ( l : L1_list ) ;                 {Сделать список пустым  }
{4}procedure GoBOL ( l : L1_list ) ;                 {Встать в начало списка  }
{5}function EOList ( l : L1_list ) : Boolean;        {Конец списка  }
{6}procedure GoNext ( l : L1_list ) ;               {Перейти к следующему элементу
списка;
                        отказ: в состояниях Null, EOList}
{7}procedure GetEl ( l : L1_list; var e : El );      {Получить текущий элемент
списка;                                                  отказ: в состояниях
Null, EOList}
{8}procedure Insert ( l : L1_list; e : El );         {Добавить элемент перед теку-
щим,
                        текущим сделать новый элемент}
{9}procedure Replace ( l : L1_list; e : El );        {Заменить текущий элемент спи-
ска;
                        отказ: в состояниях Null, EOList}
{10}procedure Delete ( l : L1_list ) ;               {Удалить текущий элемент списка,
                        сделать текущим прежний следующий;
                        отказ: в состояниях Null, EOList}
{11} procedure Destroy ( var l : L1_list );     {Закончить работу
                                (ликвидировать список)}
{================================================================}
Implementation
...  {Здесь размещаются блоки процедур и функций }
end {L1List}.
```

Рис. 1.8. Макет модуля L1List

Unit *Global*; {Здесь задается тип элементов } {Л1–списка, реализованного } { модулем *L*1*List* } **Interface** **type** *El* = *Char*; { например } **Implementation** **end** {*Global*}.	**program** *Main*; **Uses** *L*1*List*,*Global*; {Программа работы со списками,} {использующая модуль *L*1*List* } ... **var** s : *L*1*_list*; ... **begin** ... **end** {*Main*}.

Рис. 1.9. Макеты программных единиц, связанных с модулем L1List: слева - модуля Global, справа - главной программы

Таким образом, для работы с Л1–списком достаточно описать переменную типа *L*1*_list* и воспользоваться набором базовых операций, реализованных в модуле *L*1*List*.

1.4. Ссылочная реализация ограниченного Л1–списка на базе вектора

В ряде случаев можно реализовать Л1–список, опираясь на линейную (векторную) модель памяти. Такая реализация, очевидно, целесообразна на языках типа Бейсик, Фортран и т.п. Даже в тех случаях, когда язык программирования (Паскаль, Си, Ассемблер) поддерживает модель динамической памяти, векторная реализация может оказаться привлекательной тем, что позволяет программисту самому управлять распределением памяти, не полагаясь на исполняющую систему, частое обращение к которой может к тому же потребовать значительных затрат времени при исполнении программы.

Итак, пусть память, отводимая для хранения списка, представляется одномерным массивом (вектором) записей. Ссылкой будет являться номер элемента вектора. Поскольку размер массива в ходе обработки списка не изменяется, то и максимальная длина списка ограничена этим размером. В этом случае говорят об ограниченном Л1–списке и добавляют к базовым операцию *Not_full* , проверяющую массив на наличие свободного места, необходимого для размещения очередного элемента списка: *Not_full*: *L_list* → *Boolean*.

Основная идея реализации ограниченного Л1–списка на базе вектора ясна из рис.1.10–1.12. На рис.1.10 приведен пример размещения списка *s* из элементов ***a, b, c, d*** в массиве *Memory* типа *MemoryForLists* при *MemMaxSize* = 12. Нижние линии на рисунке обозначают связи списка *s*, а верхние линии связывают в список свободной памяти не занятые элементами списка *s* элементы массива. Соединение свободных элементов массива в линейный список значительно упрощает реализацию. Нулевой элемент массива используется для хранения ссылки на первый элемент списка свободной памяти. При этом на базе одного вектора (массива типа *MemoryForLists*) могут быть реализованы несколько Л1–списков, ограниченных в совокупности.

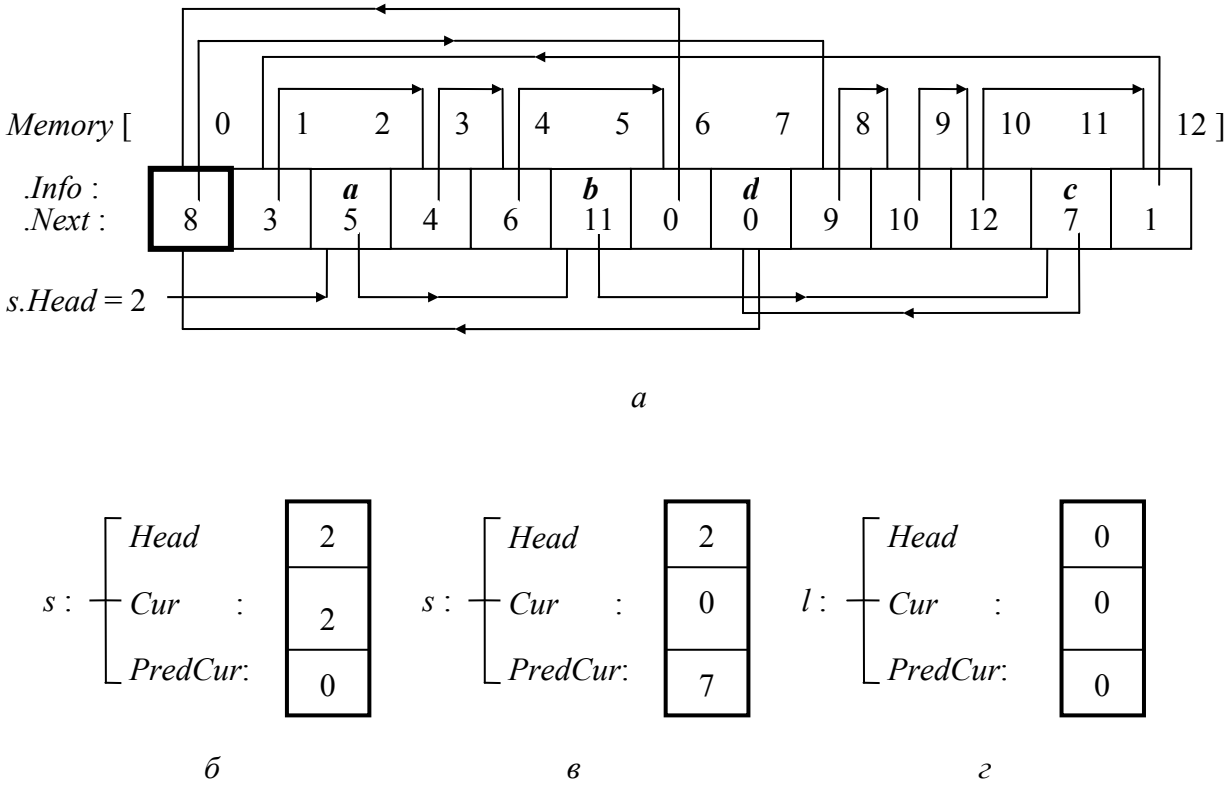


Рис. 1.10. Представление ограниченного Л1–списка при ссылочной реализации на базе вектора: а – размещение списка s в массиве Memory; б,в – формуляры списка s в состояниях BOList и EOList соответственно; г – формуляр пустого списка l

На рис. 1.11 изображен макет секции **Interface** модуля *L*1*ListV*, реализующего на Турбо–Паскале типы данных и базовые операции Л1–списка на базе вектора. Типу *L_list* функциональной спецификации здесь соответствует тип *L*1*_listV*, а список идентифицируется записью типа *RepList.* При необходимости можно добиться большего соответствия с предыдущей реализацией, представляя список указателем на такую запись: **type** *L*1*_listV* = ^ *RepList.*

Заметим, что описание массива *Memory*, предназначенного для хранения элементов списков, равно как и описание соответствующего типа данных *MemoryForLists*, помещено в секцию **Implementation** модуля *L*1*ListV* (рис. 1.12). Начальное состояние списка свободной памяти задается в секции инициализации и устанавливается в процессе загрузки модуля *L*1*ListV*, а в модуле *Global* (рис.1. 13) теперь задается не только тип *El* элементов списка, но и размер *MemMaxSize* поля памяти, отведенного для размещения списков.

Обратим внимание на то, что здесь некоторые операции могут быть реализованы более эффективно, чем в случае с динамической памятью. Например, при выполнении операций *Empty*(*s*) и *Destroy*(*s*) необходимо освободить память, занимаемую списком *s*. Освобождение динамической памяти, занятой под каждый элемент списка, производится с помощью вызова предназначенной для этого процедуры (*dispose* в языке Паскаль). Освобождение же занятой под список части массива может быть проделано непосредственным изменением ссылок: последний элемент "опустошаемого" списка *s* должен теперь ссылаться на прежний первый элемент списка свободной памяти, а пер-

вый элемент списка *s* становится первым элементом списка свободной памяти (в результате "опустошаемый" список сцепляется со списком свободной памяти).

```
Unit L1ListV;  { Реализация ограниченного L1-списка на Турбо-Паскале }
Interface
Uses Global;{модуль, в котором описан тип элементов списка El  }
{=================================================================}
{                Ссылочное представление на базе вектора          }
{const MemMaxSize = максимальный размер "рабочей" памяти,
                  описание дано в модуле Global }
type
        Index = 0..MemMaxSize;
        Link = Index;  { ссылка на звено цепи }
          формуляр ("представитель") списка: }
RepList = record
Head, Cur, PredCur : Link;
end  {RepList} ;
      L1_listV = RepList;       {Л1–список=формуляр списка   }
{================================================================= }
{1} procedure Create  ( var l: L1_listV );
{ начать работу ( создать пустой список ) }
{2} function Null ( l : L1_listV ) : Boolean;      { список пуст }
{3} procedure Empty ( var l : L1_listV ) ;         { сделать список пустым }
{4} procedure GoBOL ( var l : L1_listV ) ;         { встать в начало списка }
{5} function EOList ( l : L1_listV ) : Boolean;  { конец списка }
{6}procedure GoNext ( var l : L1_listV ) ;
{ перейти к следующему элементу списка; отказ: в состояниях Null, EOList }
{7} procedure GetEl ( l : L1_listV; var e : El );
{ получить текущий элемент списка;  отказ: в состояниях Null, EOList  }
{8} procedure Insert ( var l : L1_listV; e : El );
{ добавить элемент перед текущим, текущим сделать новый элемент }
{9} procedure Replace ( l : L1_listV; e : El );
{ заменить текущий элемент списка;  отказ: в состояниях Null, EOList }
{10} procedure Delete ( var l : L1_listV ) ;
{ удалить текущий элемент списка, сделать текущим прежний следующий; }
{ отказ : в состояниях Null, EOList }
{11} procedure Destroy ( var l : L1_listV );
{ закончить работу (ликвидировать список) }
{12} function Not_full : Boolean;
{ в "рабочей" памяти есть место  для размещения элемента списка }
```

*Рис. 1.11. Секция **Interface** модуля L1ListV*

```
Implementation
Const NilLst = 0 ;  {nil для списков, в том числе   для списка свободной памяти }
type
ord              {звено цепи ( списка ):              }
 Info : El;      {- содержимое (элемент списка) }
 Next : Link;    { - ссылка на следующее звено   }
end  { Node } ;
MemoryForLists = array [ Index ] of Node;
{рабочий массив для размещения списков }
      var Memory : MemoryForLists;
i : Index;
...    {  На месте ... размещаются блоки процедур и функций }
begin
{ инициализация списка свободной памяти в массиве Memory }
Memory [ MemMaxSize ] . Next := NilLst;
for i := 1 to MemMaxSize do
Memory [ i-1 ] . Next :=  i ;
end {L1ListV}.
```

*Рис. 1.12. Макет секции **Implementation** модуля L1ListV*

```
Unit Global;
   {Здесь задаются тип элементов Л1–списка  и максимальный размер "рабочей" памяти }
Interface
const
MaxSize =          {  максимальный размер  }
00;   {например } {"рабочей" памяти          }
type El = Char; {например}
Implementation
end { Global }.
```

Рис. 1.13. Модуль Global

1.5. Линейный двунаправленный список

Как уже отмечено в 1.1, Л2–список представляет собой такой линейный список, по которому можно перемещаться от текущего элемента как в направлении конца списка, так и в направлении его начала. Функциональная спецификация Л2–списка может быть получена расширением приведенной на рис. 1.6 функциональной спецификации Л1–списка, например, путем задания трех дополнительных базовых операций (рис. 1.14).

Семантику дополнительных базовых операций определим с помощью троек Хоара аналогично тому, как это было сделано в 1.2 для Л1–списка. Пусть по-прежнему $w = [\ x_1, x_2, ..., x_{n-1}, x_n\ ]$ – непустая последовательность эле

ментов типа *El*, и пусть *Last*(*w*) = x_n – последний элемент последовательности *w*, а *Lead*(*w*) = [x_1, x_2, ..., x_{n-1}], так что *Postfix*(*Lead*(*w*), *Last*(*w*)) = *w*. Тогда

*Op*12: $\{s_0 = s\}$
GoEOL(*s*)
{ *Left*(*s*) = *Seq*(s_0) **&** *Right*(*s*) = Δ }

*Op*13: { $s_0 = s$ **&** *L* = *Left*(*s*) }
b := *BOList*(*s*)
{ $s_0 = s$ **&** *b* = (*L* = Δ) }

*Op*14: {**not** *Null*(*s*) **&** **not** *BOList*(*s*) **&** *L* = *Left*(*s*) **&** *R* = *Right*(*s*) }
GoPrev(*s*)
{*Left*(*s*) = *Lead*(*L*) **&** *Right*(*s*) = *Prefix*(*Last*(*L*) , *R*) }

12. *GoEOL* : *L_list* → *L_list*	Встать в конец списка
13. *BOList* : *L_list* → *Boolean*	Начало списка (текущим является первый элемент списка)
14. *GoPrev* : *L_list* → *L_list*	Перейти к предыдущему элементу; *отказ*: в состояниях *Null* и*BOList*

Рис. 1.14. Дополнительные базовые операции Л2–списка

На Паскале реализация цепного представления Л2–списка в связанной памяти может быть выполнена, например, с использованием следующих типов данных:

```
type El   = ... ;                    { базовый тип для элементов списка}
     Link = ^ Node;                  { ссылка на звено цепи}
     Node = record                   { звено  цепи  ( списка ) :          }
            Info : El;               { - содержимое (элемент списка)      }
            Next : Link;             { - ссылка  на следующее звено       }
            Prev : Link;             { - ссылка на предшествующее звено}
        end  { Node } ;
```

По сравнению с Л1–списком это означает добавление поля *Prev* ссылки на предшествующее звено в каждое звено цепи. Формуляр списка здесь может иметь тот же вид, что и при реализации *L*1–списка. В этом случае в поле *Prev* первого элемента списка следует хранить ссылку на последний элемент. В качестве формуляра списка можно использовать также запись вида

```
type
     RepList = record          {формуляр ("представитель") списка }
                 Head , Cur, Last : Link;
               end  { RepList } ;
```

в поле *Last* которой будет храниться ссылка на последний элемент списка. Особые случаи при этом определяются как

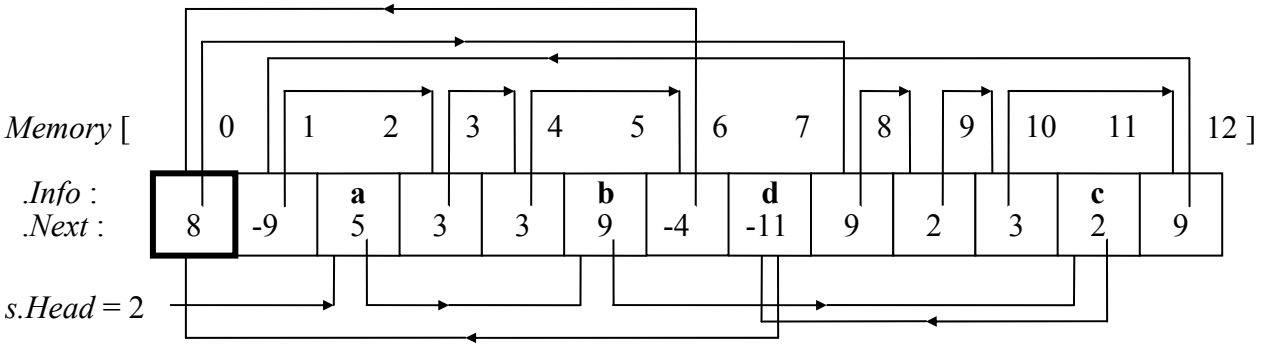
Null ↔ (*Head* = **nil**) **&** (*Cur* = **nil**) **&** (*Last* = **nil**);
BOList **&** **not** *Null* ↔ (*Head* = *Cur* ≠ **nil**) **&** (*Last* ≠ **nil**);
EOList **&** **not** *Null* ↔ (*Head* ≠ **nil**) **&** (*Last* ≠ **nil**) **&** (*Cur* = **nil**).

Реализация Л2–списка на базе вектора также могла бы быть произведена с помощью добавления в каждое звено цепи (элемент массива типа *Memory-ForLists*) поля *Prev* ссылки на предшествующее звено. Однако с целью экономии памяти может быть рекомендована замена пары полей *Next* и *Prev* одним полем *Diff*, предназначенным для хранения разности ссылок на последующее и предшествующее звенья. Макет модуля *L2ListV*, содержащий описание соответствующих типов данных, приведен на рис.1.15. Описанный в модуле тип данных *L2_listV* соответствует типу данных *L_list* расширенной функциональной спецификации. Для краткости в макет модуля не включены заголовки и блоки процедур и функций.

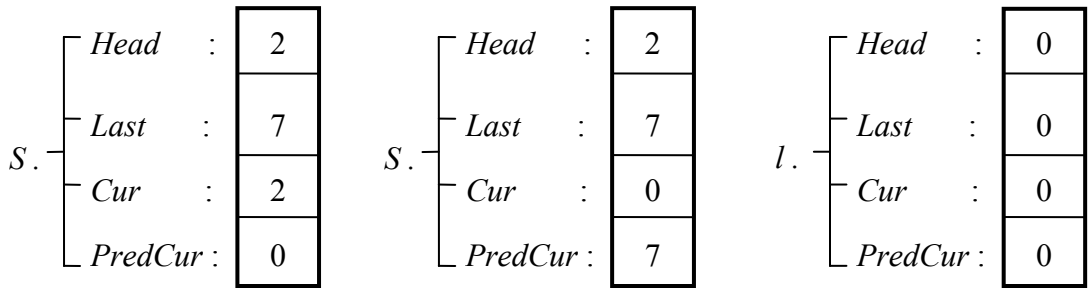
```
Unit L2ListV; {  Реализация ограниченного L2-списка на Турбо-Паскале     }
Interface
Uses Global;      {  модуль, в котором описан тип элементов списка El  }
                  {Ссылочное представление на базе вектора }
{const MemMaxSize = максимальный размер "рабочей" памяти, описан в модуле
Global}
type Index = 0..MemMaxSize;
Link     = Index;      {  ссылка на звено цепи              }
cord   {формуляр ("представитель") списка  }
Head , Cur, PredCur : Link ;
    end  { RepList };
    L2_listV = RepList;     { Л2–список = формуляр списка    }
     ...     { Здесь на месте ...  размещаются  заголовки  процедур и функций      }
Implementation
const NilLst = 0;             { nil для списков, в том числе  для списка свободной памяти
}
type DiffLink = - MemMaxSize..MemMaxSize;
record                  {звено цепи ( списка ) :                  }
    Info : El;          {   - содержимое (элемент списка)    }
    Diff : DiffLink;  {   - разность ссылок                      }
end  { Node } ;
MemoryForLists = array [ Index ] of Node;
{ "рабочий" массив для размещения списков  }
var Memory : MemoryForLists; i : Index;
    ...         {Здесь на месте "..." размещаются блоки процедур и функций   }
begin
{ инициализация списка свободной памяти в массиве Memory  }
Memory [ 0 ] . Diff := 1;
for i := 1 to ( MemMaxSize – 1 ) do Memory [ i ] . Diff := 2 ;
Memory [ MemMaxSize ] . Diff := NilLst - ( MemMaxSize - 1 );
end { L2ListV }.
```

Рис. 1.15. Макет модуля L2ListV

В качестве пустой ссылки используется значение *NilLst* = 0. При этом пример размещения списка в массиве, приведенный на рис.1.10, принимает вид рис. 1.16. Список свободной памяти здесь также организован как Л2–список, что позволяет эффективно реализовать операции *Empty* и *Destroy* (см. последний абзац 1.4).



а



а *в* *г*

Рис. 1.16. Представление ограниченного Л2-списка при ссылочной реализации на базе вектора: а – размещение списка s в массиве Memory; б,в – формуляры списка s в состояниях BOList и EOList соответственно; г – формуляр пустого списка l

Вычисление ссылки на элемент списка *s*, следующий за текущим, производится как *Memory* [*s*. *Cur*]. *Diff* + *s*. *PredCur* (не следует вычислять в состоянии *EOList*). Ссылка на элемент списка *s*, предшествующий тому, на который указывает *s*.*PredCur*, вычисляется как *s*. *Cur* – *Memory*[*s*. *PredCur*].*Diff* (не следует вычислять в состоянии *BOList*). Дополнительное поле *Last* в фор- муляре списка предназначено для хранения ссылки на последний элемент списка.

Модуль *Global* здесь имеет тот же вид, что и на рис. 1.13.

1.6. Рекурсивная обработка линейных списков

Рассмотренный ранее подход к организации линейных списков ориентирован на итеративную их обработку, при которой на каждом шаге выделяется

элемент списка, называемый текущим. Относительно этого элемента и производятся некоторые действия. Здесь список *s* фактически представляется парой последовательностей *Left*(*s*), *Right*(*s*), вторая из которых начинается с текущего элемента.

Рассмотрим теперь другой подход к организации и обработке списков, основанный на систематическом применении рекурсии и не предполагающий ни явного выделения текущего элемента, ни разделения списков на однонаправленные и двунаправленные. На рис.1.17 изображен линейный список, разделенный на две неравные части: "голову" (первый элемент списка) и "хвост" (все остальное).

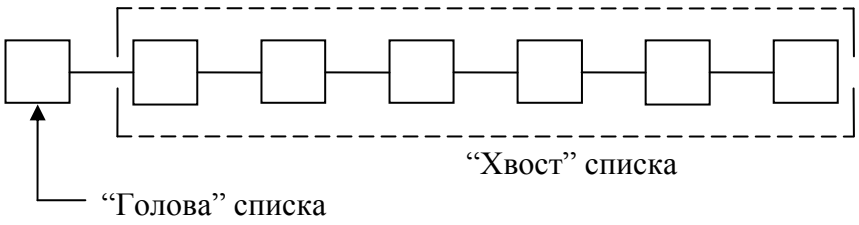


Рис. 1.17. Модельное представление линейного списка

Линейный список, таким образом, или пуст (не содержит ни одного элемента) или представляет собой упорядоченную пару "голова"–"хвост", в которой "голова" есть элемент базового типа *El*, а "хвост", в свою очередь, есть линейный список (возможно пустой). Отметим, что такое определение линейного списка по существу рекурсивно.

Пусть множество T_0 состоит из единственного элемента, которым является пустой список, и пусть T_n – множество всех линейных списков, состоящих ровно из n элементов базового типа *El*. Тогда множество T_1 одноэлементных линейных списков есть декартово произведение вида $T_1 = El \otimes T_0$, а множество T_n может быть определено как $T_n = El \otimes T_{n-1}\ \forall n : n > 0$. Такое индуктивное определение линейного списка отражает существенные характеристики его структуры и не зависит от конкретной формы наглядного представления или реализации списка. В дальнейшем будем считать его определением понятия "линейный список".

Списки, особенно обсуждаемые в 1.7 иерархические списки, часто представляют в виде различных, наглядных изображений или специфических форм записи. Одной из распространенных форм представления списков является так называемая скобочная запись, применяемая, например, в языке функционального программирования Лисп.

Используя форму Бэкуса-Наура (БНФ), дадим рекурсивное определение скобочной записи линейного списка, соответствующее ранее приведенному индуктивному определению множества линейных списков:

< *L_list*(*El*) > ::= < *Null_list* > | < *Non_null_list*(*El*) >
< *Null_lis t*> ::= *Nil*
< *Non_null_list*(*El*) > ::= < *Pair*(*El*) >
< *Pair*(*El*) > ::= (< *Head_l*(*El*) > . < *Tail_l*(*El*) >)
< *Head_l*(*El*) > ::= < *El* >
< *Tail_l*(*El*) > ::= < *L_list*(*El*) >

Здесь *L_list*(*El*) – линейный список из элементов типа *El*, *Null_list* – пустой список, *Non_null_list*(*El*) – непустой линейный список, *Head_l*(*El*) – "голова" списка, *Tail_l*(*El*) – "хвост" списка, *Pair*(*El*) – упорядоченная пара "голова"–"хвост", т.е. элемент соответствующего декартова произведения. Терминальным символом *Nil* обозначен пустой список, а терминальные символы (.) использованы для обозначения элемента декартова произведения. Выбор других терминальных символов в определении линейного списка породил бы другое сходное представление списка, сохранив при этом существенные характеристики его структуры.

Скобочное представление списка из элементов ***a, b, c, d*** типа *El* согласно приведенному определению есть конструкция вида (***a*** . (***b*** . (***c*** . (***d*** . *Nil*)))) или в сокращенной записи (***a b c d***). Пустой список представляется символом *Nil* или парой скобок ( ), а список из одного элемента (***d*** . *Nil*) в сокращенной записи имеет вид (***d***). Переход к сокращенной записи производится с помощью отбрасывания конструкции «. *Nil*» и удаления необходимое число раз пары скобок вместе с предшествующей открывающей скобке точкой. Пробелы в сокращенной записи используются для обеспечения однозначности прочтения конструкции, количество их выбирается произвольно. Если вместо пробелов в качестве разделителя использовать, например запятую, то сокращенная скобочная запись списка из элементов ***a, b, c, d*** будет иметь вид (***a , b , c , d***).

Зададим не зависящую от формы представления списка функциональную спецификацию линейного списка, определив с помощью системы правил (аксиом) *A*1-*A*5, справедливых для всех *x* типа *El*, всех *y* типа *L_list*(*El*), всех *z* типа *Non_null_list*(*El*), четыре базовые функции: предикат (индикатор) *Null* (список пуст), селекторы *Head* ("голова" списка) и *Tail* ("хвост" списка), конструктор *Cons* :

0) *Nil* : $\rightarrow$ *Null_list*;
1) *Null* : *L_list*(El) $\rightarrow$ *Boolean*;
2) *Head* : *Non_null_list*(*El*) $\rightarrow$ *El*;
3) *Tail* : *Non_null_list*(*El*) $\rightarrow$ *L_list*(*El*);
4) *Cons* : *El* $\otimes$ *L_list*(*El*) $\rightarrow$ *Non_null_list*(*El*);
*A*1) *Null*(*Nil*) = *true*;
*A*2) *Null*(*Cons*(*x* , *y*)) = *false*;
*A*3) *Head*(*Cons*(*x* , *y*)) = *x*;
*A*4) *Tail*(*Cons*(*x* , *y*)) = *y*;
*A*5) *Cons*(*Head*(*z*) , *Tail*(*z*)) = *z*.

Пустой список рассматривается здесь как значение типа *L_list*(*El*), возвращаемое функцией без параметров *Nil*. Доступ к элементам списка осуществляется с помощью соответствующей суперпозиции вызовов функций *Head* и *Tail*. Так, например, если $y = (\boldsymbol{a}\ \ \boldsymbol{b}\ \ \boldsymbol{c}\ \ \boldsymbol{d})$, то $Head(\ Tail(\ y\ )\ ) = \boldsymbol{b}$, а $Head(\ Tail(\ Tail(\ Tail(\ y\ )\ )\ )\ ) = \boldsymbol{d}$.

Способ использования функции *Cons* для конструирования списков поясняется приведенными далее примерами:

(***a***) = (***a*** . *Nil*) = *Cons*(***a*** , *Nil*);
(***a b c***) = (***a***. (***b***. (***c*** . *Nil*))) = *Cons*(***a*** , *Cons* (***b*** , *Cons* (***c*** , *Nil*))).

Отметим, что построение каждой "точечной" пары (значения типа *Pair*(*El*)) в скобочной записи списка требует однократного применения конструктора *Cons*.

Отвлекаясь от конкретного (скобочного) представления и опираясь на функциональную спецификацию линейного списка, можно дать новое определение типа *Pair*(*El*)

< *Pair* (*El*) > ::= *Cons*(< *Head_l* (*El*) > , < *Tail_l* (*El*) >)

в котором с помощью терминальных символов «*Cons* (,)» явно указывается на возможность построения пары "голова"–"хвост" (списка типа *Non_null_list*) только с помощью вызова конструктора *Cons*. Такое "операционное" представление списка вполне соответствует ранее рассмотренному. Так список из элементов ***a, b ,c*** типа *El* представляется теперь как *Cons*(***a***,*Cons*(***b***,*Cons*(***c***,*Nil*))). Нетрудно проверить, что правило *A*5 становится при этом излишним и может быть выведено из аксиом *A*3, *A*4.

Рассмотрим примеры рекурсивных определений некоторых функций обработки линейных списков с применением определенных выше базовых функций. В формулировках примеров и комментариях к ним использована сокращенная скобочная запись списков. Паскалеподобная нотация, примененная в определениях функций, ясна без дополнительных пояснений.

Пример 1.1. Функция Sum, вычисляющая сумму элементов списка.

Sum(*y*) = **if** *Null*(*y*) **then** 0
else *Head*(*y*) + *Sum*(*Tail*(*y*)).

Пример 1.2. Функция *Concat*, соединяющая два списка в один.

Например, *Concat*(*y* , *z*) = (***a b c d***) для *y* = (***a b***), *z* = (***c d***).

Concat(*y* , *z*) = **if** *Null*(*y*) **then** *z*
else *Cons*(*Head*(*y*), *Concat*(*Tail*(*y*), *z*)).

Отметим, что конструктор *Cons* вызывается при выполнении функции *Concat* столько раз, сколько элементов в списке y.

Пример 1.3. Функция *Append*, добавляющая элемент в конец списка.

Например, *Append*(*y*, *x*) = (***a b c***) для *y* = (***a b***), *x* = ***c***

Append(*y*, *x*) = *Concat*(*y*, *Cons*(*x*, *Nil*)).

Пример 1.4. Функция *Reverse*, обращающая список.

Например, *Reverse*(*y*) = (***d c b a***) для *y* = (***a b c d***).

Reverse(*y*) = **if** *Null*(*y*) **then** *Nil*

else *Concat* (*Reverse*(*Tail*(*y*)), *Cons*(*Head*(*y*), *Nil*)).

Для лучшего понимания работы функции *Reverse* рассмотрим последовательность вызовов функций и возвращаемых ими значений при обращении списка из двух элементов *y* = (***a b***) (рис.1.18).

Вызовы	Возвращаемые значения
Reverse((***a b***))	(***b a***)
Concat(*Reverse*((***b***)),*Cons*(***a***, *Nil*))	(***b a***)
* *Reverse*((***b***))	(***b***)
Concat(*Reverse*(*Nil*),*Cons*(***b***, *Nil*))	(***b***)
* *Reverse*(*Nil*)	*Nil*
* *Cons*(***b***,*Nil*)	(***b***)
* *Cons*(***a***,*Nil*)	(***a***)
Cons(***b***, *Concat*(*Nil*, (***a***)))	*(* ***b a***)
* *Concat*(*Nil*, (***a***))	(***a***)

Рис. 1.18. Последовательность вызовов функций при обращении списка из двух элементов

Вызов функции, с которого начинается вычисление значения фактического параметра ранее вызванной функции, на рисунке помечен символом *. Возвращаемые значения, формируемые в процессе вычислений позднее по времени, сдвинуты вправо относительно ранее сформированных значений. На рисунке для краткости не отражены вызовы селекторов *Head* и *Tail*. Вместо этого на соответствующие места подставлены возвращаемые ими значения.

Заметим, что количество вызовов конструктора *Cons* при обращении списка из *n* элементов равно $n + (n - 1) + .. + 1 = n (n + 1) / 2$ и может быть достаточно велико при большом *n*. Это важно, поскольку, как станет ясно из дальнейшего изложения, именно количество вызовов конструктора *Cons* в значительной степени определяет объем потребляемых вычислительных ресурсов. В следующем примере рассмотрено другое определение функции обращения списка, позволяющее сократить объем вычислений.

Пример 1.5. Функция *Reverse*1, обращающая список. С целью уменьшения количества вызовов конструктора *Cons* введена вспомогательная функция *Rev*, второй параметр которой используется как "накапливающий".

Rev(*y*, *z*) = **if** *Null*(*y*) **then** *z*

else *Rev* (*Tail*(*y*), *Cons* (*Head*(*y*), *z*));

*Reverse*1 (*y*) = *Rev* (*y*, *Nil*).

Последовательность вызовов функций и возвращаемых ими значений при обращении списка из двух элементов *y* = (***a b***), порождаемая вызовом функции *Reverse*1, приведена на рис. 1.19.

Вызовы	Возвращаемые значения
*Reverse*1 ((***a b***))	(***b a***)
Rev ((***a b***), *Nil*)	(***b a***)
Rev ((***b***), *Cons*(***a***, *Nil*))	(***b a***)
* *Cons* (***a***, *Nil*)	(***a***)
Rev (*Nil*, *Cons* (***b***, (***a***)))	(***b a***)
* *Cons* (***b***, (***a***)))	(***b a***)

Рис. 1.19. Последовательность вызовов функций, порождаемая вызовом Reverse1

Количество вызовов конструктора *Cons* в последнем варианте обращения списка длины *n*, очевидно, равно *n*, что по сравнению с первым вариантом дает существенный выигрыш в объеме вычислений.

Вопросы реализации рекурсивной обработки списков на Паскале рассмотрены далее в 1.7 сразу для иерархических списков, частным случаем которых являются списки линейные.

1.7. Иерархические списки

Рассмотренные в предыдущих разделах динамические структуры однотипных данных предназначены только для обработки *последовательностей* элементов, тогда как в практических приложениях возникает необходимость работы с более сложными, нелинейными конструкциями. Рассмотрим одну из них, называемую иерархическим списком элементов базового типа *El* или *S*-выражением. Определим соответствующий тип данных *S_expr* (*El*) рекурсивно, используя данное в 1.6 определение линейного списка (типа *L_list*):

< *S_expr* (*El*) > ::= < *Atomic* (*El*) > | < *L_list* (*S_expr*(*El*)) >

< *Atomic* (*E*) > ::= < *El* >

Иерархический список согласно определению представляет собой или элемент базового типа *El*, называемый в этом случае атомом (атомарным *S*–выражением), или линейный список из *S*–выражений. Приведенное определение задает структуру непустого иерархического списка как элемента размеченного объединения множества атомов и множества пар "голова"–"хвост" и порождает различные формы представления в зависимости от принятой формы представления линейного списка. Традиционно иерархические списки представляют или графически, используя для изображения структуры списка двухмерный рисунок, или в виде одномерной скобочной записи.

Рассмотрим примеры (рис. 1.20) иерархических списков из элементов базового типа *El*, представляющие списки в полной и сокращенной скобочной записи. Переход к сокращенной записи произведен по правилам, изложенным в 1.6.

Полная запись	Сокращенная запись
a *Nil* (**a .** (**b .** (**c .** *Nil*))) (**a .** ((**b .** (**c .** *Nil*)) **.** (**d .** (**e .** *Nil*))))	**a** ( ) (**a b c**) (**a** (**b c**) **d e**)

Рис. 1.20. Примеры иерархических списков

Зададим аналогично тому, как это было сделано в 1.6 для линейного списка, функциональную спецификацию иерархического списка, определив с помощью системы правил (аксиом) *A*1–*A*7, справедливых для всех *t* типа *El*, всех *u* типа *L_lis t*(*S_expr* (*El*)), всех *v* типа *Non_null_list* (*S_expr* (*El*)), всех *w* типа *S_expr* (*El*), константу *Nil*, обозначающую пустой список, четыре ранее уже встречавшиеся базовые функции *Null*, *Head*, *Tail*, *Cons*, а также предикат *Atom*, проверяющий *S*–выражение на атомарность:

0) *Nil* : $\rightarrow$ *Null_list*;
1) *Null* : *L_list* (*S_expr* (*El*)) $\rightarrow$ *Boolean*;
2) *Head* : *Non_null_list* (*S_expr* (*El*)) $\rightarrow$ *S_expr* (*El*);
3) *Tail* : *Non_null_list* (*S_expr* (*El*)) $\rightarrow$ *L_list* (*S_expr* (*El*));
4) *Cons* : *S_expr*(*El*) $\otimes$ *L_list*(*S_expr*(*El*)) $\rightarrow$ *Non_null_list*(*S_expr*(*El*));
5) *Atom* : *S_expr*(*El*) $\rightarrow$ *Boolean*;

*A*1) *Null* (*Nil*) = *true*;
*A*2) *Null* (*Cons* (*w*, *u*)) = *false*;
*A*3) *Head* (*Cons* (*w*, *u*)) = *w*;
*A*4) *Tail* (*Cons* (*w*, *u*)) = *u*;
*A*5) *Cons* (*Head*(*v*), *Tail*(*v*)) = *v*.
*A*6) *Atom* (*t*) = *true*;
*A*7) *Atom* (*u*) = *false*;

Согласно правилу *A*7 *Atom* (*Nil*) = *false*, что отличается от распространенного представления пустого списка как атома специального вида, но соответствует приведенному ранее определению иерархического списка. Доступ к компонентам *S*–выражения, которыми могут быть как атомы, так и списки, осуществляется с помощью селекторов *Head* и *Tail*. Так, например, если *u* = (***a*** (***b c***) ***d e***), то

Head (*Tail* (*u*)) = (***b c***);
Head (*Tail* (*Head* (*Tail* (*u*)))) = ***c***.

Рассмотрим теперь примеры использования функции *Cons* для конструирования списков:

(***a*** (***b c***) ***d e***) = (***a*** **.** ((***b*** **.** (***c*** **.** *Nil*)) **.** (***d*** **.** (***e*** **.***Nil*))))=
Cons (***a***, *Cons* (*Cons* (***b***, *Cons* (***c***, *Nil*)) , *Cons* (***d***, *Cons* (***e***, *Nil*))));
(***a*** (( ) (***b c***) ***d***) ***e***) = (***a***.((*Nil* . ((***b*** . (***c*** . *Nil*)) . (***d*** . *Nil*))) . (***e*** . *Nil*))) =
Cons(***a***, *Cons*(*Cons*(*Nil*,*Cons*(*Cons*(***b***,*Cons*(***c***,*Nil*)),*Cons*(***d***,*Nil*))),*Cons*(***e***,*Nil*))).

Рассмотренные примеры можно считать также и примерами "операционного" представления иерархических списков. Нетрудно заметить, что построение каждой "точечной" пары (значения типа *Pair* (*S_expr*(*El*))) в скобочной записи списка требует, как и в случае линейного списка, однократного применения конструктора *Cons*.

Еще одним традиционным способом представления иерархических списков является графическое их представление. На рис.1.21 приведены изображения иерархических списков, ориентированные на ссылочную реализацию в динамической памяти. Каждое звено списка на рисунке соответствует вызову конструктора *Cons* в "операционном" представлении или "точечной" паре в скобочной записи списка.

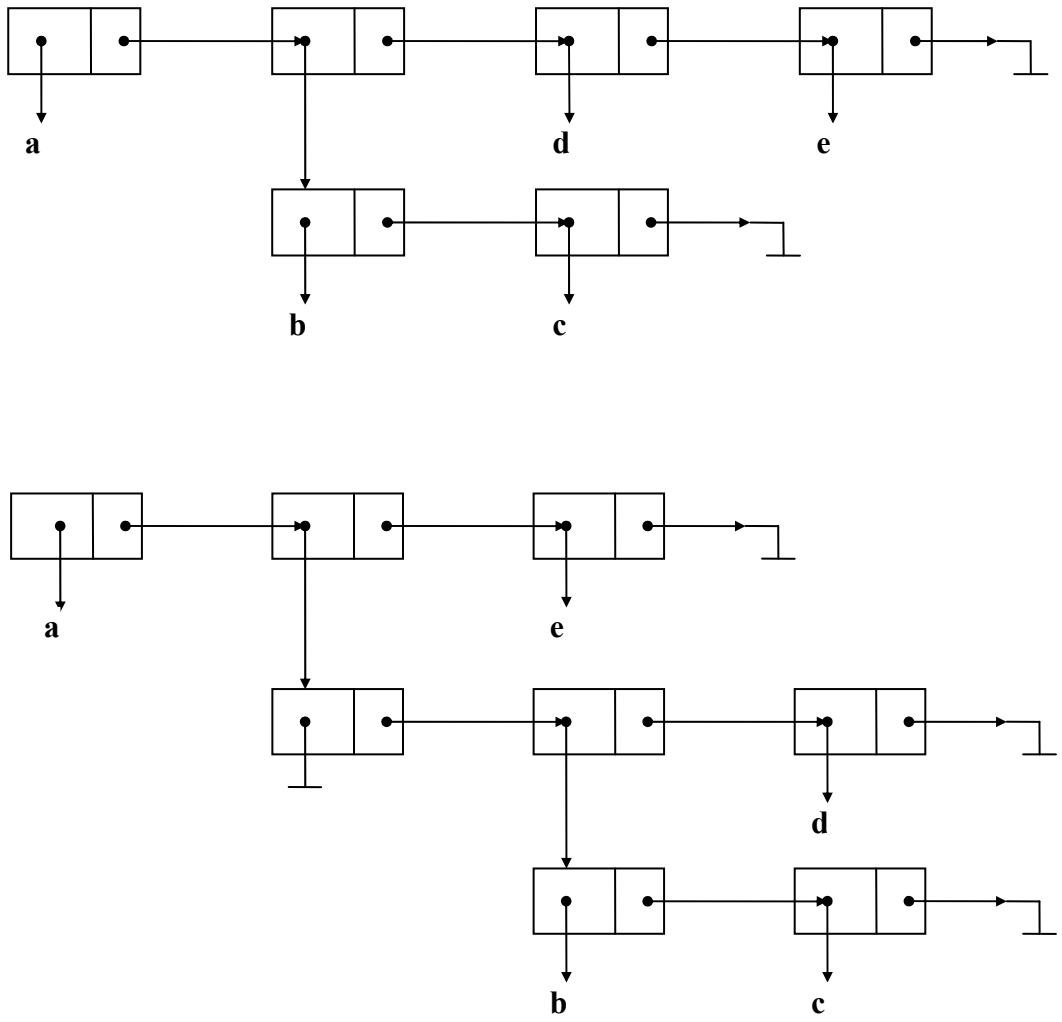


Рис. 1.21. Графическое представление иерархических списков:
*вверху – изображение списка (**a (b c) d e**);*
*внизу – изображение списка (**a (() (b c) d) e**)*

На рис. 1. 22 представлен макет модуля *HList*, реализующего на Турбо-Паскале типы данных и базовые функции, необходимые для организации и рекурсивной обработки иерархических списков. Значение типа *H_list* (иерархический список) здесь представляет собой ссылку на запись с вариантами типа *S_expr*, поле *Tag* которой идентифицирует запись как атом или как пару "голова"–"хвост", т.е. служит индикатором размеченного объединения. Предполагается, что пара "голова"–"хвост" может быть получена только с помощью вызова функции *Cons*, создающей экземпляр записи типа *S_expr* и возвращающей ссылку на эту запись, а содержимое полей *Hd* и *Tl* - как результат вызова функций *Head* и *Tail* соответственно.

Функция *Nil* не включена в состав модуля *HList*, так как при использовании предлагаемого представления списка пустой список *s* естественным образом представляется пустой ссылкой: *s* = **nil**.

Реализация абстрактного типа *Atomic* как одного из вариантов записи типа *S_expr* привела к появлению в составе модуля *HList* двух новых функций, ранее не включенных в число базовых:

$$Make\_Atom \ : \ El \ \rightarrow Atomic(\ El\ );$$

$$Get\_El : Atomic(\ El\ ) \rightarrow El.$$

Первую из них предлагается использовать для конструирования атомарных *S* – выражений, а вторую – для получения содержимого поля *Atm* варианта *Atomic* записи типа *S_expr*.

Формально это можно представить как изменение данного ранее определения абстрактного типа *Atomic* на определение

$$< Atomic\ (\ El\ ) > ::= \ Make\_Atom\ (\ < El >)$$

Функциональную спецификацию иерархического списка при этом необходимо дополнить правилом *A*8 для всех *t* типа *El*:

$$A8) \quad Get\_El\ (\ Make\_Atom(\ t\ )\ ) = t.$$

В дополнение к базовым функциям в состав модуля *HList* включены также функция копирования списка *Copy*, полезность которой будет ясна из дальнейшего изложения, и процедура *Destroy*, предназначенная для "сборки мусора", т.е. освобождения памяти, занятой ненужными уже списочными структурами. После выполнения *Destroy* (*x*) память, занятая списком *x*, освобождается, а значение переменной *x* становится неопределенным.

```
Unit HList;
        {Реализация иерархического списка на Турбо-Паскале}
Interface
Uses Global;        { модуль, в котором описан тип элементов списка El }
{================================================================== }
        {Ссылочное представление в динамической памяти  }
type
{ El – базовый тип для элементов списка (описан в Global )   }
Ref_S_expr = ^ S_expr;       {  ссылка на S – выражение        }
record                       { S – выражение:  }
 case Tag : ( Atomic, Pair ) of
Atomic : ( Atm: El );              { атомарное        }
   Pair : ( Hd, T l: Ref_S_expr )  { или пара          }
    end { S_expr };                    {"голова"–"хвост"}
     H_list = Ref_S_expr;          {иерархический список  }
{================================================================== }
{1} function Null ( x : H_list ) : Boolean;
{ список пуст }
{2}function Head ( x : H_list ) : H_List;
{селектор "голова" иерархического списка; отказ, если x – атом или пустой список }
{3} function Tail ( x : H_list ) : H_list;
 { селектор "хвост" иерархического списка; отказ, если x – атом или пустой список }
{4} function Cons ( x, y : H_list ) : H_list;
{ конструктор;  отказ, если y – атом }
{5} function Atom ( x : H_list ) : Boolean;
{ список атомарен }
{6} function Make_Atom ( x : El ) : H_list;
{ создать атомарное S – выражение }
{7} function Get_El ( x : H_list ) : El;
{ получить содержимое атомарного S – выражения; отказ, если x - не атом  }
{ } function Copy ( x : H_list ): H_list;
{ копировать список }
{ }  procedure Destroy ( x : H_list );
{освободить память, занятую под список }
{==================================================================}
Implementation
...    {Здесь на месте ... размещаются блоки процедур функций }
end { HList }.
```

Рис. 1.22. Макет модуля Hlist

Далее приведен один из возможных вариантов реализации описанных функций в секции **Implementation** модуля *HList*.

```
{1} function Null ( x : H_list ): Boolean;
     begin
         Null:= ( x = nil );
     end { Null };
```

```
{2} function Head ( x : H_list ): H_list;
    begin
      if  Null( x ) then
            begin WriteLn; WriteLn( ' ! Head( Nil ) ' ); Halt ;
            end
    else
      if Atom( x ) then
            begin WriteLn; WriteLn( ' ! Head( Atom ) ' ); Halt;
            end
      else Head := x ^ . Hd;
      end { Head };

{3} function Tail ( x : H_list ): H_list;
    begin
      if  Null( x ) then
            begin WriteLn; WriteLn( ' ! Tail( Nil ) ' ); Halt;
            end
    else
      if Atom( x ) then
            begin WriteLn; WriteLn( ' ! Tail( Atom ) ' ); Halt;
            end
      else Tail := x ^ . Tl;
    end { Tail };

{4} function Cons ( x, y : H_list ): H_list;
    var p : H_list;
    begin
      if Atom( y ) then
    begin WriteLn; WriteLn( ' ! Cons( x, Atom ) ' ); Halt; end
    else
      if MaxAvail >= SizeOf( S_expr ) then
            begin New( p );
              p ^ . Tag:= Pair;   p ^ . Hd := x;   p ^ . Tl := y;
              Cons := p;
            end
    else
      begin WriteLn; WriteLn( ' ! Исчерпана память ' ); Halt;
      end;
    end { Cons };

{5} function Atom ( x : H_list ): Boolean;
    begin
      if  Null ( x ) then Atom := false
      else Atom := ( x ^.Tag = Atomic );
       end { Atom };
```

```
{6} function Make_Atom ( x : El ): H_list;
    var p : H_list;
    begin
      if MaxAvail >= SizeOf ( S_expr ) then
            begin New ( p );
                  Make_Atom := p;
                  p ^ . Tag := Atomic;   p ^ . Atm := x;
    end
    else
    begin WriteLn; WriteLn ( ' ! Исчерпана память ' ); Halt; end;
     end { Make_Atom };

{7} function Get_El ( x : H_list ): El;
    begin
      if not Atom ( x ) then
            begin WriteLn; WriteLn( ' ! Get_El ( not Atom ) ' ); Halt; end
      else Get_El := x ^ . Atm;
    end { Get_El };
{ }  function Copy ( x : H_list ): H_list;
    begin
      if  Null ( x ) then Copy := nil
      else if Atom ( x ) then Copy := Make_Atom ( x ^.Atm )
            else Copy := Cons ( Copy( Head( x ) ), Copy ( Tail( x ) ) );
      end { Copy };
{ }  procedure Destroy ( x : H_list );
    begin
      if not Null ( x ) then
            begin
                  if not Atom ( x ) then
                        begin Destroy ( Head ( x ) );   Destroy ( Tail( x ) );  end;
                  Dispose ( x );
            end;
      end { Destroy };
```

Отметим, что предложенная функция *Cons* не формирует копий исходных *S* – выражений, а только создает экземпляр записи типа *S_expr* (вариант *Pair*) и записывает ссылки на "голову" и "хвост" конструируемого *S* – выражения в поля *Hd* и *Tl* этого экземпляра соответственно. Программист, таким образом, должен сам заботиться о копировании списочных структур, избегая побочного эффекта от включения одних списков в другие в качестве фрагментов. Поясним сказанное на ранее рассмотренных в 1.6 примерах функций рекурсивной обработки линейных списков. Вернемся еще раз к этим примерам и рассмотрим, как указанные функции могут быть реализованы применительно к иерархическим спискам с использованием модуля *HList*.

Определенная в примере 1.2 функция *Concat*, примененная, например, к спискам y = (***a*** (***b ш***)), z = (***d e***) должна возвратить значение *Concat* (y, z) = = (***a*** (***b c***) ***d e***) . На Паскале она может быть записана следующим образом:

```
function Concat ( y, z : H_list ) : H_list;
begin
      if Null ( y ) then Concat := Copy ( z )
      else Concat := Cons( Copy( Head( y ) ), Concat( Tail( y ), z ) );
end;
```

Функция создает новый иерархический список из копий атомов, входящих в соединяемые списки. На рис.1.23 изображена последовательность вызовов функций и возвращаемых ими значений, порождаемая вызовом функции *Concat*. Верхние индексы на рисунке использованы для обозначения номеров экземпляров (копий) атомов. Для краткости все вызовы функций *Head*, *Tail*, а также большинство вызовов *Copy*, заменены на возвращаемые этими функциями значения.

Вызовы	Возвращаемые значения
Concat((***a*** (***b c***)), (***d e***))	(a^1 (b^1 c^1) d^1 e^1)
Cons(a^1, *Concat*(((***b c***)), (***d e***)) **Concat*(	(a^1 (b^1 c^1) d^1 e^1)
((***b c***)), (***d e***))	((b^1 c^1) d^1 e^1)
Cons((b^1 c^1),*Concat*(*Nil*,(***d e***)))	((b^1 c^1) d^1 e^1)
Concat*(*Nil*, (***d e*))	(d^1 e^1)
Copy((***d e***))	(d^1 e^1)

Рис. 1.23. Последовательность вызовов функций при соединении двух списков

Обращающая список функция *Reverse*, определение которой дано в примере 1.4, также может быть применена к иерархическим спискам. Например, *Reverse*(y) = (***d*** (***b c***) ***a***) для y = (***a*** (***b c***) ***d***). На Паскале функция может иметь вид

```
function Reverse ( y : H_list ) : H_list;
var p1, p2 : H_list;
begin
      if Null( y ) then Reverse := nil
      else begin p1 := Reverse ( Tail( y ) );
                 p2 := Cons ( Copy ( Head( y ) ), nil );
                 Reverse := Concat ( p1, p2 );
                 Destroy( p1 );   Destroy ( p2 );
       end;
end;
```

Как можно видеть на рис. 1.24, в процессе выполнения *Reverse* создаются рабочие копии атомов, входящие во временно существующие списки. Так, атом ***d*** копируется четыре раза, причем все копии, кроме четвертой копии d^4, входящей в результирующий список, уничтожаются вместе с временно существующими списками. На рис. 1.24 уничтожаемые при выполнении процедуры *Destroy* списки изображены в квадратных скобках.

Вызовы	Результаты вызовов
Reverse((***a*** (***b c***) ***d***))	(d^4 (b^3 c^3) a^2)
Reverse(((***b c***) ***d***))	(d^3 (b^2 c^2))
Reverse((***d***))	(d^2)
Reverse(*Nil*)	( )
Cons(d^1, *Nil*)	(d^1)
Concat(*Nil*, (d^1))	
Destroy(*Nil*)	(d^2)
Destroy((d^1))	[(d^1)]
Cons((b^1 c^1), *Nil*)	((b^1 c^1))
Concat((d^2), ((b^1 c^1)))	(d^3 (b^2 c^2))
Destroy((d^2));	[(d^2)] [((b^1 c^1))]
Destroy(((b^1 c^1)))	(a^1)
Cons(a^1, Nil)	(d^4 (b^3 c^3) a^2)
Concat((d^3 (b^2 c^2)), (a^1))	[(d^3 (b^2 c^2))]
Destroy((d^3 (b^2 c^2)))	[(a^1)]
Destroy((a^1))	

Рис. 1.24. Последовательность вызовов подпрограмм при обращении иерархического списка

Для краткости на рисунке не приведены вызовы функций *Head*, *Tail*, *Copy*, а также последовательности вызовов функций, порождаемые вызовом *Concat*.

Использующая накапливающий параметр функция обращения списка *Rev* реализуется, например как

```
function Rev ( y, z : H_list ) : H_list;
begin
      if Null( y ) then Rev := z
      else Rev := Rev ( Tail ( y ), Cons ( Copy ( Head( y ) ), z ) );
end;
```

Функция *Append* из примера 1.3, добавляющая элемент в конец списка, может быть реализована как

```
function Append ( y : H_list;  x : El ) : H_list;
var p : H_list;
begin
      p := Cons( Make_Atom( x ), nil );
      Append := Concat ( y, p );
      Destroy ( p );
 end;
```

Подпрограммы ввода–вывода иерархических списков должны быть написаны применительно к конкретной форме представления списка и помещены в отдельный модуль. Так, если список представляется сокращенной скобочной записью, размещенной в текстовом файле, а *El* – любой из типов, совмести-

мых с типом *Char*, то процедура *Read_H_list* ввода иерархического списка и используемые ею вспомогательные процедуры *Read_S_expr* и *Read_seq* могут быть записаны в виде

```
{1} procedure Read_H_list ( var f: Text;  var y: H_list );
      var x : El;
      begin
        repeat Read( f, x );  until x <> ' ';
              Read_S_expr( f, x, y );
        end { Read_H_list };
{2} procedure Read_S_expr ( var f: Text; prev: El; var y: H_list );
            { prev – ранее прочитанный символ }
      var x : El;
      begin
        if prev = ' ) ' then begin Writeln( ' ! Ошибка 1 ' ); Halt; end
        else
          if prev <> ' ( ' then y := Make_Atom( prev )
          else Read_seq ( f, y );
      end { Read_S_expr };
{3} procedure Read_seq ( var f : Text; var y: H_list);
      var x : El;   p1, p2 : H_list;
      begin
       if Eof( f ) then begin WriteLn(' ! Ошибка 2 ' ); Halt; end
       else
         begin   repeat Read( f , x ) until  x <>'   ';
           if x = ' ) '  then y := Nil
           else begin Read_S_expr( f ,  x ,  p1 );
                      Read_Seq( f ,  p2 );    y := Cons( p1 , p2 );
                end;
         end;
end {Read_Seq};
```

Процедура вывода списка с обрамляющими его скобками *Write_H_list*, а без обрамляющих скобок *Write_List.*

```
{4} procedure Write_H_list ( S : H_list );
    begin                                {пустой список выводится как () }
      if Null( x ) then Write( ' (  ) ' )
      else
            if Atom( x ) then Write( ' ', x ^ . Atm )
            else                                          {непустой список}
                  begin Write( ' ( ' );  Write_List( x );   Write( ' ) ' ) end;
    end {Write_H_list};
```

```
{5} procedure Write_List ( x : Ref_S_expr );
    begin                          {выводит последовательность элементов }
      if not Null( x ) then     {списка без обрам ляющихего скобок}
            begin Write_H_list ( Head ( x ) );  Write_List ( Tail ( x ) )
            end;
    end {Write_List};
```

Смысл сообщений об ошибках, обнаруженных при вводе иерархического списка, очевиден. Пробелы при вводе списка игнорируются. Процедуру вывода списка предлагается написать самостоятельно.

1.8. Упражнения

Каждое задание предполагает самостоятельную разработку студентом одного или нескольких модулей Турбо-Паскаля, реализующих согласованный с преподавателем набор операций над списками, а также главной программы, непосредственно решающей поставленную задачу.

В заданиях 1-16 термином "список" обозначен линейный список. При выполнении этих заданий следует применять по указанию преподавателя итеративные или рекурсивные процедуры обработки линейных списков. Задание 17 предлагается выполнить в двух вариантах: с использованием базовых функций рекурсивной обработки иерархических списков и без использования рекурсии.

Во всех случаях, когда это не оговорено особо, предполагается, что исходные и результирующие списки размещаются в файлах подходящего типа. Для представления иерархических списков в задании 17 рекомендуется использовать сокращенную скобочную запись.

1. Вставить в список *l* элементов типа *Real*:

а) новый элемент *e*1 перед каждым вхождением элемента *e*;

б) новый элемент *e*1 за каждым вхождением элемента *e*;

в) новый элемент *e* так, чтобы сохранилась упорядоченность по неубыванию исходного непустого списка.

2. Удалить из списка *l* элементов типа *Real*:

а) за каждым вхождением элемента *e* один элемент, если такой есть и он отличен от *e*;

б) все отрицательные элементы.

3. Проверить:

а) равны ли списки *l*1 и *l*2;

б) входит ли список *l*1 в список *l*2;

в) равны ли множества, представленные списками *l*1 и *l*2 (рассматривать одинаковые элементы списка как один элемент множества);

г) составляют ли элементы списка *l*1 подмножество элементов списка *l*2;

д) есть ли в списке *l* хотя бы два одинаковых элемента;

е) предшествует ли в списке *l* первое вхождение элемента *e*1 первому вхождению элемента *e*2.

4. Выполнить следующие действия:

а) добавить список $l2$ в конец списка $l1$;

б) добавить в конец списка $l1$ все элементы списка $l2$ в том порядке, в котором они впервые встречаются в списке $l2$;

в) добавить в конец списка $l1$ все элементы списка $l2$ в порядке обратном тому, в котором они впервые встречаются в списке $l2$;

г) вставить список $l2$ в список $l1$ за первым вхождением элемента e, если e входит в $l1$;

д) обратить список l, т.е. расположить элементы списка l в обратном порядке;

е) в списке l из каждой группы подряд идущих одинаковых элементов оставить только один;

ж) оставить в списке l только первые вхождения одинаковых элементов;

з) перегруппировать элементы списка l так, чтобы одинаковые элементы, если они есть в списке, стояли все подряд.

5. Вычислить для списка l элементов типа *Integer*:

а) число пар соседних взаимно простых элементов;

б) число локальных максимумов (текущий элемент будем называть локальным максимумом, если нет соседнего элемента, большего, чем текущий элемент);

в) число элементов списка, больших всех предыдущих элементов;

г) число элементов списка, меньших всех последующих элементов.

6. Сформировать список l, включив в него по одному разу элементы, которые входят:

а) хотя бы в один из списков $l1$, $l2$;

б) одновременно в оба списка $l1$, $l2$;

в) в список $l1$, но не входят в список $l2$;

г) в один из списков $l1$ или $l2$, но не входят в другой.

7. Пусть список l содержит упорядоченную (например, по возрастанию) последовательность элементов типа *El*, не содержащую одинаковых элементов. Будем говорить, что список l представляет упорядоченное множество S, и кратко записывать это, как $l = R(S)$. Определим четыре варианта операции X над множествами:

1) $\otimes = \cup$ (объединение),

2) $\otimes = \cap$ (пересечение),

3) $\otimes = \setminus$ (разность),

4) $\otimes = \Delta$ (симметрическая разность).

Выполнить задание, используя по указанию преподавателя одно из четырех приведенных ранее определений операции X:

а) заданы списки $l1 = R(S1)$ и $l2 = R(S2)$. Вычислить количество элементов множества $S3 = S1 \otimes S2$;

б) заданы списки $l1 = R(S1)$ и $l2 = R(S2)$. Проверить истинность утверждения $S1 \subset S2$;

в) заданы списки $l1 = R(S1)$ и $l2 = R(S2)$. Сформировать список $l3 = R(S3)$, где $S3 = S1 \otimes S2$;

г) заданы списки $l1 = R$ ($S1$), $l2 = R$ ($S2$) и $l3 = R$ ($S3$). Проверить истинность утверждения

А) $S3 \subset S1 \otimes S2$,

Б) $S1 \otimes S2 \subset S3$.

8. Упорядочить по неубыванию список l элементов типа *Real* с помощью одного из методов сортировки:

а) выбором: отыскивается максимальный элемент и переносится в конец списка; затем этот метод применяется ко всем элементам, кроме последнего (он уже на месте) и т.д.;

б) обменом (прогоном пузырька): последовательно, начиная с первого элемента списка, упорядочиваются пары соседних элементов (при этом максимальный элемент оказывается в конце списка); затем этот метод применяется ко всем элементам списка, кроме последнего и т.д.;

в) вставками: пусть элементы списка, предшествующие текущему, уже упорядочены; берется текущий элемент (текущим после этого становится следующий за ним) и размещается среди упорядоченных так, чтобы не нарушить упорядоченности; этот метод применяется для всех элементов, начиная со второго и до конца списка;

9. Объединить два упорядоченных по неубыванию списка $l1$ и $l2$ (элементы обоих списков имеют тип *Real*) в один упорядоченный по неубыванию список:

а) построив новый список l;

б) дополнив элементами списка $l2$ список $l1$.

10. В списке l заменить:

а) первое вхождение списка $l1$ (если такое есть) на список $l2$;

б) все вхождения списка $l1$ на список $l2$;

11. Разработать процедуру или функцию, обеспечивающую выполнение произвольного, задаваемого с помощью входного параметра набора действий:

а) для каждого элемента списка l;

б) для каждого элемента списка l от начала списка и до элемента, предшествующего текущему;

в) для каждого элемента списка, начиная с текущего и до конца списка.

12. Разработать процедуру, образующую списки $l1$ и $l2$ из элементов списка l, соответственно обладающих и не обладающих произвольным, задаваемым с помощью входного параметра свойством.

13. Заданный во входном файле текст переписать в выходной файл в обратном порядке.

14. Пусть задан список l слов, где

< слово > ::= < буква > | < буква > < слово >

< буква > ::= $a \mid b \mid c \mid d \mid e \mid f \mid g \mid h \mid i \mid j \mid k \mid l \mid m \mid n \mid o \mid p \mid q \mid r \mid s \mid t \mid u \mid v \mid w \mid x \mid y \mid z$

а) сформировать список из первых букв всех слов списка l;

б) удалить из всех слов списка l их последние буквы.

15. Многочлен $p(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$ можно представить в виде упорядоченного по убыванию степени i одночленов $a_i x^i$ списка пар (i,a_i), не содержащего пар вида (i , 0).

Описать тип данных, соответствующий представлению многочлена с вещественными коэффициентами в виде списка, и выполнить следующие действия со списками–многочленами:

а) проверить на равенство многочлены p и q;

б) вычислить значение многочлена p в целочисленной точке x;

в) построить многочлен p – производную многочлена q;

г) построить многочлен p – сумму многочленов q и r;

д) построить многочлен p путем возведения многочлена q в заданную целую положительную степень n;

е) построить многочлены p и q как частное и остаток от деления многочленов r и s;

ж) построить многочлен p как наибольший общий делитель многочленов q и r;

з) напечатать многочлен p как многочлен от переменной, однобуквенное имя которой является значением параметра типа *Char*;

и) ввести из входного файла многочлен, записанный в виде неупорядоченной совокупности пар вида (i, a_i), и сформировать соответствующий список–многочлен p.

16. Пусть произвольный текст рассматривается как список строк. Разработать программу для построчного редактирования текста в режиме диалога с пользователем.

17. Решить следующие задачи с использованием базовых функций рекурсивной обработки списков:

а) проверить иерархический список на наличие в нем заданного элемента (атома) x;

б) удалить из иерархического списка все вхождения заданного элемента (атома) x;

в) заменить в иерархическом списке все вхождения заданного элемента (атома) x на заданный элемент (атом) y;

г) подсчитать число атомов в иерархическом списке; сформировать линейный список атомов, соответствующий порядку подсчета;

д) подсчитать число различных атомов в иерархическом списке; сформировать из них линейный список;

е) сформировать линейный список атомов исходного иерархического списка путем устранения всех внутренних скобок в его сокращенной скобочной записи;

ж) проверить идентичность двух иерархических списков;

з) вычислить глубину (число уровней вложения) иерархического списка как максимальное число одновременно открытых левых скобок в сокращенной скобочной записи списка; принять, что глубина пустого списка и глубина атомарного S – выражения равны нулю; например, глубина списка (***a*** (***b*** ( ) ***c***) ***d***) равна двум;

и) обратить иерархический список на всех уровнях вложения; например, для исходного списка (***a*** (***b*** ***c***) ***d***) результатом обращения будет список (***d*** (***c b***) ***a***).

2. СТЕКИ И ОЧЕРЕДИ

2.1. Спецификация стека и очереди

Ранее в 1.2 и 1.5 при задании спецификации линейных списков использовалась абстракция (модель) последовательности. При этом были определены функции над последовательностями *First*, *Last*, *Rest*, *Lead*, *Prefix* и *Postfix*. Напомним эти определения, сохранив те же обозначения, что и ранее, и дополнительно обозначив тип последовательности $w = [\, x_1 , x_2 , \ldots , x_n \,]$ из элементов x_i типа *El* как *Sequence* (*El*). Функции *First*, *Last*, *Rest*, *Lead* определены только для непустых последовательностей ($w \neq \Delta$):

First: *Sequence* (*El*) $\rightarrow$ *El*; *First* (w) = x_1;
Last: *Sequence* (*El*) $\rightarrow$ *El*; *Last* (w) = x_n;
Rest: *Sequence* (*El*) $\rightarrow$ *Sequence* (*El*); *Rest* (w) = $[\, x_2 , \ldots , x_n \,]$;
Lead: *Sequence* (*El*) $\rightarrow$ *Sequence* (*El*); *Lead* (w) = $[\, x_1 , x_2 , \ldots , x_{n-1} \,]$.

Функции *Prefix* и *Postfix* определены для любых последовательностей:

Prefix: *El* $\otimes$ *Sequence* (*El*) $\rightarrow$ *Sequence* (*El*);
Postfix: *Sequence* (*El*) $\otimes$ *El* $\rightarrow$ *Sequence* (*El*);
Prefix (x_0 , w) = $[\, x_0 , x_1 , x_2 , \ldots , x_n \,]$; *Postfix* (w , x) = $[\, x_1 , x_2 , \ldots , x_n , x \,]$.

Таким образом, набор функций *First*, *Rest*, *Prefix*, *Last*, *Lead*, *Postfix* обеспечивает доступ к элементам последовательности (“чтение”элементов из последовательности и дописывание элементов в последовательность) только через ее начало и ее конец. Последовательность может рассматриваться как самостоятельная структура данных. Тогда функции *First*, *Rest*, *Last*, *Lead* - селекторы, а функции *Prefix* и *Postfix* - конструкторы типа *Sequence* (*El*). Более того, используя разные подмножества набора базовых функций этой структуры, получают различные полезные структуры данных. Так, если ограничиться только функциями *First*, *Rest*, *Prefix* (или только функциями *Last*, *Lead*, *Postfix*), то получается структура данных, известная как *стек* (или магазин). Рассмотрение только функций *First*, *Rest*, *Postfix* (или только *Last*, *Lead*, *Prefix*) соответствует структуре данных *очередь* (англ. *queue*). Если же используют весь набор функций, то соответствующую структуру данных обычно называют *дек* (от англ. *deq* - аббревиатуры сочетания double-ended-queue, т.е. “очередь с двумя концами”). Во всех этих структурах данных необходимо добавить еще две функции: 1) предикат-индикатор *Null* : *Sequence* (*El*) $\rightarrow$ *Boolean*,

идентифицирующий пустую последовательность Δ, и 2) либо константу, обозначающую пустую последовательность, либо функцию, порождающую значение Δ, например *Create* : → *Sequence* (*El*).

Рассмотрим формальную спецификацию ***стека*** из элементов типа α (*Stack of* α ≡ *Stack* (α)). При этом для обозначения функций *First*, *Rest*, *Prefix* будем использовать исторически сложившиеся синонимы *Top*, *Pop* и *Push* соответственно (*Top* − верхушка стека, *Pop* (*up*) − вытолкнуть (вверх), *Push* (*down*) – втолкнуть, вжать (вниз)). Функциональная спецификация стека задается следующими определениями (множество непустых стеков обозначим как *Non_null_stack*):

1) *Create*: → *Stack* (α) ;
2) *Null*: *Stack* (α) → *Boolean* ;
3) *Top*: *Non_null_stack* (α) → α ;
4) *Pop*: *Non_null_stack* (α) → *Stack* (α) ;
5) *Push*: α ⊗ *Stack* (α) → *Stack* (α)

и набором аксиом (∀*p*: α; ∀*s*: *Stack* (α); ∀*t*: *Non_null_stack* (α)):

A1) *Null* (*Create*) = *true* ;
A2) *Null* (*Push* (*p* , *s*)) = *false* ;
A3) *Top* (*Push* (*p* , *s*)) = *p* ;
A4) *Pop* (*Push* (*p* , *s*)) = *s* ;
A5) *Push* (*Top* (*s*) , *Pop* (*s*)) = *s*.

Можно заметить, что так определенный абстрактный тип *Stack* (α) фактически (с точностью до обозначений и несущественных деталей) совпадает с ранее рассмотренным в 1.6 типом *L_list* (α).

Часто при определении стека вместо функции *Pop* используют функцию (процедуру), совмещающую результат действия функций *Top* и *Pop*. Обозначим такую процедуру *Pop*2. Тогда

*Pop*2: *Non_null_stack* (α) → α ⊗ *Stack* (α).

Можно явно определить *Pop*2 через функции *Top* и *Pop*:

```
procedure Pop2 ( out p: α;   in-out s:  Stack ( α )  );
begin
  p := Top ( s );
  s := Pop ( s )
end
```

Функциональная спецификация ***очереди*** из элементов типа α (*Queue of* α ≡ *Queue* (α)) задается следующими определениями:

1) *Create*: → *Queue* (α) ;
2) *Null*: *Queue* (α) → *Boolean* ;
3) *First*: *Non_null_queue* (α) → α ;
4) *Rest*: *Non_null_queue* (α) → *Queue* (α) ;
5) *Postfix*: *Queue* (α) ⊗ α → *Queue* (α) ;

и набором аксиом (∀*p*: α; ∀ *q*: *Queue* (α)):

A1) *Null* (*Create*) = *true* ;
A2) *Null* (*Postfix* (*q* , *p*)) = *false* ;
A3) *First* (*Postfix* (*q* , *p*)) = **if** *Null* (*q*) **then** *p* **else** *First* (*q*) ;
A4) *Rest* (*Postfix* (*q* , *p*)) = **if** *Null* (*q*) **then** *Create*
else *Postfix* (*Rest* (*q*) , *p*).

В качестве примера использования базовых функций при операциях над очередью дадим определение функции сцепления двух очередей *q*1 и *q*2:

Concat (*q*1 , *q*2) ≡ **if** *Null* (*q*1) **then** *q*2 **else if** *Null* (*q*2) **then** *q*1
else {**not** *Null* (*q*1) & **not** *Null* (*q*2)}
Concat (*Postfix* (*q*1 , *First* (*q*2)) , *Rest* (*q*2))

2.2. Реализация стека и очереди

Ссылочная реализация стека и очереди в динамической памяти в основном аналогична ссылочной реализации линейных списков, подробно рассмотренной в 1.3. Упрощение здесь связано с отсутствием необходимости работать с текущим (“внутренним”) элементом списка. Идеи такой реализации ясны из рис. 2.1.

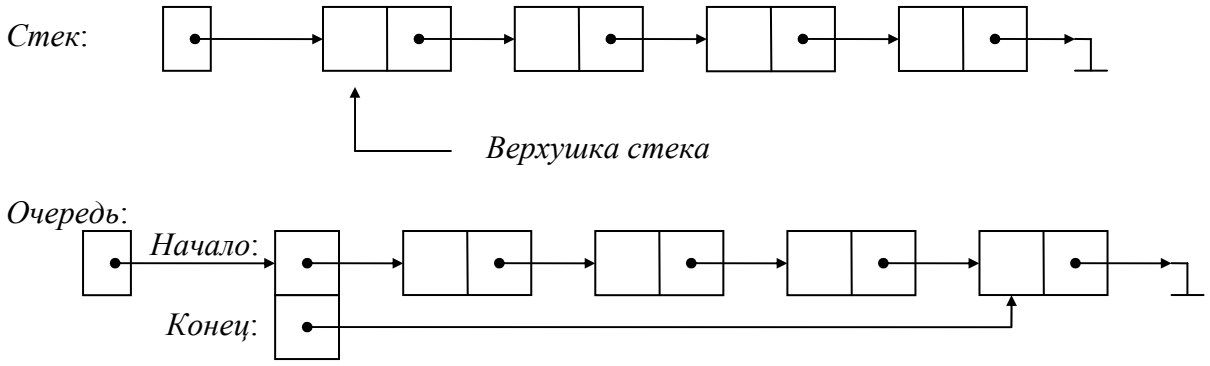


Рис. 2.1. Ссылочное представление стека и очереди

Для ссылочной реализации дека естественно использовать Л2-список (см.1.5).

Поскольку для стека, очереди и дека доступ к элементам осуществляется только через начало и конец последовательности, то эти структуры данных допускают эффективную *непрерывную реализацию на базе вектора* (в отличие от Л1- и Л2-списков, см.1.1, с.4).

При *непрерывной реализации ограниченного стека* на базе вектора для представления стека используется одномерный массив (вектор) *Мет*: **array** [1 .. *n*] **of** α и переменная *Верх*: 1 .. *n* (рис. 2.2).

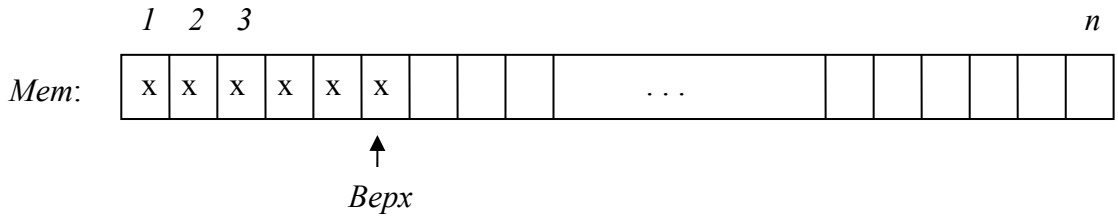


Рис. 2.2. Непрерывное представление стека в векторе

Для пустого стека *Верх* = 0, для целиком заполненного стека *Верх* = *n*. Вершина стека доступна как *Мет* [*Верх*], операция *Pop* реализуется как *Верх*:= *Верх* – 1, а операция *Push* (*p* , *s*) как **begin** *Верх*:= *Верх* + 1; *Мет* [*Верх*]:=*p* **end** при 0≤*Верх*<n.

На базе одного вектора можно реализовать два стека, ограниченных в совокупности. Такую реализацию иллюстрирует рис. 2.3.

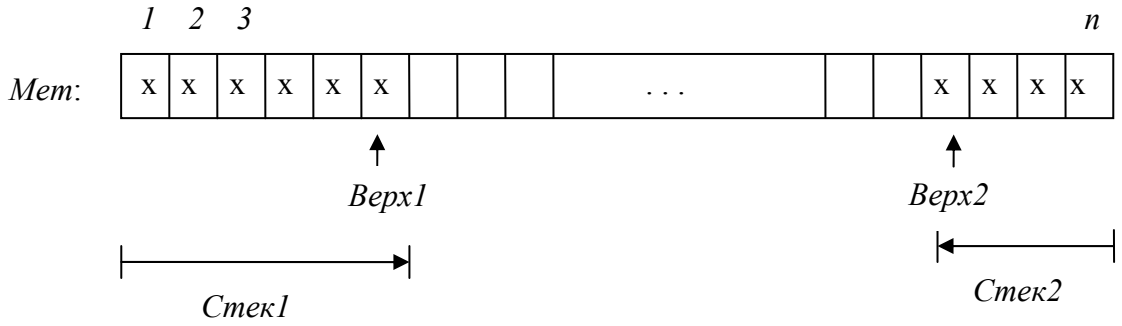


Рис. 2.3. Непрерывное представление двух стеков, ограниченных в совокупности

Рассмотрим основные идеи *непрерывной реализации ограниченной очереди* на базе вектора. На рис.2.4 изображен вектор *Мет* [1 .. *n*] и переменные *Начало, Конец*: 1..*n*, идентифицирующие начало и конец очереди при ее непрерывном размещении в векторе *Мет*.

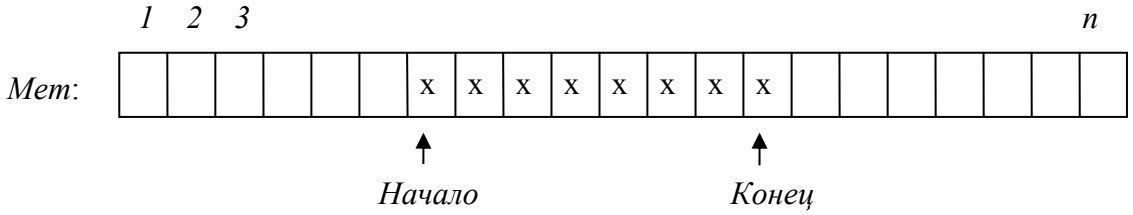


Рис. 2.4. Непрерывное представление очереди в векторе

Особенностью такого представления является наличие ситуации, когда последовательность элементов очереди по мере их добавления может выходить

за границу вектора, продолжаясь с его начала (вектор имитирует здесь так называемый *кольцевой буфер*). Эта ситуация изображена на рис. 2.5.

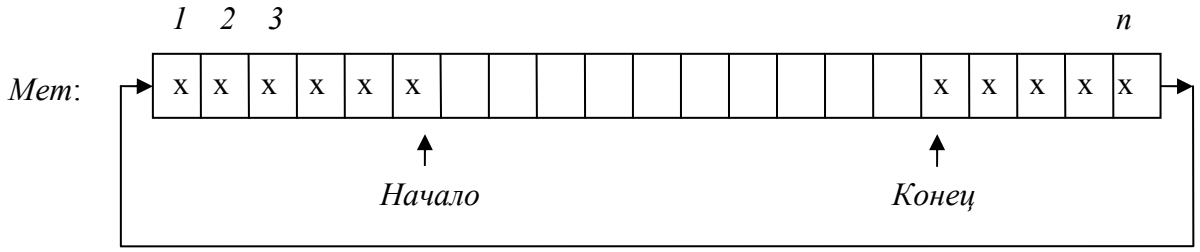


Рис. 2.5. Непрерывное представление очередив кольцевом буфере

Двух переменных *Начало* и *Конец* недостаточно, чтобы различить в данном представлении, например, два следующих состояния очереди: 1) *Начало* = *Конец* + 1 и очередь пуста (см.рис.2.6,а); 2) *Начало* = *Конец* + 1 и очередь полна (см.рис.2.6,б). Простым решением этой проблемы является введение еще одной переменной, идентифицирующей состояние очереди, а именно переменной *Длина*, значение которой задает текущее количество элементов в очереди (для пустой очереди *Длина* = 0, для полной очереди *Длина* = *n*).

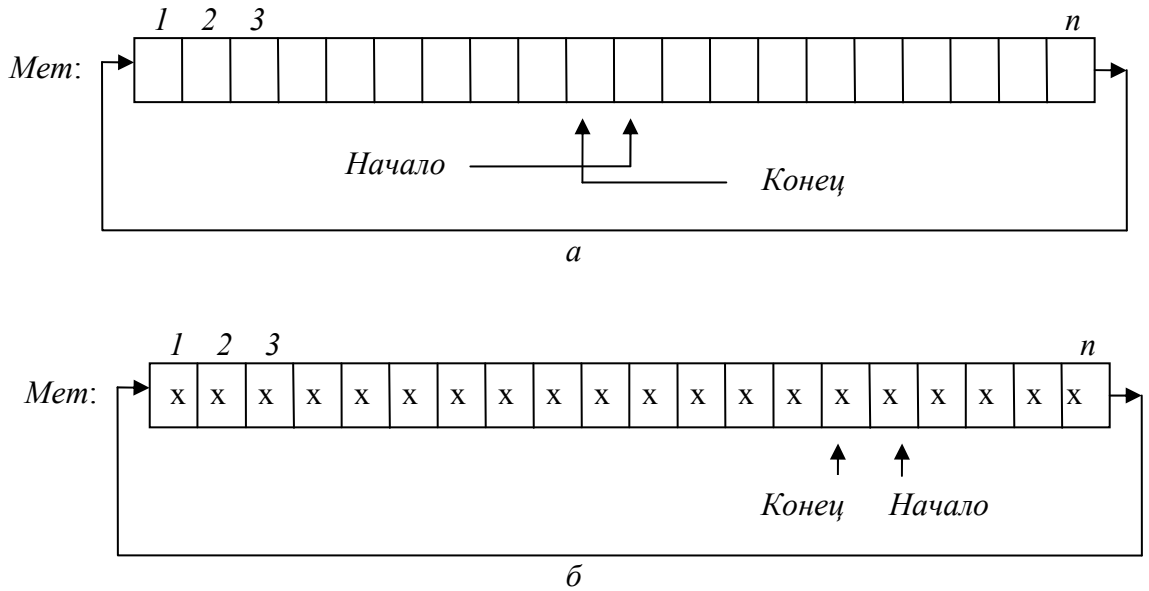


Рис. 2.6. Два состояния очереди, при которых Начало=Конец+1:
а – очередь пуста, б – очередь полна

Аналогичным образом может быть реализован *дек*.

2.3. Упражнения

В заданиях 1 – 3 следует использовать очередь и операции над ней; при этом очередь может быть реализована как на базе вектора, так и в связанной памяти (ссылочная реализация).

1. За один просмотр заданного файла *F* (типа **file of** *Real*) и без использования дополнительных файлов вывести элементы файла *F* в следующем порядке: сначала - все числа, меньшие *a*, затем - все числа на отрезке [*a , b*], и наконец - все остальные числа, сохраняя исходный взаимный порядок в каждой из этих групп чисел (*a* и *b* задаются пользователем, *a<b*).

2. Содержимое заданного текстового файла *F*, разделенного на строки, переписать в текстовый файл *G*, перенося при этом в конец каждой строки все входящие в нее цифры (с сохранением исходного взаимного порядка как среди цифр, так и среди остальных литер строки).

3. Рассматриваются следующие типы данных:

type *имя* = (*Анна , ... , Яков*);
дети = **array** [*имя , имя*] of *boolean*;
потомки = **file of** *имя*.

Задан массив *Д* типа *дети* (*Д* [*x* , *y*] = *true*, если человек по имени *y* является ребенком человека по имени *x*). Для введенного пользователем имени *И* записать в файл *П* типа *потомки* имена всех потомков человека с именем *И* в следующем порядке: сначала - имена всех его детей, затем - всех его внуков, затем - всех правнуков и т.д.

В заданиях 4 – 8 следует использовать стек и операции над ним; при этом стек может быть реализован как на базе вектора, так и в связанной памяти (ссылочная реализация).

4. Содержимое заданного текстового файла *F*, разделенного на строки, переписать в текстовый файл *G*, выписывая литеры каждой строки в обратном порядке.

5. Правильная скобочная конструкция с тремя видами скобок определяется как

< текст > ::= < пусто > | < элемент > < текст >
< элемент > ::= < символ > | (< текст >) | [< текст >] | { < текст > }

где < символ > - любой символ, кроме (,) , [,] , { , }. Проверить, является ли текст, содержащийся в заданном файле *F*, правильной скобочной конструкцией; если нет, то указать номер ошибочной позиции.

6. Проверить, является ли содержимое заданного текстового файла F правильной записью формулы следующего вида:

< формула > ::= < терм > | < терм > + < формула > | < терм > - < формула >
< терм > ::= < имя > | (< формула >) | [< формула >] | { < формула > }
< имя > ::= *x* | *y* | *z*

Если не является, то указать номер ошибочной позиции.

7. В заданном текстовом файле *F* записана формула вида

< формула > ::= < цифра > | *M* (< формула > , < формула >) |
m (< формула > , < формула >)
< цифра > ::= 0 | 1 | ... | 9

где *M* обозначает функцию *max*, а *m* – функцию *min*. Вычислить (как целое число) значение данной формулы. Например, *M* (5 , *m* (6 , 8)) = 6.

8. В заданном текстовом файле *F* записано логическое выражение (ЛВ) в следующей форме:

<ЛВ>::= *true* | *false* | (¬< ЛВ >) | (< ЛВ > ∧ < ЛВ >) | (< ЛВ > ∨ < ЛВ >)

где знаки ¬ , ∧ и ∨ обозначают соответственно отрицание, конъюнкцию и дизъюнкцию. Вычислить (как *Boolean*) значение этого выражения.

В заданиях 9 – 11 следует использовать очередь и/или стек и операции над ними.

9. В заданном текстовом файле *F* записан текст, сбалансированный по круглым скобкам:

< текст > ::= < пусто > | < элемент > < текст >

< элемент > ::= < символ > | (< текст >)

где < символ > – любой символ, кроме (,). Для каждой пары соответствующих открывающей и закрывающей скобок вывести номера их позиций в тексте, упорядочив пары в порядке возрастания номеров позиций:

а) закрывающих скобок;

б) открывающих скобок.

Например, для текста *A* + (45 - *F* (*X*) * (*B* - *C*)) надо напечатать:

а) 8 10; 12 16; 3 17;

б) 3 17; 8 10; 12 16.

10. Определить, имеет ли заданная в файле *F* символьная строка следующую структуру:

a D b D c D d ...,

где каждая строка *a*,*b*,*c*,*d*,..., в свою очередь, имеет вид *x*1 *C* *x*2, где *x*1 есть строка, состоящая из символов *A* и *B*, а *x*2 - строка, обратная строке *x*1 (т.е. если *x*1 = *ABABB*, то *x*2 = *BBABA*). Таким образом, исходная строка состоит только из символов *A*,*B*,*C* и *D*. Исходная строка может читаться только последовательно (посимвольно) слева направо.

11. Рассматривается выражение следующего вида:

< выражение >::= < терм > | < терм > + < выражение > |
< терм > - < выражение >

< терм > ::= < множитель > | < множитель > * < терм >

< множитель > ::= < число > | < переменная > | (< выражение >) |
< множитель > ^ < число >

< число > ::= < цифра >

< переменная > ::= < буква >

Такая форма записи выражения называется ***инфиксной***.

Постфиксной (префиксной) формой записи выражения *aDb* называется запись, в которой знак операции размещен за (перед) операндами: *abD* (*Dab*).

Примеры

Инфиксная	***Постфиксная***	***Префиксная***
a-b	*ab-*	*-ab*
*a*b+c*	*ab*c+*	*+*abc*
a(b+c)*	*abc+**	**a+bc*
*a+b^c^d*e*	*abc^d^e*+*	*+a*^b^cde*.

Отметим, что постфиксная и префиксная формы записи выражений не содержат скобок.

Требуется:

а) вычислить как целое число значение выражения (без переменных), записанного в постфиксной форме в заданном текстовом файле *postfix*;

б) то же для выражения в префиксной форме (задан текстовый файл *prefix*);

в) перевести выражение, записанное в обычной (инфиксной) форме в заданном текстовом файле *infix*, в постфиксную форму и в таком виде записать его в текстовый файл *postfix*;

г) то же, но в префиксную форму (записать в файл *prefix*);

д) вывести в обычной (инфиксной) форме выражение, записанное в постфиксной форме в заданном текстовом файле *postfix* (рекурсивные процедуры не использовать и лишние скобки не выводить);

е) то же, но при заданной префиксной форме (в файле *prefix*).

3. ДЕРЕВЬЯ

Наиболее полезной нелинейной структурой данных является дерево (или лес).

3.1. Определения дерева, леса, бинарного дерева. Скобочное представление

Дадим формальное определение *дерева*, следуя [10].

Дерево – конечное множество T, состоящее из одного или более узлов, таких, что

а) имеется один специально обозначенный узел, называемый *корнем* данного дерева;

б) остальные узлы (исключая корень) содержатся в $m \geq 0$ попарно не пересекающихся множествах T_1 , T_2 , ... , T_m, каждое из которых, в свою очередь, является деревом. Деревья T_1 , T_2 , ... , T_m называются *поддеревьями* данного дерева.

При программировании и разработке вычислительных алгоритмов удобно использовать именно такое *рекурсивное* определение, поскольку рекурсивность является естественной характеристикой этой структуры данных.

Каждый узел дерева является корнем некоторого поддерева. В том случае, когда множество поддеревьев такого корня пусто, этот узел называется *концевым узлом*, или *листом*. *Уровень* узла определяется рекурсивно следующим образом: 1) корень имеет уровень 1; 2) другие узлы имеют уровень, на единицу больший их уровня в содержащем их поддереве этого корня. Используя для уровня узла «a» дереве T обозначение *уровень* (a , T), можно записать это

определение в виде

$$\textit{уровень}(a, T) = \begin{cases} 1, & \text{если } a - \text{корень дерева } T \\ \textit{уровень}(a, T_i) + 1, & \text{если } a - \text{не корень дерева } T \end{cases}$$

где T_i – поддерево корня дерева T, такое, что $a \in T_i$.

Говорят, что каждый корень является *отцом* корней своих поддеревьев и что последние являются *сыновьями* своего отца и *братьями* между собой. Говорят также, что узел n – *предок* узла m (а узел m – *потомок* узла n), если n – либо отец m, либо отец некоторого предка m.

Если в определении дерева существен порядок перечисления поддеревьев $T_1, T_2, \ldots, T_m$, то дерево называют *упорядоченным* и говорят о "первом" (T_1), "втором" (T_2) и т.д. поддеревьях данного корня. Далее будем считать, что все рассматриваемые нами деревья являются упорядоченными, если явно не оговорено противное. Отметим также, что в терминологии теории графов определенное ранее упорядоченное дерево более полно называлось бы "конечным ориентированным (корневым) упорядоченным деревом".

Лес – это множество (обычно упорядоченное), состоящее из некоторого (быть может, равного нулю) числа непересекающихся деревьев. Используя понятие леса, пункт *б* в определении дерева можно было бы сформулировать так: "*узлы дерева, за исключением корня, образуют лес*".

Традиционно дерево изображают графически, например так, как на рис.3.1.

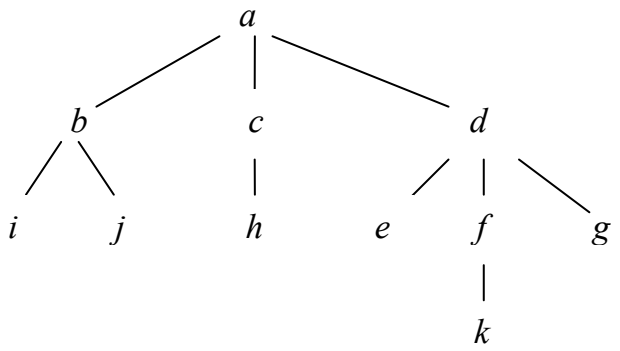


Рис. 3.1. Графическое изображение дерева

В графическом представлении для изображения структуры дерева существенно используется двухмерность рисунка. При машинной обработке часто удобнее использовать текстовое представление дерева. Например, таким представлением может быть так называемый уступчатый список. Здесь "двухмерность" проявляется за счет того, что текст разбит на строки и фиксируется позиция символа узла в строке. На рис.3.2, *а*, *б* представлено в виде уступчатого списка дерево, изображенное на рис. 3.1.

Другой вид текстового (и принципиально одномерного) представления дерева - это так называемая "скобочная запись" (ср. с записью иерархических списков в разд.1.7). Определим скобочное представление дерева и леса:

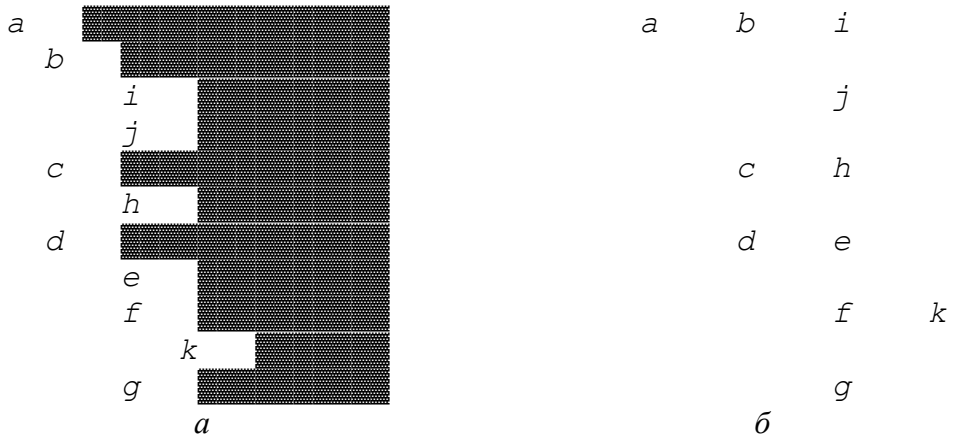


Рис. 3.2. Представление дерева: а – в виде уступчатого списка; б – в виде "упрощенного" уступчатого списка

< лес > ::= пусто | < дерево > < лес >,
< дерево > ::= (< корень > < лес >).

В скобочном представлении дерево, изображенное на рис. 3.1, запишется как

(*a* (*b* (*i*) (*j*)) (*c* (*h*)) (*d* (*e*) (*f* (*k*)) (*g*))).

Наиболее важным типом деревьев являются *бинарные деревья*. Удобно дать следующее формальное определение. *Бинарное дерево* – конечное множество узлов, которое либо пусто, либо состоит из корня и двух непересекающихся бинарных деревьев, называемых правым поддеревом и левым поддеревом. Так определенное бинарное дерево *не* является частным случаем дерева. Например, бинарные деревья, изображенные на рис.3.3, различны между собой, так как в одном случае корень имеет пустое правое поддерево, а в другом случае правое поддерево непусто. Если же их рассматривать как деревья, то они идентичны.

Рис. 3.3. Бинарные деревья из двух узлов

Определим скобочное представление бинарного дерева (БД):

< БД > ::= < пусто > | < непустое БД >,
< пусто > ::= Λ,
< непустое БД > ::= (< корень > < БД > < БД >).

Здесь пустое дерево имеет специальное обозначение – Λ.

Например, бинарное дерево, изображенное на рис.3.4, имеет скобочное представление

(*a* (*b* (*d* Λ (*h* Λ Λ)) (*e* Λ Λ)) (*c* (*f* (*i* Λ Λ)(*j* Λ Λ)) (*g* Λ (*k* (*l* Λ Λ) Λ)))).

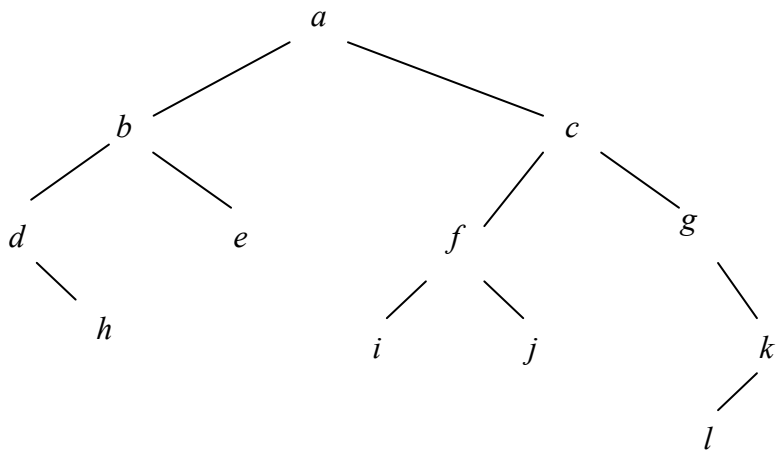


Рис. 3.4. Бинарное дерево

Можно упростить скобочную запись бинарного дерева, исключив "лишние" знаки Λ по правилам:

1) (< корень > < непустое БД > Λ) ≡ (< корень > < непустое БД >),

2) (< корень > Λ Λ) ≡ (< корень >).

Тогда, например, скобочная запись бинарного дерева, изображенного на рис.3.4, будет иметь вид

(*a* (*b* (*d* Λ *h*) (*e*)) (*c* (*f* (*i*) (*j*)) (*g* Λ (*k* (*l*))))).

3.2. Спецификация дерева, леса, бинарного дерева

Рассмотрим функциональную спецификацию структуры данных дерева с узлами типа□ : *Tree of* α = *Tree* (α). При этом лес деревьев *Forest* (*Tree* (α)) определим как *L_list* (*Tree* (α)) через уже известную структуру линейного списка *L_list* с базовыми функциями *Cons*, *Head*, *Tail*, *Null* (см.1.6). Базовые операции с деревом задаются набором функций:

1) *Root* : *Tree* → α;

2) *Listing* : *Tree* → *Forest*;

3) *ConsTree* :□ α ⊗ *Forest* → *Tree*

и аксиомами (∀*u:* α; ∀*f: Forest* (*Tree* (α)); ∀*t*: *Tree* (α)):

A1) *Root* (*ConsTree* (*u* , *f*)) = *u*;

A2) *Listing* (*ConsTree* (*u* , *f*)) = *f*;

A3) *ConsTree* (*Root* (*t*) , *Listing* (*t*)) = *t*.

Здесь функции *Root* и *Listing* - селекторы: *Root* выделяет корень дерева, а *Listing* выделяет лес поддеревьев корня данного дерева. Конструктор *ConsTree* порождает дерево из заданных узла и леса деревьев.

Тот факт, что структура данных *Forest* явно определена через *L_list* (*Tree*), позволяет реализовать структуру дерева (леса) на базе другой

структуры данных, а именно, на базе иерархических списков. Достаточно рассматривать при этом описанное в 3.1 скобочное представление дерева как S-выражение специальной структуры. Возможно и другое удобное представление дерева (леса), основанное на некотором соответствии леса и бинарного дерева, описанном далее в 3.3.

Рассмотрим функциональную спецификацию структуры данных бинарного дерева с узлами типа α: *BinaryTree* (α) ≡ *BT* (α). Здесь важно различать ситуации обработки пустого и непустого бинарного дерева, поскольку некоторые операции определяются только на непустых бинарных деревьях. Далее считаем, что значение типа *BT* есть либо Λ (пустое бинарное дерево), либо значение типа *NonNullBT*. Тогда базовые операции типа *BT* (α) задаются набором функций:

1) *Root*: *NonNullBT* → α;
2) *Left*: *NonNullBT* → *BT*;
3) *Right*: *NonNullBT* → *BT*;
4) *ConsBT*: α ⊗ *BT* ⊗ *BT* → *NonNullBT*;
5) *Null*: *BT* → *Boolean*;
6) Λ : → *BT*

и набором аксиом (∀ *u*: α, *b*: *NonNullBT* (α), *b*1, *b*2: *BT* (α)):

A1) *Null* (Λ) = *true*;
A1') *Null* (*b*) = *false*;
A2) *Null* (*ConsBT* (*u* , *b*1 , *b*2)) = *false*;
A3) *Root* (*ConsBT* (*u* , *b*1 , *b*2)) = *u*;
A4) *Left* (*ConsBT* (*u* , *b*1 , *b*2)) = *b*1;
A5) *Right* (*ConsBT* (*u* , *b*1 , *b*2)) = *b*2;
A6) *ConsBT* (*Root* (*b*) , *Left* (*b*) , *Right* (*b*)) = *b*.

Здесь функции *Root*, *Left* и *Right* – селекторы: *Root* выделяет корень бинарного дерева, а *Left* и *Right* – его левое и правое поддеревья соответственно. Конструктор *ConsBT* порождает бинарное дерево из заданных узла и двух бинарных деревьев. Предикат *Null* – индикатор, различающий пустое и непустое бинарные деревья.

3.3. Каноническое соответствие бинарного дерева и леса

Бинарные деревья особенно полезны, в том числе потому, что существует естественное взаимно однозначное соответствие между лесами и бинарными деревьями, и многие операции над лесом (деревом) могут быть реализованы как соответствующие операции над бинарным деревом, представляющим этот лес (дерево).

Опишем такое представление леса бинарным деревом (и далее будем называть его *каноническим*). Пусть лес *F* типа *Forest* задан как список деревьев T_i типа *Tree* (для $\forall i \in 1..m$):

$$F = (T_1\ T_2\ ...\ T_m).$$

Тогда *Head* (*F*) – первое дерево T_1 леса *F*, а *Tail* (*F*) – лес остальных деревьев (T_2 ... T_m). Если далее в дереве *Head* (*F*) выделить корень *Root* (*Head* (*F*)) и лес поддеревьев *Listing* (*Head* (*F*)),то исходный лес *F* представляется как совокупность трех частей:

1) корня первого дерева – *Root* (*Head* (*F*)),

2) леса поддеревьев первого дерева – *Listing* (*Head* (*F*)),

3) леса остальных деревьев – *Tail* (*F*).

Из этих трех частей рекурсивно порождается бинарное дерево *B* (*F*), представляющее лес *F*:

B (*F*) ≡ **if** *Null* (*F*) **then** Λ
else *ConsBT* (*Root* (*Head* (*F*)),
B (*Listing* (*Head* (*F*))),
B (*Tail* (*F*)))

Согласно этому определениею корнем бинарного дерева *B* (*F*) является корень первого дерева T_1 в лесу *F*, левым поддеревом бинарного дерева *B* (*F*) является бинарное дерево, представляющее лес поддеревьев первого дерева T_1, а правым поддеревом бинарного дерева *B* (*F*) является бинарное дерево, представляющее лес остальных (кроме первого) деревьев исходного леса *F*.

Это формальное определение имеет следующую наглядную интерпретацию. Рассмотрим, для примера, лес из двух деревьев, представленный на рис. 3.5.

Ошибка!

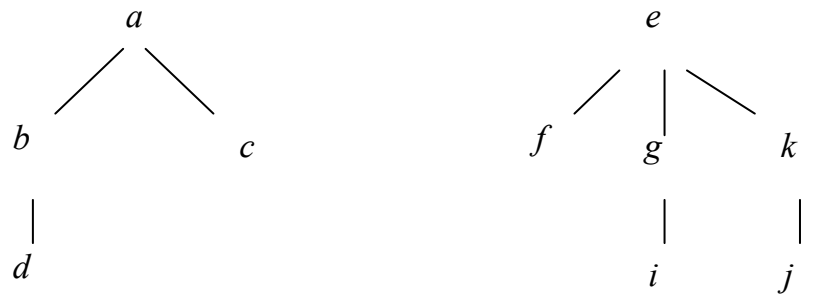


Рис. 3.5. Лес из двух деревьев

Представляющее этот лес бинарное дерево можно получить, если соединить последовательно сыновей каждой семьи и убрать все вертикальные связи, кроме связей, идущих от отцов к их первым сыновьям, как это показано на рис. 3.6.

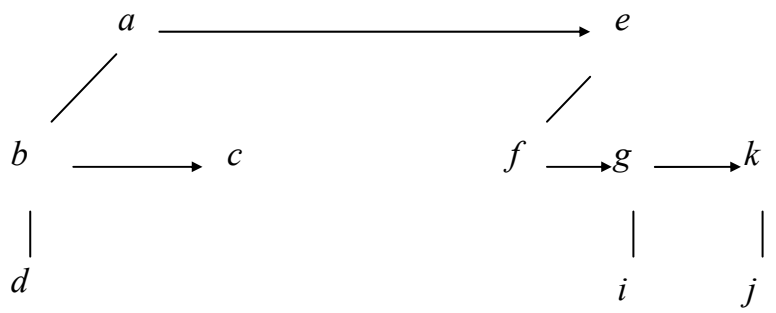


Рис. 3.6. Преобразованный лес

Повернув затем изображение леса из рис. 3.6 на 45^{o}, получаем бинарное дерево, изображенное на рис. 3.7.

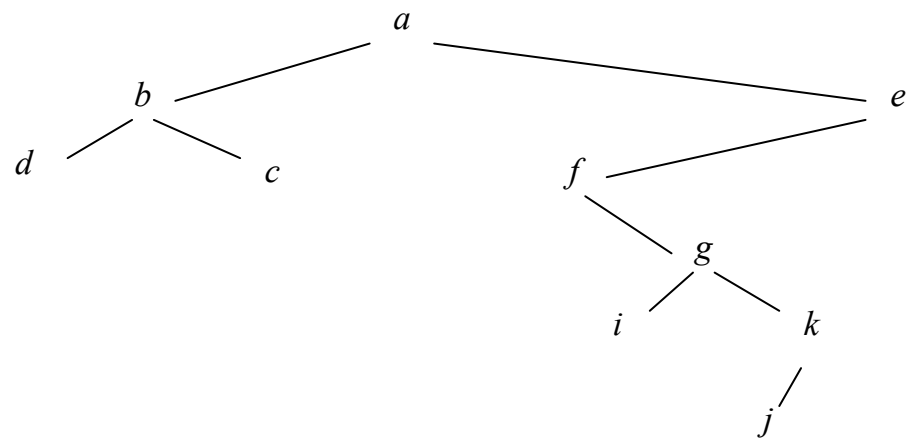


Рис. 3.7. Бинарное дерево, соответствующее лесу из рис. 3.5

Легко видеть, что, выполняя эти действия в обратном порядке, мы получим единственный лес, соответствующий данному бинарному дереву. Это обратное преобразование формально описывается следующим образом (здесь *F* – лес типа *Forest*, а *B* – бинарное дерево типа *BT*):

F (*B*) ≡ **if** *Null* (*B*) **then** *NIL*
else *Cons* (*ConsTree* (*Root* (*B*) , *F* (*Left* (*B*))),
F (*Right* (*B*))).

Согласно этому определению первое дерево T_1 леса *F* образуется из корня бинарного дерева *B* и леса поддеревьев, полученного из левого поддерева бинарного дерева *B*. Остальные деревья (T_2 ... T_m) леса $F = (T_1\ T_2 \ldots T_m)$ составляют лес, полученный из правого поддерева бинарного дерева *B*.

3.4. Обходы бинарных деревьев и леса

Многие алгоритмы работы с бинарными деревьями основаны на последовательной (в определенном порядке) обработке узлов дерева. В этом случае говорят об обходе (прохождении) бинарного дерева. Такой обход порождает

определенный порядок перечисления узлов бинарного дерева. Выделяют несколько стандартных вариантов обхода. Будем именовать их в зависимости от того порядка, в котором при этом посещаются корень дерева и узлы левого и правого поддеревьев. Например, при КЛП-обходе сначала посещается корень, затем обходятся в КЛП-порядке последовательно левое и правое поддеревья. Приведем рекурсивные процедуры КЛП-, ЛКП- и ЛПК-обходов, прямо соответствующие их рекурсивным определениям (операция обработки узла обозначена как “посетить(узел)”):

```
procedure обходКЛП ( b: BTree ); {прямой}
begin
  if not Null ( b )
  then begin
       посетить  (  Root  ( b ) );
       обходКЛП  (  Left  ( b ) );
       обходКЛП  (  Right  ( b ) );
  end
end{обходКЛП};
```

```
procedure обходЛКП ( b : BTree );
{обратный}
begin
  if not Null ( b ) then
  begin
    обходЛКП ( Left ( b ) );
    посетить ( Root ( b ) );
    обходЛКП ( Right ( b ) );
  end
end{обходЛКП};
```

```
procedure обходЛПК ( b : BTree );
 {концевой}
begin
  if not Null ( b ) then
  begin
    обходЛПК ( Left ( b ) );
    обходЛПК ( Right ( b ) );
    посетить ( Root ( b ) );
end
end{обходЛПК}
```

Следует обратить внимание на то, что в литературе используются различная терминология при именовании видов обхода бинарного дерева. Перечислим наиболее популярные варианты терминологии (виды обходов перечислены во всех вариантах в одном и том же порядке):

1) КЛП, ЛКП, ЛПК [14];
2) прямой, обратный, концевой [10];
3) прямой, симметричный, обратный [13];
4) сверху вниз, слева направо, снизу вверх [5,6];
5) префиксный (*PreOrder*), инфиксный (*InOrder*), постфиксный (*PreOrder*) [5,6].

Иногда обход КЛП называют обходом в глубину. В некоторых случаях используется смешанная терминология [16]. Будем придерживаться далее

вариантов терминологии 1 и 2. В варианте 2 названия обходам даны соответственно тому, что в КЛП-порядке корень посещается перед посещением узлов левого поддерева (прямой порядок), а в ЛКП-порядке корень посещается после обхода узлов левого поддерева (обратный порядок). В ЛПК-порядке корень посещается после обхода узлов левого и правого поддеревьев (концевой порядок). Такая терминология основана на важной роли левого поддерева в каноническом соответствии бинарного дерева и леса (см. 3.3).

Терминология варианта 5 явно связана с обходом бинарного дерева, представляющего арифметическое выражение с бинарными операциями. Пусть, например, дано арифметическое выражение

$$(a + b) * c - d / (e + f * g) .$$

На рис. 3.8 представлено соответствующее ему бинарное дерево.

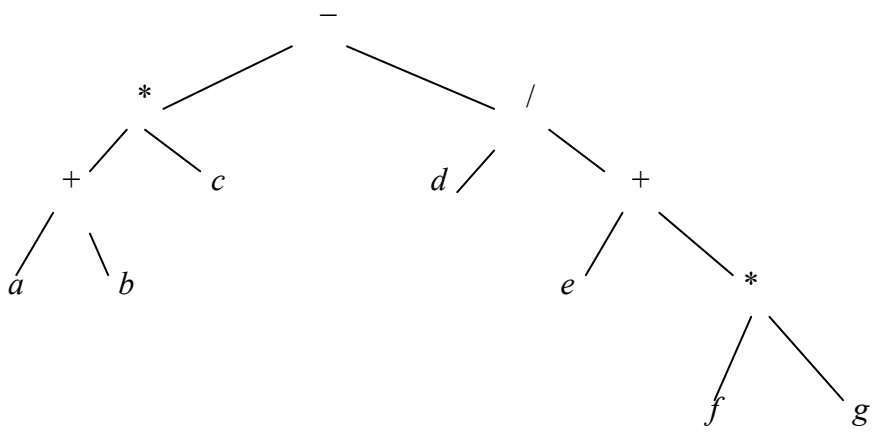


*Рис. 3.8. Бинарное дерево, представляющее арифметическое выражение (a + b) * c – d / (e + f * g)*

Тогда три варианта обхода этого дерева порождают три известные формы записи арифметического выражения:

1) КЛП – префиксную запись

$$– * + a b c / d + e * f g ;$$

2) ЛКП – инфиксную запись (без скобок, необходимых для задания последовательности выполнения операций)

$$a + b * c – d / e + f * g ;$$

3) ЛПК – постфиксную запись

$$a b + c * d e f g * + / – .$$

В качестве упражнения полезно дать интерпретацию и остальным вариантам названий обходов.

Нерекурсивные процедуры обхода бинарных деревьев

Учитывая важность эффективной реализации обходов бинарных деревьев, рассмотрим *нерекурсивные* процедуры обходов. Общий нерекурсивный алго-

ритм для всех трех порядков обхода использует стек *S* для хранения упорядоченных пар (*p* , *n*), где *p* − узел бинарного дерева, а *n* - номер операции, которую надо применить к *p*, когда пара (*p* , *n*) будет выбрана из стека. Операции “посетить корень”, “пройти левое поддерево”, “пройти правое поддерево” нумеруются числами 1,2,3 в зависимости от порядка их выполнения в данном варианте обхода. В алгоритме эти операции будут реализованы следующим образом:

посетить корень – посетить (*p*);

пройти левое поддерево -

if not *Null* (*Left* (*p*)) **then** S ← (*Left* (*p*) , 1);

пройти правое поддерево -

if not *Null* (*Right* (*p*)) **then** S ← (*Right* (*p*) , 1).

Здесь и далее для краткости и наглядности использованы следующие обозначения операций со стеком: *S* ← *e* вместо *S* := *Push* (*e* , *S*) и *e* ← *S* вместо *Pop*2 (*e* , *S*).

Тогда общий алгоритм имеет вид:

```
procedure обход ( b: BTree );
   var S: Stack of  ( BTree , operation ) ;
        p: BTree;       op: operation { =1..3};
begin
   S:= Create;  S ← ( b , 1);
   while not Null ( S ) do
   begin ( p , op )  ← S;
      if    op = 1    then begin S ← ( p , 2); операция 1 end
      else if op = 2 then begin S ← ( p , 3); операция 2 end
      else {  op = 3 }                          операция 3
   end{while}
end{обход}
```

В случае КЛП-обхода можно существенно упростить алгоритм, исключая лишние для этого варианта манипуляции со стеком. Здесь нет необходимости хранить в стеке номер операции. Итак, конкретизация (с упрощением) общего алгоритма для КЛП-обхода имеет вид:

```
procedure обход_КЛП ( b: BTree ); {прямой}
   var S: Stack of BTree;  p: BTree;
begin
     S:= Create; S ← b;
     while not Null ( S ) do
     begin
        p ← S;
        посетить ( p );
        if not Null ( Right ( p ) )  then S ← Right ( p );
        if not Null ( Left ( p ) )   then S ← Left ( p )
     end{while}
end{обход_КЛП}
```

В случае ЛКП-обхода также возможно некоторое упрощение – в стеке сохраняются указания лишь на операции 1 или 2:

```
procedure обход_ЛКП ( b: BTree ); {обратный}
   var S: Stack of ( BTree , operation );
       p: BTree; op: operation { =1..2};
begin  S:= Create; S ← ( b , 1);
   while not Null ( S ) do
   begin ( p , op ) ← S;
      if op = 1 then
      begin
         S ← ( p , 2) ;  if  not Null ( Left ( p ) )  then S ← ( Left ( p ) , 1 )
      end
      else {op=2}
      begin
         посетить ( p );  if not Null ( Right ( p ) )  then S ← ( Right ( p ) , 1 )
      end
end{while}
end{обход_ЛКП}
```

Конкретизация общего алгоритма для ЛПК-обхода (здесь нет упрощений) имеет вид:

```
procedure обход_ЛПК ( b: BTree ); {концевой}
   var S: Stack of ( BTree , operation );
       p: BTree;       op: operation { =1..3};
begin  S:= Create; S ← ( b, 1);
   while not Null ( S ) do
   begin
      ( p , op ) ← S;
      if op = 1 then
      begin
         S ← ( p , 2) ;  if not Null ( Left ( p ) ) then S ← ( Left ( p) , 1 )
      end
      else if op = 2 then
      begin
         S ← ( p, 3); if not Null (Right ( p ) ) then S ← ( Right ( p ) , 1 )
      end else { op=3 } посетить ( p )
   end{while}
end{обход_ЛПК}
```

Еще один полезный способ обхода бинарного дерева – обход в *горизонтальном* порядке (в ширину). При таком способе узлы бинарного дерева проходятся слева направо, уровень за уровнем от корня вниз (поколение за поколением от старших к младшим). Легко указать нерекурсивную процедуру горизонтального обхода, аналогичную процедуре КЛП-обхода (в глубину), но использующую *очередь* вместо стека:

```
procedure обход_горизонтальный ( b: BTree );
   var Q: queue of BTree;
       p: BTree;
begin
    Q := Create;
    Q ← b;
    while not Null ( Q ) do
    begin
       p ← Q;
       посетить ( p );
       if not Null ( Left ( p ) ) then Q ← Left ( p );
       if not Null ( Right ( p ) ) then Q ← Right ( p )
    end{while}
end{обход_горизонтальный}
```

Обходы леса

Следуя естественному (каноническому) соответствию между бинарными деревьями и лесами, можно на основе КЛП-, ЛКП- и ЛПК-обходов бинарного дерева получить три соответствующие порядка прохождения леса (и, следовательно, произвольного дерева).

Прямой порядок:
а) посетить корень первого дерева;
б) пройти поддеревья первого дерева (в прямом порядке);
в) пройти оставшиеся деревья (в прямом порядке).

Обратный порядок:
а) пройти поддеревья первого дерева (в обратном порядке);
б) посетить корень первого дерева;
в) пройти оставшиеся деревья (в обратном порядке).

Концевой порядок:
а) пройти поддеревья первого дерева (в концевом порядке);
б) пройти оставшиеся деревья (в концевом порядке);
б) посетить корень первого дерева.

При необходимости применить какой-либо из этих обходов к лесу (дереву) можно сначала построить бинарное дерево, представляющее этот лес, а затем применить соответствующий обход бинарного дерева.

3.5. Представления и реализации бинарных деревьев

Рассмотрим варианты представления и реализации структуры данных бинарного дерева. Пусть базовый тип узлов есть *Elem*.

Ссылочная реализация бинарного дерева в связанной памяти основана на представлении типа *BT* (*Elem*) рекурсивными типами *BinT* и *Node*:

```
type
   BinT = ^Node;              { представление бин.дерева }
   Node = record              { узел :                   }
            Info : Elem;      {        = содержимое      }
            LSub : BinT;      {        = левое  поддерево }
            RSub : BinT       {        = правое поддерево }
          end { Node }
```

Здесь каждый узел дерева рассматривается как корень соответствующего поддерева, и этому поддереву сопоставляется запись из трех полей: поле *Info* хранит значение корня (типа *Elem*), а поля *LSub* и *RSub* – указатели на левое и правое поддеревья. Пустому дереву сопоставляется константа *NilBT* (на абстрактном уровне обозначаемая ранее как Λ). На рис.3.9, *а*, *б* изображены бинарное дерево и его представление в ссылочной реализации.

Интерфейсная часть модуля для работы с бинарным деревом на основе ссылочной реализации представлена на рис. 3.10.

Здесь в сравнении с функциональной спецификацией из 3.2 добавлены функция *CreateBT* и процедура *DestroyBT*, используемые для начала и завершения работы с экземпляром динамической структуры. Функция *CreateBT* формально специфицируется соотношениями

$$CreateBT: \rightarrow BT; \qquad Null\ (\ CreateBT\ ) = True.$$

Кроме того, добавлена процедура *Otkaz*, вызываемая при попытке некорректного применения основных функций.

Тип узлов дерева *Elem* должен быть задан в модуле *GlobalBT*, например, таким образом:

```
Unit GlobalBT;
Interface type Elem = Char;
Implementation
begin
end.
```

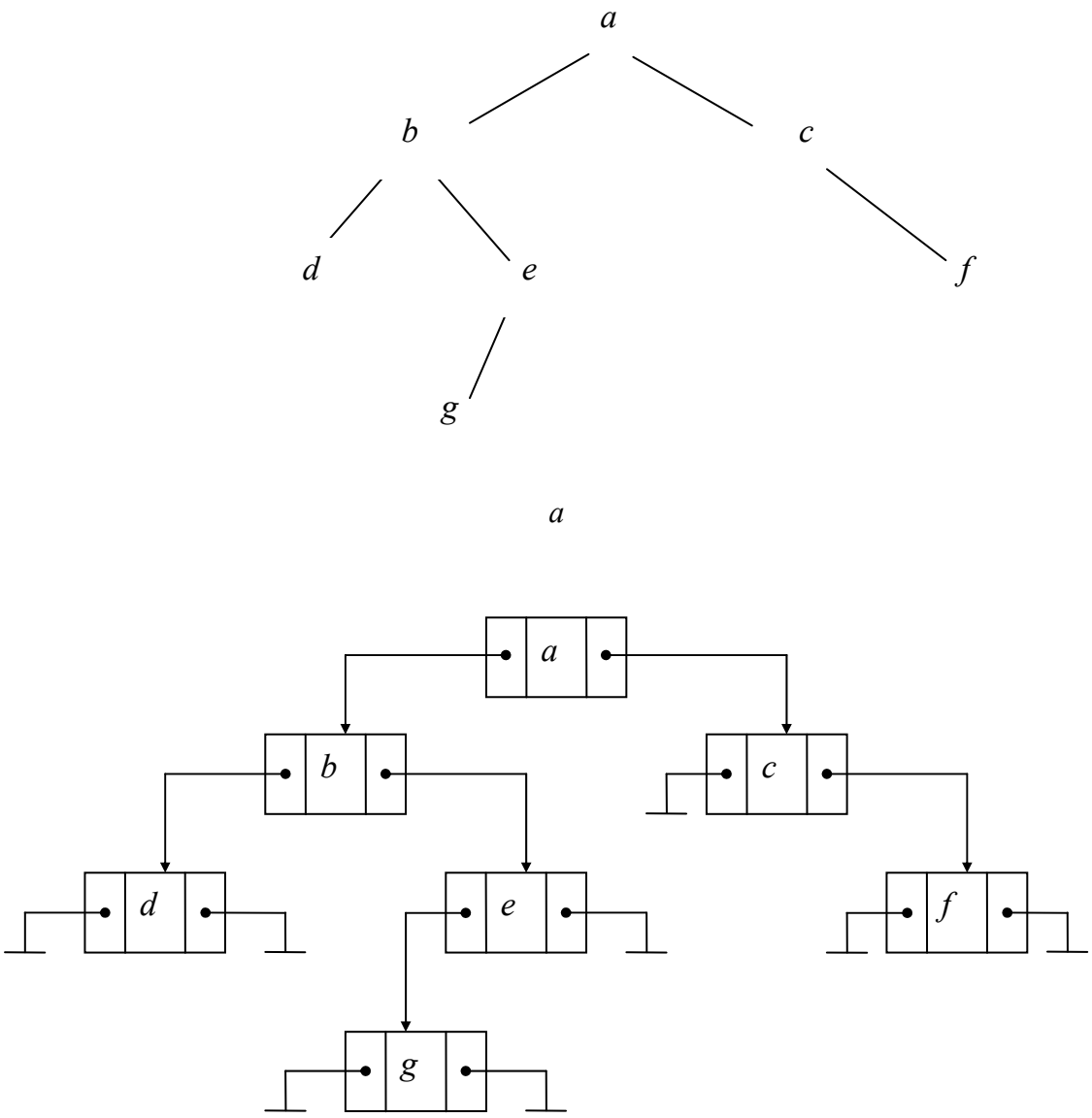


Рис. 3.9. Пример бинарного дерева (а) и его ссылочного представления (б)

```
{ Модуль для работы с бинарными деревьями }
Unit BinTree;
 Interface
   uses GlobalBT;
   const
       NilBT = nil;
   type
       BinT = ^Node;              { представление бинарного дерева  }
                                  { тип Elem описан в GlobalBT      }
       Node = record              { узел :                          }
                   Info : Elem;   {      = содержимое               }
                   LSub : BinT;   {      = левое  поддерево         }
                   RSub : BinT    {      = правое поддерево         }
                end { Node };
function CreateBT : BinT;
function NullBT ( t : BinT ) : Boolean;
function RootBT ( t : BinT ) : Elem;
function LeftBT ( t : BinT ) : BinT;
function RightBT ( t : BinT ) : BinT;
function ConsBT ( e : Elem; LS, RS : BinT ) : BinT;
procedure DestroyBT ( var b : BinT );
procedure Otkaz ( n : Byte );
{-------------------------------------------------------------------------------}
Implementation
...
end.
```

Рис. 3.10. Модуль для работы с бинарными деревьями

Часть **Implementation** модуля *BinTree* может иметь следующий вид:

```
Implementation
procedure Otkaz ( n : Byte );
begin
 Case n of
     1 : Write ( 'ОТКАЗ : RootBT( Null_Bin_Tree ) !');
     2 : Write ( 'ОТКАЗ : LeftBT( Null_Bin_Tree ) !');
     3 : Write ( 'ОТКАЗ : RightBT( Null_Bin_Tree ) !');
     4 : Write ( 'ОТКАЗ : исчерпана память !')
     else  Write ( 'ОТКАЗ : ?')
 end;
 Halt
end { Otkaz };

function CreateBT : BinT;
begin
     CreateBT := nil
end { CreateBT };
```

```
function NullBT ( t : BinT ) : Boolean;
begin
     NullBT := ( t = nil )
end { NullBT };

function RootBT ( t : BinT ) : Elem;
begin
    if t <> nil then RootBT :=  t^.Info else Otkaz(1)
end { RootBT };

function LeftBT ( t : BinT ) : BinT;
begin
   if t <> nil then LeftBT := t^.LSub  else Otkaz(2)
end { LeftBT };

function RightBT ( t : BinT ) : BinT;
begin
   if t <> nil then RightBT := t^.RSub else Otkaz(3)
end { RightBT };

function ConsBT ( e : Elem; LS, RS : BinT ) : BinT;
 var b : BinT;
begin
     if MaxAvail >= SizeOf ( Node ) then
      begin
       New ( b );
       b^.Info := e;
       b^.LSub := LS;
       b^.RSub := RS;
       ConsBT := b
      end
     else Otkaz(4)
end { ConsBT };

procedure DestroyBT( var b : BinT );
begin
    if b <> nil then
      begin
        DestroyBt ( b^.LSub );
        DestroyBt ( b^.RSub );
        Dispose ( b )
      end
end { DestroyBT };
begin
end.
```

Далее приведем пример использования модуля *BinTree*, в котором для ввода бинарного дерева из файла и вывода его на экран используется КЛП-обход бинарного дерева, описанный в 3.4.

```
{ Пример работы с бинарными деревьями }
Uses BinTree;
var b : BinT;
    Fin : Text;
function EnterBt : BinT;
 { ввод узлов в КЛП-порядке и построение бинарного дерева }
 var c : Char;
begin
    Read ( Fin, c );
    if c = '/'
     then EnterBT := NilBT  {Create}
     else EnterBT := ConsBT ( c, EnterBT, EnterBT )
end { EnterBT };

procedure OutBT( b : BinT );
 { вывод узлов бинарного дерева в КЛП-порядке }
begin
 if not NullBT ( b ) then
  begin
     Write ( RootBT ( b ) );
     OutBT ( LeftBT ( b ) );
     OutBT ( RightBT ( b ) )
  end
 else Write ('/')
end { OutBT };

procedure DisplayBT ( b : BinT );
 { вывод построчного и повернутого изображения бин. дерева}
begin
 if NullBt ( b ) then  { ? }
    else
     begin
        Write ( RootBT ( b ) );
        if not NullBT ( RightBT ( b ) ) then
           begin
            Write (' ');              { вправо }
            DisplayBT ( RightBT ( b ) );
            Write (#8); Write (#8);     { возврат влево }
           end;
        if not NullBT ( LeftBT ( b ) ) then
           begin
```

```
            Write (#10);              { вниз   }
            Write (' ');              { вправо }
            DisplayBT ( LeftBT ( b ) );
            Write (#8);  Write (#8);     { возврат влево }
           end;
     end {else}
end { DisplayBT };
begin
    Assign ( Fin , 'inbintree.dat');   Reset ( Fin );
    b := EnterBT;
    WriteLn ( 'Построили бинарное дерево по его КЛП-представлению,',
          'вот КЛП-представление :');
    OutBT ( b );       WriteLn;
    WriteLn ('... а теперь – вид дерева ...');
    DisplayBT ( b );
    WriteLn;
    DestroyBt ( b );
end.
```

Определение структуры данных бинарного дерева через описанный в 3.2 набор базовых операций естественно с математической точки зрения и удобно при функциональном (рекурсивном) стиле программирования (в особенности в языках функционального программирования, например, в Лиспе). Однако при итеративном стиле программирования в случае необходимости модификации дерева возникают некоторые неудобства, связанные с употреблением функции *ConsBT*. Дело в том, что если мы перестраиваем бинарное дерево, работая только через базовые функции, т.е. не используя особенности ссылочного представления и работу с указателями, то нам приходится сначала “разбирать”текущее поддерево с помощью селекторов, а затем “собирать” заново с помощью конструктора *ConsBT*. При этом приходится производить соответствующие операции с динамической памятью для замены старого узла на новый. Это приводит к дополнительным накладным расходам при выполнении программы. Во многих случаях этих накладных расходов можно избежать, вводя более удобный набор конструкторов. Например, полезно выделить как самостоятельные следующие дополнительные функции:

1) функцию *MakeRoot*, порождающую бинарное дерево из одного узла (корня);

2) функцию *SetLeft*, модифицирующую заданное бинарное дерево таким образом, что на месте его пустого левого поддерева появляется заданный лист;

3) функцию *SetRight*, аналогичную *SetLeft*, но для правого поддерева.

Функциональная спецификация этих функций имеет вид:

$$MakeRoot : \alpha \rightarrow NonNullBT;$$

$$SetLeft : \alpha \otimes NonNullBT \rightarrow NonNullBT;$$

$$SetRight: \alpha \otimes NonNullBT \rightarrow NonNullBT$$

с набором аксиом:

– аксиомы для *MakeRoot* (∀ *u*: α):

Root (*MakeRoot* (*u*)) = *u*;
Null (*Left* (*MakeRoot* (*u*))) = *True*;
Null (*Right* (*MakeRoot* (*u*))) = *True*;

– аксиомы для SetLeft (∀*u*: α; ∀ *b*:*NonNullBT*: *Null* (*Left* (*b*))):

Left (*SetLeft* (*u* , *b*)) = *MakeRoot* (*u*);
Root (*SetLeft* (*u* , *b*)) = *Root* (*b*);
Right (*SetLeft* (*u* , *b*)) = *Right* (*b*);

– аксиомы для *SetRight* (∀*u*: α; ∀*b*:*NonNullBT*: *Null* (*Right* (*b*))):

Right (*SetRight* (*u* , *b*)) = *MakeRoot* (*u*);
Root (*SetRight* (*u* , *b*)) = *Root* (*b*);
Left (*SetRight* (*u* , *b*)) = *Left* (*b*).

На уровне функциональной спецификации все эти функции могут быть выражены через функцию *ConsBT*. При этом приводимые далее соотношения имеют место для переменных *u*, *b*1, *b*2 таких, что ∀*u*: α;
∀*b*2:*NonNullBT*: *Null* (*Right* (*b*2)); ∀*b*1: *NonNullBT*: *Null* (*Left* (*b*1)):

MakeRoot(*u*) = *ConsBT*(*u*, Λ, Λ);
SetLeft (*u* , *b*1) = *ConsBT* (*Root* (*b*1), *MakeRoot* (*u*), *Right* (*b*1));
SetRight (*u* , *b*2) = *ConsBT* (*Root* (*b*2), *Left* (*b*2), *MakeRoot* (*u*)).

Абстрактные функции *SetLeft* и *SetRight* удобнее реализовать в виде процедур, например:

```
procedure SetLeft ( a: Elem; b: BinT);
  {Pred: Not Null ( b )  &  Null ( Left ( b ) ) }
begin
 if Null ( b ) then Otkaz ( 5 )
    else
     if Not Null ( Left ( b ) ) then Otkaz ( 6 )
     else b^.LSub := MakeRoot ( a )
end {SetLeft}
```

Рассмотрим ***ссылочную реализации ограниченного бинарного дерева на базе вектора***:

```
type Adr  = 0 .. MaxAdr;          {диапазон "адресов"в векторе      }
     BinT = Adr;                  {представление бинарного дерева   }
     Node = record                { узел :                          }
              Info : Elem;        {     = содержимое                }
              LSub : BinT;        {     = левое  поддерево          }
              RSub : BinT         {     = правое поддерево          }
     end { Node };
     Mem = array [Adr] of Node    { вектор для хранения дерева      }
```

Здесь вектор типа *Mem* представляет собой память для хранения одного бинарного дерева (или нескольких бинарных деревьев, ограниченных в совокупности). Элемент вектора есть запись типа *Node*, содержащая узел и ссылки на поддеревья. На рис.3.11 приведен пример представления бинарного дерева. Дерево представляется переменной типа *BinT*, например b:*BinT*, значение которой *b*=3 есть номер элемента массива, в котором хранится корень дерева. Фактически переменная *b* играет роль ссылки на корень дерева (или адреса корня в векторной памяти). Пустому дереву соответствовало бы значение *b*=0, т.е. в этом представлении константа *NillBT* = 0. Элементы вектора, не занятые под хранение узлов дерева, образуют свободную память, которую удобно организовать в виде линейного циклического списка (для этого используется одно из полей звена *Node* (например, поле *LSub*)). При этом элемент вектора с индексом (адресом) 0 играет роль ссылки на начало списка свободной памяти.

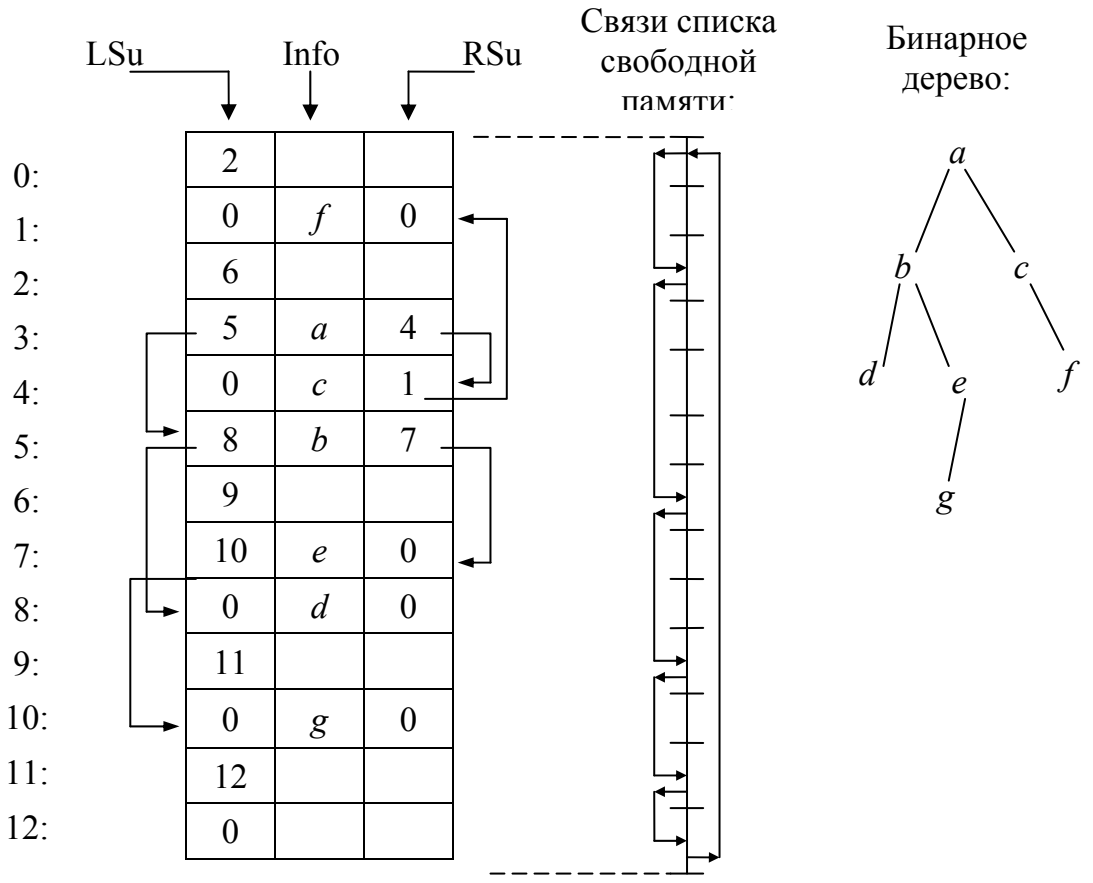


Рис. 3.11. Пример ссылочного представления бинарного дерева на базе вектора. Справа – бинарное дерево; слева – его размещение в массиве, играющем роль динамической памяти.

Мы не будем более подробно рассматривать эту реализацию, поскольку она во многом аналогична ссылочной реализации линейного списка на базе вектора, рассмотренной ранее в 1.4.

3.6. Упражнения

При выполнении упражнений следует использовать варианты реализации бинарного дерева (см.3.5), указанные преподавателем.

Для представления деревьев во входных данных рекомендуется использовать скобочную запись, кроме случаев, специально оговоренных в условии задачи. В выходных данных рекомендуется представлять дерево (лес) в горизонтальном виде (т.е. с поворотом на 90°), а бинарные деревья - в виде уступчатого списка.

В заданиях 1 - 3 предлагается реализовать как рекурсивные, так и не рекурсивные процедуры (функции); в последнем случае следует использовать стек и операции над ним (см. разд.2).

1. Задано бинарное дерево *b* типа *BT* с типом элементов *Elem*. Для введенной пользователем величины *E* (**var** *E*: *Elem*):

а) определить, входит ли элемент *E* в дерево *b*;

б) определить число вхождений элемента *E* в дерево *b*;

в) найти в дереве *b* длину пути (число ветвей) от корня до ближайшего узла с элементом *E* (если *E* не входит в *b*, за ответ принять -1).

2. Для заданного бинарного дерева *b* типа *BT* с произвольным типом элементов:

а) определить максимальную глубину дерева *b*, т.е. число ветвей в самом длинном из путей от корня дерева до листьев;

б) вычислить длину внутреннего пути дерева *b*, т.е. сумму по всем узлам длин путей от корня до узла;

в) напечатать элементы из всех листьев дерева *b*;

г) подсчитать число узлов на заданном уровне *n* дерева *b* (корень считать узлом 1-го уровня);

д) определить, есть ли в дереве *b* хотя бы два одинаковых элемента.

3. Заданы два бинарных дерева *b*1 и *b*2 типа *BT* с произвольным типом элементов. Проверить:

а) подобны ли они (два бинарных дерева ***подобны***, если они оба пусты, либо они оба непусты и их левые поддеревья подобны и правые поддеревья подобны);

б) равны ли они (два бинарных дерева ***равны***, если они подобны и их соответствующие элементы равны);

в) зеркально подобны ли они (два бинарных дерева ***зеркально подобны***, если они оба пусты, либо они оба непусты и для каждого из них левое поддерево одного подобно правому поддереву другого);

г) симметричны ли они (два бинарных дерева ***симметричны***, если они зеркально подобны и их соответствующие элементы равны).

4. Задано бинарное дерево *b* типа *BT* с произвольным типом элементов. Используя очередь и операции над ней (см.п.2), напечатать все элементы де-

рева *b* по уровням: сначала - из корня дерева, затем (слева направо) - из узлов, сыновних по отношению к корню, затем (также слева направо) - из узлов, сыновних по отношению к этим узлам, и т.д.

5. Для заданного леса с произвольным типом элементов:

а) получить каноническое представление леса бинарным деревом;

б) вывести изображение леса и бинарного дерева;

в) перечислить элементы леса в горизонтальном порядке (в ширину).

6. (Обратная задача.) Для заданного бинарного дерева с произвольным типом элементов:

а) получить лес, канонически представленный этим бинарным деревом;

б) вывести изображение бинарного дерева и леса;

в) см.п.5,в.

7. Рассматриваются бинарные деревья с элементами типа *char*.

Заданы перечисления узлов некоторого дерева *b* в порядке КЛП и ЛКП. Требуется:

а) восстановить дерево *b* и вывести его изображение;

б) перечислить узлы дерева *b* в порядке ЛПК.

8. Рассматриваются бинарные деревья с элементами типа *char*. Заданы перечисления узлов некоторого дерева *b* в порядке ЛКП и ЛПК. Требуется:

а) восстановить дерево *b* и вывести его изображение;

б) перечислить узлы дерева *b* в порядке КЛП.

9. Формулу вида

< формула > ::= < терминал > | (< формула > < знак > < формула >)

< знак > ::= + | - | *

< терминал > ::= 0 | 1 | ... | 9 | *a* | *b* | ... | *z*

можно представить в виде бинарного дерева ("***дерева-формулы***") с элементами типа *char* согласно следующим правилам:

- формула из одного терминала представляется деревом из одной вершины с этим терминалом;

- формула вида (f_1 s f_2) представляется деревом, в котором корень – это знак *s*, а левое и правое поддеревья – соответствующие представления формул f_1 и f_2. Например, формула (5 * (*a* + 3)) представляется деревом-формулой, показанной на рис.3.12.

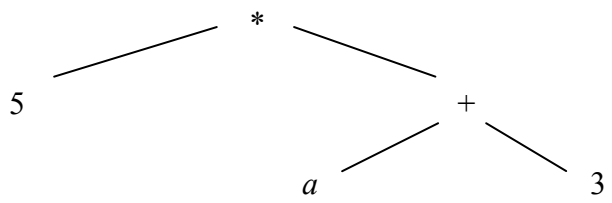


Рис. 3.12. Дерево-формула

Требуется:

а) для заданной формулы *f* построить соответствующее дерево-формулу *t*;

б) для заданного дерева-формулы *t* напечатать соответствующую формулу *f*;

в) с помощью построения дерева-формулы *t* преобразовать заданную формулу *f* из инфиксной формы в префиксную (перечисление узлов *t* в порядке КЛП) или в постфиксную (перечисление в порядке ЛПК);

г) выполнить с формулой *f* преобразования, обратные преобразованиям из п. в);

д) если в дереве-формуле *t* терминалами являются только цифры, то вычислить (как целое число) значение дерева-формулы *t*;

е) упростить дерево-формулу *t*, заменяя в нем все поддеревья, соответствующие формулам (*f* + 0), (0 + *f*), (*f* - 0), (*f* * 1), (1 * *f*), на поддеревья, соответствующие формуле *f*, а поддеревья, соответствующие формулам (*f* * 0) и (0 * *f*), - на узел с 0;

ж) преобразовать дерево-формулу *t*, заменяя в нем все поддеревья, соответствующие формулам (f_1 * (f_2 + f_3)) и ((f_1 + f_2) * f_3), на поддеревья, соответствующие формулам ((f_1 * f_2) + (f_1 * f_3)) и ((f_1 * f_3) + (f_2 * f_3));

з) выполнить в дереве-формуле *t* преобразования, обратные преобразованиям из п. ж);

и) построить дерево-формулу *t*1 - производную дерева-формулы *t* по заданной переменной.

10. ***Деревом поиска***, или таблицей в виде дерева, называется бинарное дерево, в котором слева от любого узла находятся узлы с элементами, меньшими элемента из этого узла, а справа – с большими элементами. Предполагается, что все элементы дерева попарно различны и что их тип *Elem* допускает применение операций сравнения; пример дерева поиска показан на рис.3.13.

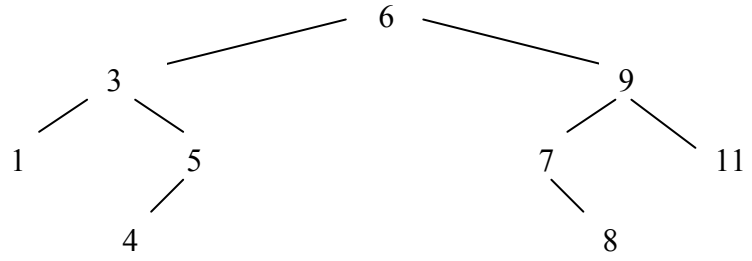


Рис. 3.13. Пример дерева поиска

Требуется:

а) по заданному файлу *F* (типа **file of** *Elem*), все элементы которого различны, построить соответствующее дерево поиска *ts*;

б) для заданного дерева поиска *ts* проверить, входит ли в него элементе типа *Elem*, и если не входит, то добавить элементе в дерево поиска *ts*;

в) записать в файл *F* элементы заданного дерева поиска *ts* в порядке их возрастания.

Примечания и библиографические указания

Материал, изложенный в учебном пособии, на протяжении ряда лет был опробован при выполнении лабораторных и курсовых работ по отдельным разделам учебных дисциплин «Программирование» (линейные списки, иногда – иерархические списки) и «Алгоритмы и структуры данных» (иерархические списки, стеки, очереди, деки, деревья и леса) для специальностей 073000 – «Прикладная математика» , 220400 – «Программное обеспечение вычислительной техники и автоматизированных систем» и направлений 552800 и 644600 – «Информатика и вычислительная техника» и 510200 – «Прикладная математика и информатика». Классические книги [2, 10, 16], в которых рассматриваются основные динамические структуры данных в связи с разработкой эффективных алгоритмов, в большей степени ориентированы на алгоритмический аспект проблемы. Известные книги Н.Вирта [5, 6] ориентированы на *программирование* эффективных алгоритмов, но в меньшей степени затрагивают *технологию* программирования. Используемая в пособии идеология абстрактных типов данных, основанная на выделении этапов спецификации, представления и реализации динамических структур данных, представлена в той или иной форме в [7, 8, 11, 12, 13, 14, 15]. В оригинальном учебном курсе [11] фактически последовательно реализован именно такой подход, включая идею реализации одних структур данных на базе других. Более формальное изложение вопросов, связанных со спецификацией, представлением и реализацией структур данных, можно найти в [8] и в цитированной там литературе.

Материал по рекурсивной обработке списков из 1.6 и 1.7 может рассматриваться как "мостик" в функциональное программирование. На абстрактном уровне эти вопросы хорошо изложены в [18, 19], а основы реализации иерархических списков на Паскале можно найти в [1, 4].

При рассмотрении стека, очереди и дека мы в значительной степени опирались на [3, 11, 13] и на статью К.Хоора в [7].

Дополнительную информацию по программированию деревьев можно найти в [10, 13, 15, 20].

В качестве основных источников заданий для упражнений использовались [10, 13, 17]. Дополнительно можно рекомендовать [9].

Список литературы

1. Алагич С., Арбиб М. Проектирование корректных структурированных программ. М.: Радио и связь, 1984.
2. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. М.: Мир, 1979.
3. Ахо А.В., Хопкрофт Дж., Ульман Дж. Д. Структуры данных и алгоритмы: Уч. пос. – М.: Издательский дом «Вильямс», 2000.
4. Бауэр Ф.Л., Гооз Г. Информатика. Вводный курс: В 2 ч. М.: Мир, 1990.
5. Вирт Н. Алгоритмы + структуры данных = программы. М.: Мир, 1985.
6. Вирт Н. Алгоритмы и структуры данных. – М.: Мир, 1989. (СПб.: Невский Диалект, 2001. (2-е изд., испр.))
7. Дал У., Дейкстра Э., Хоор К. Структурное программирование. М.: Мир, 1975.
8. Калинин А.Г., Мацкевич И.В. Универсальные языки программирования. Семантический подход. М.: Радио и связь, 1991.
9. Касьянов В.Н., Сабельфельд В.К. Сборник заданий по практикуму на ЭВМ: Учеб. пособие для вузов. М.: Наука, 1986.
10. Кнут Д. Искусство программирования для ЭВМ. В 3 т. Т.1. Основные алгоритмы. М.: Мир, 1976. (Учеб. пособие: В 3 т. Т. 1. Основные алгоритмы. 3-е изд.–М.: Издательский дом «Вильямс», 2000.)
11. Кушниренко А.Г., Лебедев Г.В. Программирование для математиков: Учеб. пособие для вузов. М.: Наука, 1988.
12. Лисков Б., Гатэг Дж. Использование абстракций и спецификаций при разработке программ. М.: Мир, 1989.
13. Лэнгсам Й., Огенстайн М., Тененбаум А. Структуры данных для персональных ЭВМ. М.: Мир, 1989.
14. Мейер Б., Бодуэн К. Методы программирования: В 2 т. М.: Мир, 1982.
15. Райли Д. Абстракция и структуры данных: Вводный курс. М.: Мир, 1993.
16. Рейнгольд Э., Нивергельт Ю., Део Н. Комбинаторные алгоритмы. Теория и практика. М.: Мир, 1980.
17. Пильщиков В.Н. Сборник упражнений по языку Паскаль: Учеб. пособие для вузов. М.: Наука, 1989.
18. Хендерсон П. Функциональное программирование. Применение и реализация. М.: Мир, 1983.
19. Берж ??
20. Седжвик Р. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск: Пер. с англ. – К.: Издательство «ДиаСофт», 2001.

О Г Л А В Л Е Н И Е