Московский государственный университет имени
М. В. Ломоносова
Факультет вычислительной математики и кибернетики

А.А. Вылиток, Т.К. Матвеева

ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ. ЗАДАНИЕ ПРАКТИКУМА. ЯЗЫК ПАСКАЛЬ

МАКС ПРЕСС

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В. ЛОМОНОСОВА

Факультет вычислительной математики
и кибернетики

А.А. Вылиток, Т.К. Матвеева

Динамические структуры данных. Задание практикума. Язык Паскаль

Учебно-методическое пособие
для студентов 1 курса

Издание второе, переработанное и дополненное

МАКС ПРЕСС

МОСКВА – 2022

УДК 519.682(075.8)
ББК 32.973-018я73
В92

https://elibrary.ru/ieykvd

Печатается по решению Редакционно-издательского совета факультета вычислительной математики и кибернетики МГУ имени М.В. Ломоносова

Рецензенты:
Е.А. Кузьменкова – доцент кафедры системного программирования ВМК МГУ;
С.Ю. Соловьев – профессор кафедры алгоритмических языков ВМК МГУ

Вылиток А.А., Матвеева Т.К.

В92 **Динамические структуры данных. Задание практикума. Язык Паскаль :** Учебно-методическое пособие для студентов 1-го курса / А.А. Вылиток, Т.К. Матвеева. – 2-е изд., перераб. и доп. – Москва : МАКС Пресс, 2022. – 64 с.
ISBN 978-5-317-06835-6

Учебно-методическое пособие «Динамические структуры данных. Задание практикума. Язык Паскаль» предназначено для студентов 1 курса факультета ВМиК МГУ имени М.В. Ломоносова. В пособии подробно освещается одна из непростых тем для начинающих программистов – работа с динамическими структурами данных.

Рассмотрены линейные списки и двоичные деревья, их реализация на языке Паскаль посредством как статических, так и динамических структур данных. Приведено большое количество примеров с решениями.

В заключение сформулировано задание практикума, предназначенное для студентов факультета ВМиК. Дан образец решения одного из возможных вариантов задания.

Библиогр. 12.

УДК 519.682(075.8)
ББК 32.973-018я73

ISBN 978-5-317-06835-6

Статические и динамические объекты. Указательный тип в языке Паскаль.

Во многих языках программирования объекты программы (переменные, константы, функции и др.) могут быть статическими или динамическими. Здесь мы ограничимся рассмотрением одного вида программных объектов – переменных. Примером статического объекта в языке Паскаль является переменная, описанная в блоке программы или подпрограммы (процедуры, функции). Так, описание

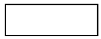
```
var n : integer;
```

порождает статическую переменную целого (или целочисленного) типа. Отметим три важных момента, связанных с описанием переменной:

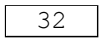
1) Переменная получает имя (в нашем примере это имя `n`), с помощью которого она изображается в программе. Другими словами, данное имя позволяет *обращаться* к переменной в программе[1].
2) Тип переменной задаёт множество допустимых для неё значений. Для переменной `n` это целые числа из диапазона от – `Maxint` до + `Maxint`, где `Maxint` – константа, определяемая реализацией языка Паскаль.
3) Тип переменной задаёт её размер, т.е. объём машинной памяти, необходимый для размещения в этой переменной значений данного типа. Размер значений того или иного типа определяется конкретной реализацией языка. Так, в системе Турбо Паскаль все значения типа `integer` в машинном представлении имеют одинаковый размер – два байта (или 16 двоичных разрядов).

Отвлекаясь от машинного представления, переменную можно считать ячейкой или ящиком, который предназначен для хранения значений определённого типа, причём в каждый момент в ящике хранится не более одного значения.

Для наглядного представления характера значений и действий над переменными договоримся о следующей схеме их графического изображения. Переменные целого типа и других простых типов будем изображать на рисунках в виде прямоугольников. Слева от фигуры, изображающей переменную, указывается её имя. Для целочисленной переменной `n` рисунок будет таким:



В начальный момент выполнения программы значение переменной не определено. Этот факт изображается пустым прямоугольником. Значение, присвоенное переменной в процессе выполнения программы, будем вписывать внутрь соответствующего прямоугольника. Например, после выполнения оператора `n:=32`, рисунок изменится так:

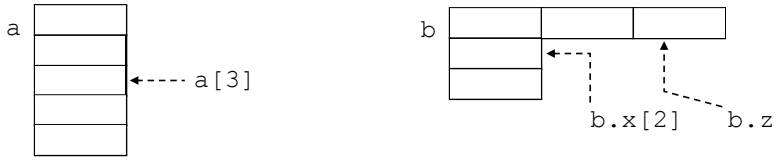


[1] Переменная и ее имя – это две разные сущности. Вместо полной фразы «переменная, имеющая имя n, порождённая данным описанием **var** `n : integer`» будем использовать более короткую – «переменная n» (там, где это не вызовет недоразумений).

Массивы и записи будем изображать фигурами, составленными из фигур, изображающих их компоненты и расположенных соответственно одна под другой по вертикали или одна рядом с другой по горизонтали. Ниже показаны порожденные описанием

```
var a : array[1..5] of integer;
    b : record x: array[1..3] of integer; y,z: real end;
```

массив из пяти чисел и запись с тремя полями. В записи первое поле – массив из трех целых чисел, второе и третье поля – вещественные числа. Имена `a` и `b` обозначают объекты целиком. Для указания компонент сложных объектов используются более сложные обозначения, например `a[3]`, `b.z` или `b.x[2]`.



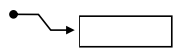
Статические программные объекты порождаются **автоматически** перед выполнением программы или подпрограммы, в которой они **описаны**, и существуют, пока выполнение этой программы или подпрограммы не завершится. Размер статических объектов не изменяется на протяжении всего времени их существования.

К *динамическим* относятся объекты, которые с помощью специальных действий создаются (или изменяют свой размер) уже в **процессе выполнения** программы (подпрограммы).

В языке Паскаль для создания динамических объектов используется стандартная процедура **`new`**, имеющая один параметр. Объект, созданный с помощью `new` в процессе выполнения программы или подпрограммы, существует вплоть до завершения основной программы, или до тех пор, пока он не будет уничтожен явно с помощью другой стандартной процедуры – **`dispose`**.

Для того, чтобы в программе на Паскале обращаться к статическому объекту, мы используем имя, данное ему в разделе описаний. Динамические объекты заранее не описываются – как же к ним обращаться? Для этого вместе с порождаемым объектом создается специальное значение – *указатель* на этот объект[1]. Чтобы хранить указательные значения, используются переменные специального *указательного* типа. При порождении объекта указатель на него помещается в переменную, являющуюся фактическим параметром оператора вызова процедуры `new`. Каждому динамическому объекту соответствует свой указатель, различные объекты имеют различные указатели.

На рисунках будем изображать указатель точкой и связывать его стрелкой с соответствующим объектом:



[1] В предыдущем издании мы использовали вместо указателя термин «ссылка». Однако в стандарте Паскаля [1] используется термин «pointer», который на русский язык лучше перевести как «указатель», в то время как термин «reference» (ссылка) в стандарте используется для обозначения параметра-переменной, описываемого с помощью `var`. Ссылки и указатели как различающиеся понятия есть и в других языках, например в Си++.

В машинных терминах указатель – это адрес объекта в оперативной памяти компьютера. Язык Паскаль не позволяет явно записывать адреса в программе, к тому же на этапе составления программы не известно, по какому адресу будет расположен тот или иной динамический объект. Поскольку указатели не имеют явного обозначения в программе, работа с динамическими объектами происходит посредством статических переменных указательного типа.

Синтаксис задания указательного типа определяется следующим правилом БНФ (описание понятия БНФ см. в [2-4]):

⟨задание указательного типа⟩::= ↑⟨имя типа⟩ ,

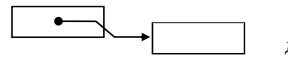
где стрелка означает, что задаётся указательный тип, а ⟨имя типа⟩ – это **имя** любого стандартного или ранее описанного **типа**.

Значениями переменных указательного типа могут быть указатели на динамические объекты, причём только того типа, имя которого указано в задании после стрелки. Переменные указательного типа описываются обычным способом. Например, в силу описаний

```
type pointer_to_integer = ↑integer;
var p : pointer_to_integer;
```

значением переменной p может быть указатель на динамический объект целого типа.[21] В начальный момент выполнения программы переменная p не имеет никакого значения (значение не определено) :

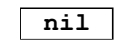
Если далее с помощью оператора new(p) порождается динамический объект, указатель на него автоматически присваивается переменной p. Схематично результат изображается следующим образом:



Можно сказать, что переменная p теперь "указывает" на объект целого типа. Поэтому саму указательную переменную тоже называют указателем. Заметим, что параметр процедуры new однозначно определяет, какого типа объект порождается. В данном случае из описания типа переменной p следует, что порождается объект типа integer. Отметим также, что порождаемые объекты не имеют никакого начального значения.

Среди значений любого указательного типа есть специальное значение, которое называется *пустой указатель*. Оно не связано ни с каким объектом, т.е. ни на что фактически не указывает. В программе такое значение для всех указательных типов явно изображается служебным словом **nil**.

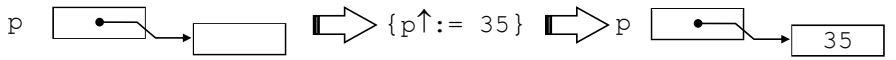
Пусть q – указательная переменная того же типа, что и переменная p. Рисунок



[1] Переменную указательного типа можно также описать непосредственно, используя задание типа в разделе описания переменных, например **var** p :↑integer.

говорит о том, что значением q является пустой указатель, то есть в данный момент q не указывает ни на какой объект (пустой прямоугольник означал бы, что значение q не определено).

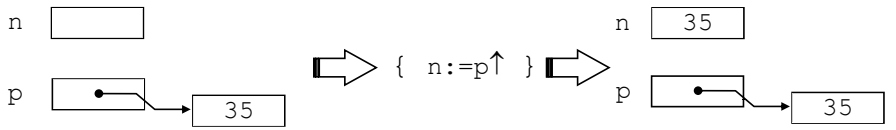
Осталось выяснить, как в программе работать с порождённым динамическим объектом, например, как присвоить ему значение. Если к обозначению указательной переменной приписать справа стрелку ↑, то мы получим обозначение объекта, указатель на который хранится в данной переменной. Выполнение оператора p↑:= 35 приведёт к тому, что целая переменная, на которую указывает p, получит значение 35:



значение p↑ до присваивания не определено

значение p↑ после присваивания равно 35

Пусть n – имя переменной типа integer. Тогда при выполнении оператора n:=p↑ значение динамической переменной будет присвоено статической переменной n:



значение n до присваивания не определено

значение n после присваивания равно 35

Итак, приписывание стрелки справа к указателю – это способ изобразить динамическую переменную в программе. Такое обозначение (со стрелкой справа) называется *переменная с указателем*. С помощью металингвистических формул (БНФ) подытожим все возможные обозначения переменных в языке Паскаль:

⟨переменная⟩::= ⟨полная переменная⟩ | ⟨переменная с индексом⟩ | ⟨компонента записи⟩ | ⟨буферная переменная⟩ | ⟨переменная с указателем⟩

⟨полная переменная⟩ ::= ⟨имя переменной⟩

⟨переменная с индексом⟩ ::= ⟨переменная-массив⟩ **[**⟨индексное выражение⟩ {**,**⟨индексное выражение⟩}**]**

⟨компонента записи⟩ ::= ⟨переменная-запись⟩**.**⟨имя поля⟩| ⟨имя поля⟩

⟨буферная переменная⟩ ::= ⟨файловая переменная⟩↑

⟨переменная с указателем⟩ ::= ⟨указательная переменная⟩↑

⟨переменная-массив⟩ ::= ⟨переменная⟩$_{\text{типа массив}}$

⟨переменная-запись⟩ ::= ⟨переменная⟩$_{\text{типа запись}}$

⟨файловая переменная⟩ ::= ⟨переменная⟩$_{\text{типа файл}}$

⟨указательная переменная⟩ ::= ⟨переменная⟩$_{\text{типа указатель}}$

⟨имя переменной⟩::=⟨идентификатор⟩

Описания понятий «идентификатор», «имя поля» и др. можно найти в [3,4]. Заметим, что возможны случаи, когда синтаксически верное обозначение переменной не имеет смысла. Тогда программа, в которой встретилось такое обозначение, не может быть исполнена (или её исполнение приведёт к ошибке). Например, обозначение `a[i]`ошибочно, если имя `a` в программе не означает массив. Если это имя описано с помощью **`var`** `a` **`:`** **`array`**`[1..5]` **`of`** `integer`, а значение индекса `i` во время исполнения программы равно 6, то обращение к `a[i]` приведёт к ошибке, так как не существует компоненты массива с таким индексом.

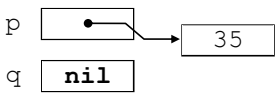
Следует проявлять осторожность и при работе с указателями. Если, например, значение указателя `q` **не определено** или **равно `nil`**, то обращение к `q`↑ приведёт к ошибке.

Теперь рассмотрим операции над значениями указателей одного и того же типа. В Паскале указатели можно присваивать и сравнивать на равенство и неравенство. Присваивание осуществляется, как обычно, с помощью оператора присваивания:

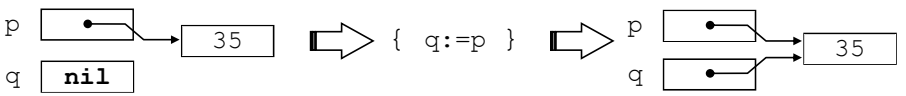
⟨указательная переменная⟩ `:=` ⟨указательное выражение⟩ ,

где ⟨указательная переменная⟩ – обозначение переменной указательного типа, ⟨указательное выражение⟩ – это либо пустой указатель **`nil`**, либо указательная переменная, либо вызов функции указательного типа.

Пусть переменные `p` и `q` имеют значения, изображенные на схеме:



Тогда логические выражения `p=q` и `q<>`**`nil`** дают значение "ложь". После присваивания `q:=p` (результат изображен на рисунке)



значение `q` до присваивания равно **`nil`**

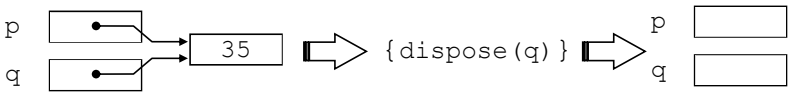
значение `q` после присваивания совпадает со значением `p`, т.е. `q` указывает на тот же объект, что и `p`

логические выражения `p=q` и `q<>`**`nil`** дают значение "истина".

Если объект, созданный с помощью оператора `new`, больше не нужен программе, его целесообразно уничтожить, чтобы он зря не занимал место в оперативной памяти (на его месте могут быть созданы другие объекты).

Уничтожение динамического объекта осуществляется оператором процедуры **dispose(**⟨указательное выражение⟩**)**. Значением выражения, указанного в качестве фактического параметра должен быть указатель на уже существующий динамический объект (иначе ошибка). Этот объект перестает существовать, а указатель на него удаляется из множества значений указательного типа, в результате чего **все** переменные, содержащие указатель на уничтоженный объект, становятся неопределёнными.

Для переменных p и q, изображённых на последней схеме, выполнение оператора dispose(q) приведёт к исчезновению динамического объекта со значением 35, а значения переменных p и q станут неопределёнными.



значения p и q определены и равны между собой

значения p и q (**оба!**) не определены

Линейные списки и их реализация посредством статических и динамических структур данных

Понятие списка хорошо известно из жизненных примеров: список студентов учебной группы, список призёров олимпиады, список (перечень) документов для представления в приёмную комиссию, список почтовой рассылки, список литературы для самостоятельного чтения и т.п.

В математике *список* (или *кортеж*) – это конечная последовательность (допускающая повторения) элементов какого-нибудь множества Ψ. Список обозначается $\langle x_1, x_2, \ldots, x_n \rangle$, где n (n≥0) – количество элементов, или *длина* списка, для i=1,…,n x_i есть *i*-й *элемент* списка ($x_i \in \Psi$). Об элементе x_i говорят также, что он занимает *i-ю позицию* в списке. При n = 0 получается *пустой* список, который не содержит элементов и обозначается ⟨ ⟩.

Элементы множества Ψ могут иметь весьма сложную структуру, отражая реальные объекты и процессы. Вследствие этого возможны списки, в которых некоторые элементы сами являются списками или содержат списки внутри себя. Такие списки называют *иерархическими*. В отличие от них списки, не допускающие таковых внутри себя, называются *линейными*.

Далее в качестве множества Ψ будем рассматривать множество значений некоторого типа языка Паскаль. Этот тип в общем случае будем обозначать именем *elemtype* (от англ. element type – тип элемента).

Важной структурной особенностью списка является то, что его элементы линейно упорядочены в соответствии с их позицией в списке. Для i=1,…,n-1 элемент x_i *предшествует* элементу x_{i+1}; для i=2,…,n элемент x_i *следует за* элементом x_{i-1}.

Можно выделить следующие – наиболее употребительные – операции с линейными списками:

1) Получить значение *i*-го элемента списка или изменить *i*-й элемент;
2) Напечатать элементы списка в порядке их расположения;
3) Найти в списке элемент с заданным значением;
4) Определить длину списка;
5) Вставить новый элемент непосредственно за *i*-м элементом или перед ним, вставить элемент в пустой список;
6) Удалить *i*-й элемент;
7) Соединить два линейных списка в один список;
8) Разбить список на два списка;
9) Создать копию списка;
10) Сделать список пустым.

Возможны и более сложные операции над линейными списками.

Примеры.

1. Вставить в список целых чисел ⟨1,3,4,2,1,5⟩ число 7 после третьего элемента.

Результат: ⟨1,3,4,**7**,2,1,5⟩.

В полученном списке первые три элемента не изменились, четвертую позицию заняло вставляемое число 7, остальные элементы сдвинулись на одну позицию вправо. Длина списка увеличилась на единицу.

2. Вставить строку "фиолетовый" в конец списка строк

⟨"красный", "оранжевый", "жёлтый", "зелёный", "голубой", "синий"⟩.

Результат: ⟨"красный", "оранжевый", "жёлтый", "зелёный", "голубой", "синий", "**фиолетовый**"⟩

3. Удалить из списка символов химических элементов ⟨Ag,Au⟩ первый элемент.

Результат: ⟨Au⟩.

Длина списка уменьшилась на единицу. Если к результату применить ту же операцию – удалить первый элемент – получим пустой список ⟨ ⟩.

4. Вставить в пустой список вещественных чисел ⟨ ⟩ число 3.1415927.

Результат: ⟨3.1415927⟩.

Если в начало полученного списка вставить число 2.7182818, получим список ⟨**2.7182818,**3.1415927⟩.

5. Из списка названий месяцев удалить повторные названия: ⟨December, January, February, March, April, May, December, May, May⟩.

Результат: ⟨December, January, February, March, April, May⟩.

Реализация списков на Паскале

Списки представляют собой удобную структуру данных для решения многих практических задач. Они используются, например, в программах информационного поиска, трансляторах, при моделировании различных процессов. Однако по отношению к языку Паскаль список является абстрактным понятием: в Паскале не предусмотрены ни встроенные операции над списками, ни средства для их описания, подобные тем, что имеются, скажем, для массивов.[1] Поэтому для обработки списков на Паскале потребуется их программная *реализация*. Для этого необходимо:

1) сконструировать средствами Паскаля структуру данных, которая будет представлять в программе список (в этой структуре будут храниться элементы списка);
2) описать в виде процедур и функций требуемые операции над списками.

Мы рассмотрим два наиболее типичных способа реализации списков: посредством массивов и с помощью цепочки звеньев.

Реализация списков посредством массивов

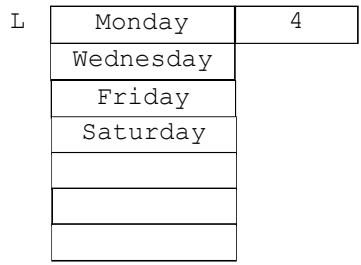
В этой реализации элементы списка располагаются по порядку в компонентах массива, начиная с первой; *i*-й элемент списка хранится в *i*-ой компоненте. Размер массива определяет максимально возможную длину списка. В программе размер массива зададим константой `maxlen`. Для работы со списком требуется ещё знать его текущую длину (или номер компоненты массива, в которой находится текущий последний элемент). Опишем тип `list` (`список`) как запись, имеющую два поля: первое с именем `elems` (от англ. elements – элементы) – собственно массив; второе с именем `last` – позиция последнего элемента списка. Дадим также описание переменной `L` типа `list`:

```
const
  maxlen =    … {подходящее для конкретной задачи целое
                                                  число};
type
  elemtype = … {подходящий для задачи тип элементов};
  list = record
           elems: array [1..maxlen] of elemtype;
           last: integer {позиция последнего элемента}
         end;
var L: list;
```

Выражение `L` (переменная `L`) соответствует понятию «список». Конструкция `L.elems[i]` обозначает в программе *i* -й элемент списка.

[1] В языке Лисп, напротив, список является основной структурой данных с богатым набором операций, а массивы отсутствуют.

Пусть в качестве elemtype задан перечислимый тип: (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday). Реализация списка названий дней недели ⟨ Monday, Wednesday, Friday, Saturday⟩ на схеме будет выглядеть так (считаем, что maxlen = 7) :

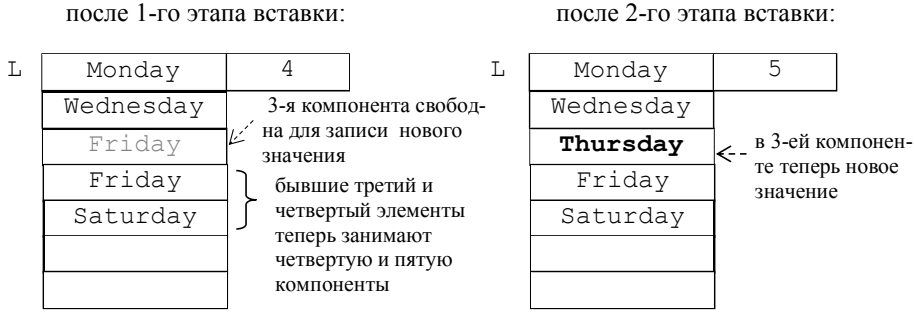


Создать такой список можно следующими действиями: сделать список L пустым, затем поочередно вставлять в него элементы в нужном порядке. Сделать список пустым весьма просто – достаточно обнулить поле last, поскольку список пуст, если его длина равна нулю. Оформим это действие в виде процедуры make_null(L):

```
procedure make_null(var L: list);
 { делает список L пустым }
  begin
    L.last:=0 { длина списка стала равна нулю }
  end; {make_null}
```

Вставка нового элемента x перед *i*-м элементом списка осуществляется в два этапа: во-первых, все значения из компонент массива elems с индексами *i*, *i*+1, …, *last* перемещаются соответственно в компоненты с индексами *i*+1, *i*+2, …, *last*+1; во-вторых, в *i*-ю компоненту помещается элемент x, и значение поля last увеличивается на единицу.

Ниже показано, что происходит при вставке значения Thursday перед третьим элементом списка ⟨ Monday, Wednesday, Friday, Saturday⟩.

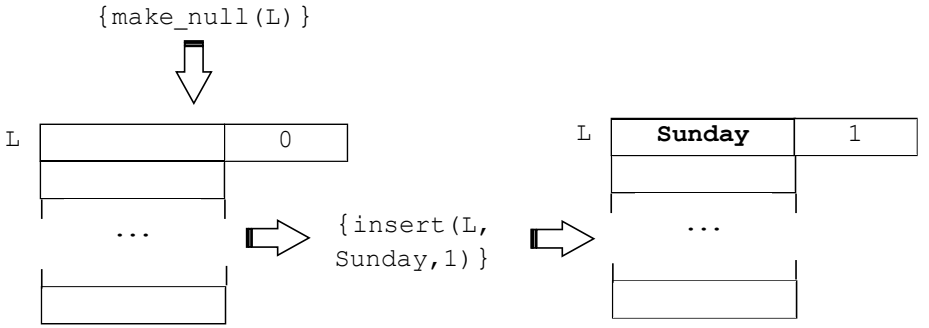


Операцию вставки оформим в виде процедуры insert(L,x,i).

```
procedure insert(var L: list; x: elemtype; i:integer);
 { вставляет в список L элемент x перед  i-м элементом }
 var p: integer;
  begin
    for  p:=L.last  downto  i  do
      {перемещение значений из компонент i,i+1,…,last на
       одну компоненту к концу массива}
           L.elems[p+1]:=L.elems[p];
    L.elems[i]:=x;
    L.last:=L.last+1; {длина списка увеличивается на 1}
  end; {insert}
```

Итак, чтобы вставить значение Thursday перед третьим элементом списка L, в программе следует использовать оператор insert(L, Thursday,3).

Процедуру insert можно использовать и для вставки элемента в пустой список, задав единицу в качестве третьего параметра. Например, make_null(L); insert(L, Sunday,1):



Теперь покажем, как создать наш исходный список ⟨Monday, Wednesday, Friday, Saturday⟩. Такой список получится в результате выполнения следующей последовательности операторов: make_null(L); insert(L, Saturday,1); insert(L, Friday,1); insert(L, Wednesday,1); insert(L, Monday,1).

Для удаления *i*-го элемента опишем процедуру delete(L,i).

```
procedure delete(var L: list; i:integer);
 { удаляет из списка L i-й элемент }
 var p: integer;
  begin
    for  p:=i  to  L.last-1  do
      {перемещение значений из компонент i+1,
       i+2,…,last на одну компоненту к началу
```

```
       массива}
           L.elems[p]:=L.elems[p+1];
    L.last:=L.last-1;{длина списка уменьшается на 1}
  end; {delete}
```

При использовании процедур `insert` и `delete` следует проявлять осторожность: нельзя вставлять элемент, если длина списка максимальна, т.е. равна `maxlen`; нельзя удалять из пустого списка; необходимо правильно задавать номер i элемента в списке: $1 \le i \le$ `last`. Проверку условий для корректной работы со списками реализуем в виде логических функций:

```
function is_full(var L: list):boolean;
 { возвращает  истину, если длина списка максимальна,
   иначе – ложь}
  begin
    is_full:=L.last=maxlen
  end; {is_full}

function is_empty(var L: list):boolean;
 { возвращает  истину, если список пуст, иначе – ложь}
  begin
    is_empty:=L.last=0
  end; {is_empty}

function is_valid(var L: list;i:integer):boolean;
 { возвращает  истину, если в списке есть i-й элемент,
   иначе - ложь }
  begin
    is_valid:=(1<=i) and (i<=L.last)
  end; {is_valid}
```

Опишем также логическую функцию `is_present(L, x)`, проверяющую, есть ли элемент `x` в списке `L`.

```
function is_present(var L: list; x: elemtype):boolean;
 { возвращает  истину, если элемент со значением x
   присутствует в списке, иначе – ложь}
var found:boolean;
       p:integer;
  begin
    found:= false;
    p:=1;
    while not found and (p <= L.last) do
      begin
       found:= x = L.elems[p];
       p:= p+1
      end;
```

```
    is_present:= found
  end; {is_present}
```

Рассмотрим работу этой функции для разных случаев. Если список `L` пуст (`L.last=0`), цикл в теле функции не выполнится ни разу, так как условие `p<=L.last` ложно, и функция возвращает значение «ложь». Если элемент со значением `x` присутствует в списке, то переменная `found` после нескольких повторений тела цикла получит значение «истина», и цикл завершится, так как условие **`not`** `found` станет ложным. Функция возвращает значение «истина». Наконец, если элемент отсутствует в непустом списке, то значение переменной `found` в цикле не изменится. Цикл завершится, поскольку на каждой итерации значение `p` увеличивается, и условие `p<=L.last` станет ложным. Значением функции будет «ложь».

Отметим, что рассмотренная реализация списков посредством массивов позволяет достаточно легко просматривать содержимое списка (есть возможность непосредственного доступа к элементу по его индексу) и вставлять новые элементы списка в его конец. Однако, при вставке элемента в середину или при его удалении требуется перемещение всех последующих элементов. Кроме того, длина списка ограничена константой `maxlen`. При достаточно больших значениях `maxlen` нерационально используется оперативная память, так как размер переменной, представляющей список, пропорционален константе `maxlen`, а не фактической длине списка. Таким образом, пустой список занимает столько же места в оперативной памяти, что и список максимальной длины.

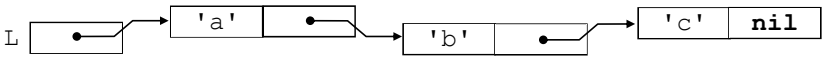
Реализация списков посредством цепочек динамических объектов

Другой подход к реализации списков на языке Паскаль заключается в следующем. Для хранения отдельного элемента списка создается динамический объект – запись из двух полей, называемая *звеном*. В одном из полей (*информационном*) располагается сам элемент, а другое поле содержит указатель на звено, содержащее следующий элемент списка, или пустой указатель, если следующего элемента нет. С помощью указателей звенья как бы сцеплены в *цепочку*. Зная указатель на первое звено можно добраться и до остальных звеньев, то есть указатель на первое звено задаёт весь список. Пустой список представляется пустым указателем. Можно сказать, что звено цепочки является «носителем» элемента списка (хранящегося в информационном поле звена). Приведём соответствующие описания на языке Паскаль.

```
type link=↑node; { указатель на звено }
elemtype = … {подходящий для задачи тип элементов};
node= record            {звено состоит из двух полей:}
      elem: elemtype; {элемент списка}
      next: link      {указатель на следующее звено}
      end;
list=link; {список задаётся указателем на звено }
var L: list;{список}
```

Обратим внимание на то, что в описании типа `link` используется имя `node`, а в описании типа `node` используется имя `link`. По правилам Паскаля в задании указательного типа можно указывать имя, которое ещё не описано.[1] Поэтому тип `link` описывается раньше, чем `node`.

Пример. Список символов ⟨`'a','b','c'`⟩, представленный цепочкой звеньев, изображается так (в переменной `L` – указатель на первое звено):



При этом в программе выражение `L` (переменная `L`) означает указатель на первое звено в цепочке; `L↑` означает само первое звено, `L↑.elem` – первый элемент списка, `L↑.next` – указатель на второе звено. Далее,
`L↑.next↑` – само второе звено,
`L↑.next↑.elem` – второй элемент списка,
`L↑.next↑.next` – указатель на третье звено,
`L↑.next↑.next↑` – само третье звено,
`L↑.next↑.next↑.elem` – третий элемент списка,
`L↑.next↑.next↑.next` – пустой указатель (конец списка).
Выражение `L` имеет и другой смысл. Оно обозначает список в программе, поскольку, зная указатель на первое звено, мы имеем доступ ко всем остальным звеньям, т.е. «знаем» весь список. С этой точки зрения выражение `L↑.next` в нашем примере обозначает список[2] ⟨`'b','c'`⟩, а выражение `L↑.next↑.next↑.next` – пустой список.

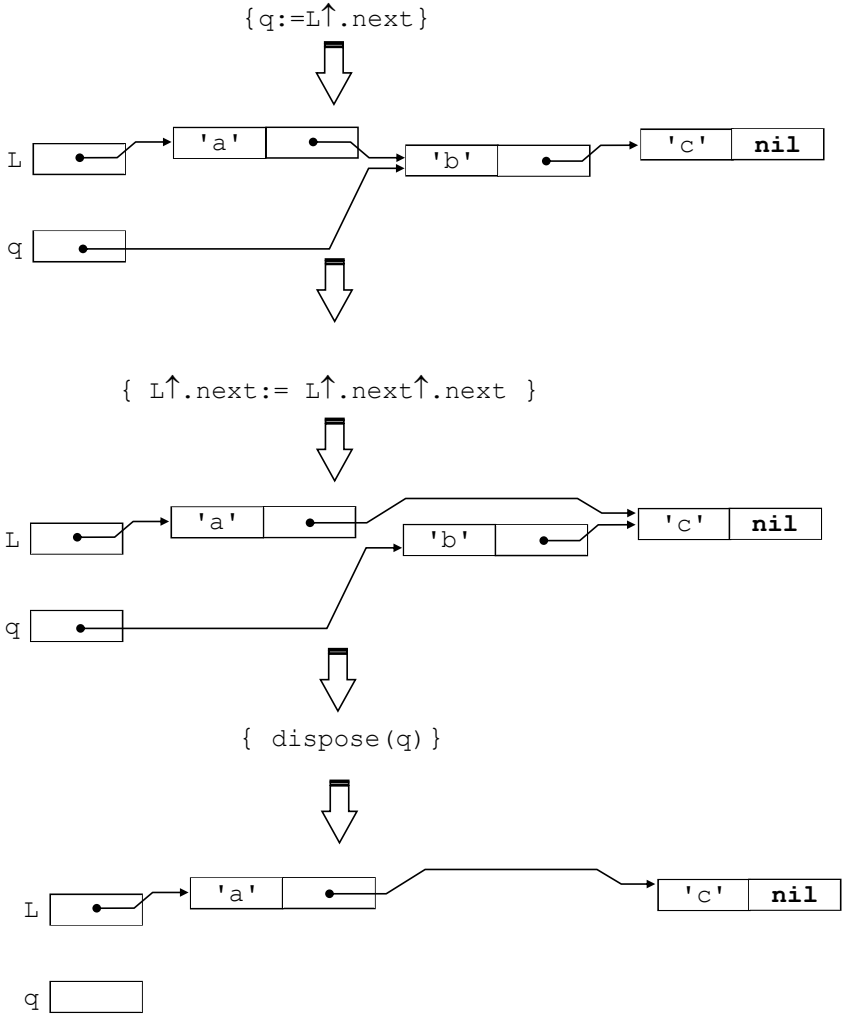
Подчеркнём, что соседние звенья цепочки располагаются в оперативной памяти произвольно относительно друг друга, в отличие от соседних компонент массива, всегда занимающих смежные участки памяти. Такое расположение звеньев облегчает операции вставки и удаления, так как нет необходимости перемещать элементы, как это было в случае реализации списков массивами. Поясним это на примерах.

Пусть из списка ⟨`'a','b','c'`⟩, представленного в программе переменной `L`, требуется **удалить второй элемент**. Для этого достаточно исключить из цепочки второе звено – «носитель» второго элемента. Изменим указатель в поле `next` первого звена: `L↑.next:= L↑.next↑.next.` Теперь после первого звена в цепочке идёт звено, содержащее элемент `'c'`. Получился список ⟨`'a','c'`⟩. Чтобы исключённое звено не занимало места в памяти, его следует уничтожить процедурой `dispose`, предварительно запомнив указатель на него во вспомогательной переменной `q`. Итак, окончательное решение таково:

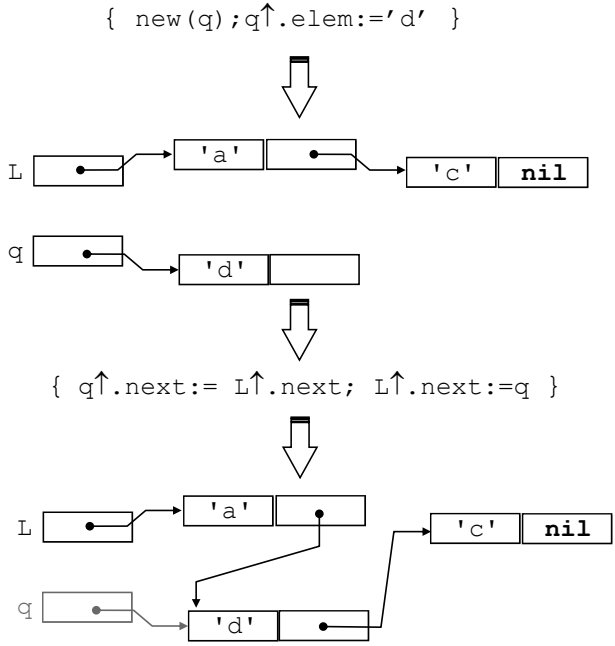
```
q:=L↑.next; L↑.next:= L↑.next↑.next; dispose(q).
```

Покажем на рисунке происходящие после каждого действия изменения.

[1] Это является исключением из общего правила «сначала опиши – потом используй».
[2] По правилам Паскаля (см. [1]) имена `link` и `list` обозначают один и тот же тип, и мы вправе рассматривать `L↑.next` как переменную типа `list`, означающую список.



Пусть теперь требуется **вставить `'d'` после первого элемента** списка ⟨`'a', 'c'`⟩. Решение состоит из двух этапов. Во-первых, необходимо создать «носитель» – звено для хранения вставляемого элемента, и занести этот элемент в поле `elem` «носителя». Во-вторых, путём изменения переменных-указателей включить созданное звено в цепочку после первого звена. Первый этап реализуется фрагментом `new(q);q↑.elem:= 'd'`, где `q` – вспомогательная переменная типа `link`. Фрагмент `q↑.next:= L↑.next; L↑.next:=q` осуществляет второй этап вставки. Следующий рисунок иллюстрирует этапы вставки.



Из примеров видно, что длина цепочки (количество звеньев в ней) может динамически изменяться, т.е. изменяться в процессе выполнения программы. Подобно цепочкам можно сконструировать и более сложные структуры, в которых объекты связаны между собой с помощью указателей.[1]
Такого рода структуры данных называются *динамическими*.

Приведём **реализацию** некоторых операций над списками.

```
function is_empty(L: list):boolean;
 { возвращает  истину, если список пуст, иначе – ложь}
  begin
    is_empty:= L = nil
  end; {is_empty}

function is_valid(L: list; i:integer):boolean;
 { возвращает  истину,если в списке есть i-й элемент,
   иначе - ложь }
 var p: link; k:integer;
  begin
  p:=L; k:=1;
  while (p<> nil) and (k<=i) do
    begin p:=p↑.next; k:=k+1 end;
    is_valid:=(k>i) and (i>0)
  end; {is_valid}
```

Оператор `p:=p↑.next` обеспечивает продвижение указателя `p` на одно звено вперед на каждой итерации цикла.

[1] Некоторые сложные структуры будут рассмотрены далее.

Рассмотрим работу функции is_valid для разных случаев. Если список L пуст (L = **nil**), цикл в теле функции не выполнится ни разу, так как условие p < > **nil** ложно. Значение переменной k равно единице, поэтому выражение (k>i) **and** (i>0) ложно для любого i. Функция возвратит значение «ложь». Если i<1, то цикл также ни разу не выполнится, так как при начальном значении k, равном единице, отношение k<=i ложно. Отношение i>0 также ложно, следовательно, значением выражения (k>i) **and** (i>0) будет «ложь», т.е. функция возвращает значение «ложь». Если i>=1 и в списке есть i-й элемент, то тело цикла выполнится ровно i раз. Переменная k получит значение i+1, выражение (k>i) **and** (i>0) истинно, и функция возвращает значение «истина». Наконец, если i>=1 и в списке нет i-го элемента, то выход из цикла произойдёт из-за ложности выражения p < > **nil** (по достижении конца списка). При этом выражение k<=i останется истинным. Следовательно, k>i ложно и (k>i) **and** (i>0) ложно. Значением функции будет «ложь».

Заметим, что вместо переменной p для продвижения по списку (т.е. по цепочке звеньев) можно было использовать параметр-значение L, являющийся локальной переменной. Изменение значения переменной L внутри функции не «испортит» значение фактического параметра, представляющего список в основной программе.

```
function pointer(L: list; x: elemtype):link;
 { возвращает указатель на звено, содержащее первое
   вхождение элемента x, если x входит в список L,
   иначе – nil }
var p: link; found: boolean;
  begin
    found:=false; p:=L;
    while not found and (p< > nil) do
      if x = p↑.elem then found:=true
                     else p:=p↑.next;
      pointer:=p ;
  end; {pointer}
```

Логическая переменная found используется для завершения цикла, когда звено с искомым элементом x найдено и p указывает на него. Если элемент x отсутствует в списке, то по достижении конца списка переменная p получит значение **nil**, цикл завершится и функция возвратит значение **nil**.

```
procedure insert_after(L: list; x,y: elemtype);
 { вставляет в список L элемент y после первого вхождения
   элемента x, если  x входит в список }
 var p,q: link;
  begin
    p:=pointer(L,x);
    if p < > nil then
      begin q:= p↑.next;
            new(p↑.next);
```

```
            p↑.next↑.elem:= y;
            p↑.next↑.next:= q
      end
  end; {insert_after}
```

Процедура insert_after оставляет пустой список пустым, а в непустом списке указатель на первое звено остается неизменным. Ниже мы приведём процедуру insert_before, вставляющую элемент в список перед первым вхождением заданного элемента. В процедуре insert_before придётся сделать параметр L (означающий список) параметром-переменной, описав его с помощью **var**.[1] Память для параметра L отводиться не будет, имя L внутри процедуры будет обозначать внешнюю переменную, указанную в качестве фактического параметра при обращении к процедуре. В случае вставки элемента в начало списка, внешняя переменная изменит свое значение.

Вообще описывать в процедуре параметр L, обозначающий список, с помощью **var** нужно в следующих случаях: 1) если может измениться указатель на первое звено списка (т.е. на первое звено цепочки, представляющей список); 2) если изначально пустой список может стать непустым; 3) если непустой список может стать пустым.

```
procedure insert_before(var L: list; y,x: elemtype);
 { вставляет в список L элемент y перед первым вхождением
   x, если  x входит в список }
 var p,q: link; found: false;
  begin
    p:=L;
    if p<>nil then
      if p↑.elem = x then {вставка в начало списка}
        begin new(L); L↑.elem:= y; L↑.next:= p
        end;
                    else {вставка не в начало}
        begin
          found:=false;
          while not found and (p↑.next <> nil) do
            if x = p↑.next↑.elem then found:=true
                                 else p:=p↑.next;
          if found then {p указывает на звено,
                предшествующее звену с элементом x}
           begin
            q:= p↑.next;
            new(p↑.next);
            p↑.next↑.elem:= y;
            p↑.next↑.next:= q
      end
  end; {insert_before}
```

[1] Такой параметр называют также передаваемым «по ссылке».

Процедура insert_before рассматривает отдельно случай вставки в начало списка, а для остальных случаев ищется звено, предшествующее «носителю» первого вхождения x, чтобы связать предшествующее звено со вставляемым звеном для y. Можно упростить процедуру insert_before, применив следующий прием: вставить новое звено за «носителем» первого вхождения x, затем скопировать x в новое звено, а в старый «носитель» x записать y. Приведем новую версию этой процедуры.

```
procedure insert_before1(L: list; x,y: elemtype);
 { вставляет в список L элемент y после первого вхождения
   элемента x, если x входит в список }
 var p,q: link;
  begin
    p:=pointer(L,x);
    if p < > nil then
      begin q:= p↑.next;
            new(p↑.next);
            p↑.next↑.next:= q;
            p↑.next↑.elem:= x;
            p↑.elem:=y;
      end
  end; {insert_before1}
```

Следующая процедура delete удаляет из списка L элемент, непосредственно следующий за x (точнее – за его первым вхождением).

```
procedure delete(L: list; x elemtype);
 { удаляет из списка L элемент, следующий за первым
   вхождением элемента x }
 var p,q: link;
  begin
    p:=pointer(L,x);
    if p< >nil then
      begin q:=p↑.next;
            if q<>nil then
               begin p↑.next:= q↑.next; dispose(q)
               end
      end
  end; {delete}
```

Для полного уничтожения списка опишем процедуру destruct.

```
procedure destruct(L: list);
 { разрушает список L,освобождая память, занимаемую
   звеньями }
 var q: link;
  begin
    while L < > nil do
```

```
    begin q:=L;
          L:=L↑.next;
          dispose(q)
    end
end; {destruct}
```

Отметим, что цель `destruct` – полностью освободить память, занимаемую списком, но не сделать список пустым. Формальный параметр процедуры `destruct` описан как параметр-значение (т.е. в теле процедуры мы работаем с локальной переменной `L`). Следовательно, на месте фактического параметра при обращении к этой процедуре может быть любое указательное выражение. Например, вызов функции, которая возвращает указатель на первое звено списка или **`nil`**.

Пусть переменные `L1` и `L2` имеют одинаковые значения – указатель на первое звено некоторого списка. Чтобы сделать список `L1` (и `L2`) пустым, понадобится такой фрагмент: `destruct(L1); L1:=nil; L2:=nil`. После выполнения оператора `destruct(L1)` значения обеих переменных `L1` и `L2` не определены, но после операторов присваивания их значением становится пустой указатель, т.е. `L1` (и `L2`) представляют теперь пустой список.

Приведённые операции со списками подразумевают, что с элементами списка (типа `elemtype`) можно выполнять сравнения и присваивания. Если это не так (например, когда `elemtype` – файловый тип), нужно отдельно реализовать функцию для сравнения и процедуру для присваивания элементов, и использовать их в приведённых функциях работы со списками.

Для списков с элементами-файлами алгоритм вставки, используемый в процедуре `insert_before` предпочтительней, чем `insert_before1`, так как не происходит лишнее копирование элемента, что для файлов весьма существенно.

Как уже отмечалось, операции вставки и удаления при реализации списков цепочками требуют меньше действий, чем при реализации массивами, так как не требуется перемещать элементы, следующие за вставляемым (или удаляемым). Поэтому данная реализация больше подходит для задач, в которых изменения в списках происходят часто. Кроме того, в ней нет ограничения на длину списка (максимальная длина зависит только от ресурсов вычислительной системы); эффективнее используется память (занимаемый объём памяти пропорционален текущей длине списка).

Недостатками данной реализации являются необходимость хранить в звеньях дополнительную информацию – указатели, а так же отсутствие прямого доступа к *i*-му элементу списка. Если нужно присвоить переменной `x` значение *i*-го элемента списка `L`, то в случае реализации массивами это делается одним оператором `x:=L.elems[i]`, а при реализации цепочками понадобится фрагмент, содержащий цикл: `p:=L; for k:=1 to i-1 do p:=p↑.next; x:= p↑.elem`. По этой причине в задачах, где списки меняются сравнительно редко, а просмотр элементов осуществляется часто, выгоднее использовать реализацию с помощью массивов.

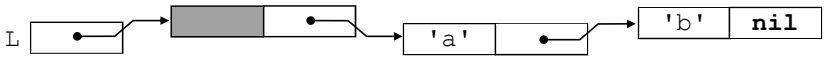
Цепочку динамических объектов, реализующую список, называют *однонаправленным списком*. В литературе по программированию под термином

«список» часто подразумевается именно однонаправленный список. Данное значение термина обычно имеется в виду и тогда, когда употребляют выражение «звено списка». В некоторых задачах важно уметь продвигаться по списку не только от начала к концу, но и в обратном направлении. Для этого используются *двунаправленные списки*. Звено двунаправленного списка содержит два указателя – на следующее и на предыдущее звенья. Эта особенность должна учитываться при реализации операций (например, вставки, удаления) для двунаправленных списков. Для ряда приложений полезны *кольцевые списки*, в которых указатель `next` последнего звена указывает на первое звено списка. Возможны двунаправленные кольцевые списки.

Ещё один вариант реализации списков в виде динамических цепочек – так называемые списки с заглавным звеном – рассматривается в следующем подразделе.

Списки с заглавным звеном

Если в начало обычного однонаправленного списка добавить дополнительное звено, не содержащее элемента (т.е. поле `elem` в этом звене не определено), то получим *список с заглавным звеном*. Пример:



Для таких списков некоторые алгоритмы обработки записываются проще, так как в списке всегда есть хотя бы одно звено (даже если он пуст). Пусть надо реализовать операцию вставки в конец списка `L` элемента `x`. Сравним решения для списка без заглавного звена и для списка с заглавным звеном:

```
procedure insert1(var L: list; x: elemtype);
 { вставляет в конец списка L  без заглавного звена
   элемент x}
 var p,q: link;
  begin
    new(q); q↑.elem:= x; q↑.next:= nil;
    {q указывает на созданное для вставляемого элемента
      звено }
    if L = nil then L:= q  {случай пустого списка
                           обрабатывается отдельно}
              else
      begin p:= L;
        while p↑.next< > nil  do p:= p↑.next;
         {p указывает на последнее звено в списке}
        p↑.next:= q
         {присоединили в конец списка новое звено}
      end
  end; {insert1}
```

```
procedure insert2(L: list; x: elemtype);
 { вставляет в конец списка L  с заглавным звеном
   элемент x}
 var p,q : link;
  begin
    new(q); q↑.elem := x; q↑.next:= nil; p:= L;
    while p↑.next< > nil do p:= p↑.next;
     {p указывает на последнее звено в списке}
    p↑.next := q {присоединили в конец списка новое звено}
  end; {insert2}
```

Как видим, для списка с заглавным звеном нет нужды рассматривать случай L = **nil** отдельно. Кроме того, параметр, означающий список, передается по значению, так как он никогда не изменяется процедурой insert2.

Рекурсивная обработка списков

Вспомним математическое определение списка: это конечная последовательность элементов (возможно, пустая). В непустой последовательности можно выделить первый элемент – «голову» последовательности – и оставшуюся часть – «хвост». Заметим, что «хвост» в свою очередь также является последовательностью. Таким образом, получаем рекурсивное определение последовательности:

⟨последовательность⟩::= ⟨голова⟩ ⟨хвост⟩ | ⟨пусто⟩

⟨голова⟩::=⟨элемент⟩

⟨хвост⟩::=⟨последовательность⟩

Опираясь на это определение, можно задать *рекурсивную процедуру (функцию)*[1] поэлементной обработки последовательности. Пусть, например, требуется напечатать список символов L (тип элементов – char). Процедуру печати можно сформулировать так: если список пуст – ничего не делать (так как нечего печатать), иначе напечатать «голову», а за ней напечатать «хвост». Для печати «хвоста» используем эту же процедуру (рекурсивное применение). Ниже приводятся рекурсивные процедуры print1 и print2 для печати списков соответственно без заглавного звена и с заглавным звеном.

```
procedure print1 (L: list);
{ печать списка без заглавного звена }
  begin if L <> nil then
    begin  write(L↑.elem,'␣'); {печать «головы»}
           print1(L↑.next)     {печать «хвоста»}
    end;
  end; {print1}
```

[1] Процедуры (функции), которые во время исполнения могут прямо или косвенно (через другие процедуры или функции) обратиться сами к себе, называются рекурсивными.

```
procedure print2 (L: list);
{ печать списка с заглавным звеном }
  begin if L↑.next <> nil then
    begin  write(L↑.next↑.elem,'␣');
           print2(L↑.next)
    end;
  end; {print2}
```

Рассмотрим ещё одну задачу: «*Не используя операторов цикла и перехода, описать функцию* `sum(L)`*, вычисляющую сумму элементов списка целых чисел* `L`*. Считать, что сумма пустого списка равна нулю, и что список представлен цепочкой звеньев без заглавного звена*».

Поскольку циклы запрещены условием задачи, для прохода по списку воспользуемся рекурсией (как в реализации процедуры `print1`). Дадим рекурсивное определение суммы списка. Сумма пустого списка равна нулю, сумма непустого списка есть результат сложения «головы» с суммой «хвоста»:

$$\texttt{sum(L)} = \begin{cases} \texttt{0, если L пуст,} \\ \texttt{L↑.elem+sum(L↑.next),иначе.} \end{cases}$$

Теперь запишем определение функции на Паскале.

```
function sum(L:list):integer;
begin
  if L = nil then sum:= 0
             else sum:= L↑.elem +
                        sum(L↑.next)
end;{sum}
```

Параметр `L` передаётся по значению. Поэтому при каждом вызове этой функции создаётся локальная (по отношению к данному вызову) переменная `L`, и ей присваивается значение фактического параметра. При выходе из функции локальная переменная исчезает, и значение функции возвращается в точку обращения к ней. При рекурсивном обращении могут одновременно существовать несколько (**различных !**) локальных переменных с именем `L` (по одной для каждого вызова). Рисунок 1 иллюстрирует выполнение оператора `write(sum(Z))` основной программы, где `Z` представляет список ⟨`5,-3`⟩.

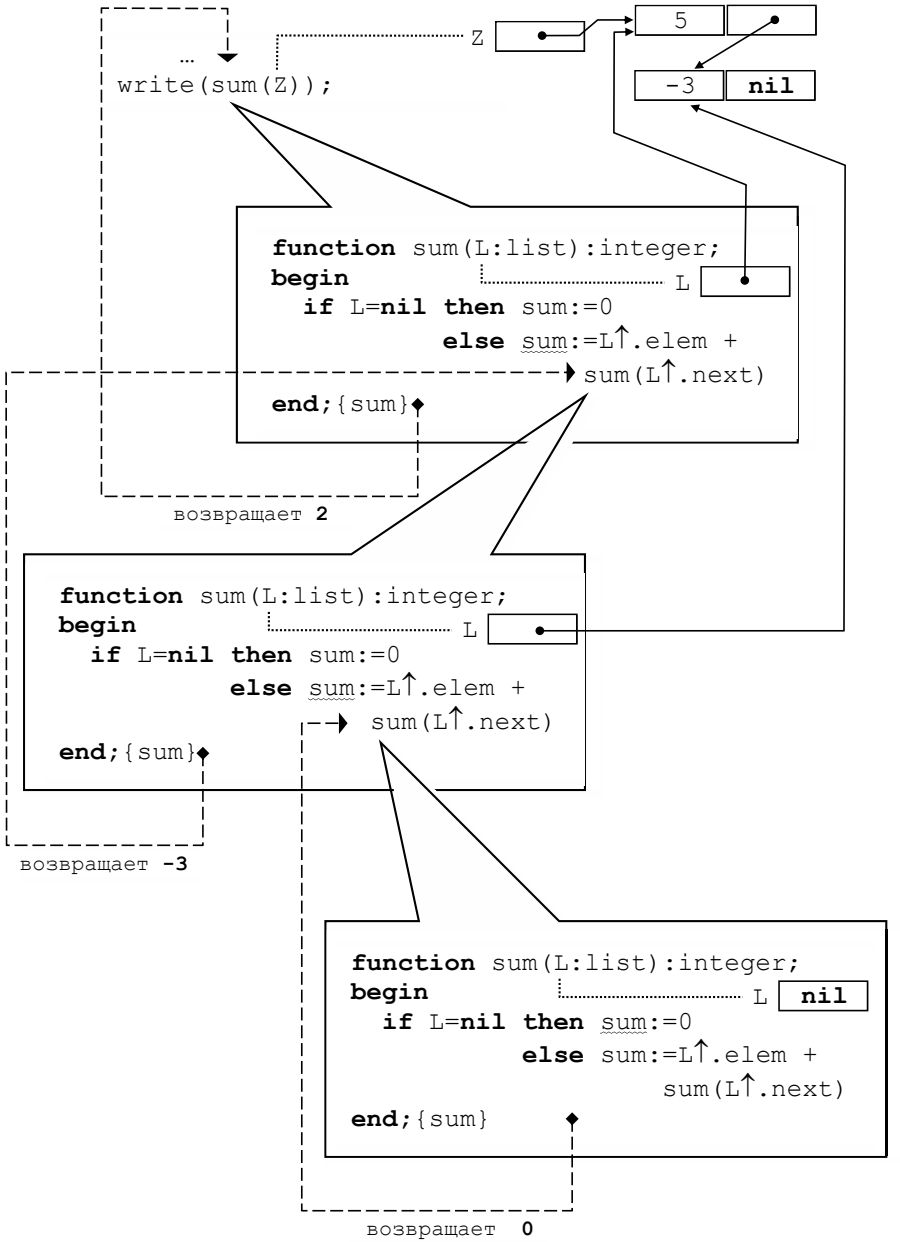


Рис. 1

Очереди и стеки

В повседневной жизни часто приходится иметь дело с хранилищами различных объектов. В библиотеке хранятся книги; в автоматическом оружии есть магазин, в котором хранятся патроны; в холле парикмахерской «хранятся» клиенты, ожидающие своей очереди на обслуживание. Хранилище может быть пустым, если в данный момент никаких объектов в нём нет. Независимо от того,

как устроено хранилище, и какие объекты в нём хранятся, можно выделить два основных действия для работы с ним: 1) добавить объект (положить на хранение); 2) взять объект (изъять из хранилища). Если потребовать, чтобы операции добавления и изъятия объектов удовлетворяли некоторым условиям, можно выделить два особых типа хранилищ: *очередь* и *стек*.

В хранилищах типа «очередь» (или FIFO[1]) для каждого добавляемого объекта справедливо следующее: он не может быть изъят раньше, чем любой другой объект, уже находящийся в данный момент на хранении. Пример – транспортёрная лента для погрузки багажа. Чемодан, раньше других попавший на ленту, раньше других сойдёт с неё.

В хранилищах типа «стек» (или LIFO[2]) каждый добавляемый объект не может быть изъят позже, чем любой другой объект, уже находящийся в данный момент на хранении. Примеры: стопка тарелок на полке буфета; магазин в автоматическом оружии. В этих примерах соблюдается дисциплина LIFO, так как взять можно только верхний предмет (тарелку или патрон), а добавить новый объект можно, только положив его на верхний.

Потребность в использовании стека или очереди возникает при программировании многих реальных процессов и ситуаций. Рассмотрим один из способов организации очередей и стеков в программе. В качестве структуры данных для хранения объектов (элементов) будем использовать список. В случае очереди элементы добавляются в конец списка, а удаляются из его начала. В случае стека элементы добавляются и удаляются с одного конца списка. Такой способ представления стеков и очередей очень распространён, так что на его основе можно дать **более узкие определения данных понятий:**

Очередь – это список, в который элементы всегда добавляются в конец, а удаляются из начала.

Стек – это список, в котором все вставки и удаления выполняются на одном конце, называемом *вершиной* стека.

Каждая из возможных реализаций списка на Паскале может быть использована для представления очередей и стеков. Для стеков рассмотрим вариант представления с помощью массивов, а для очередей – с помощью динамических цепочек.

Представление стеков с помощью массивов

Опишем тип данных `stack` и две процедуры: `push(S,x)` – положить в стек `S` элемент `x` и `pop(S,x)` – взять с вершины стека `S` элемент, присвоив его параметру `x`.

```
const
  stacksize = … {максимальное число элементов, которые
                 могут одновременно находиться в стеке};
type
```

[1]FIFO означает “First In – First Out”, т.е. «первым вошёл – первым вышел».
[2]LIFO означает “Last In – First Out”, т.е. «последним вошёл – первым вышел».

```
  elemtype = … {подходящий для задачи тип элементов
                стека};
  stack = record
           elems: array [1..stacksize] of elemtype;
           top: integer {индекс вершины стека}
          end;

var S: stack; {стек}

procedure push (var S: stack;x:elemtype);
{ положить x в стек S}
  begin S.top:=S.top+1;
        S.elems[S.top]:=x
  end; {push}

procedure pop (var S: stack; var x:elemtype);
{ взять из стека S верхний элемент и присвоить его x}
  begin x:=S.elems[S.top];
        S.top:=S.top-1;
  end; {pop}
```

Перед началом работы со стеком нужно сделать его пустым. Для этого опишем процедуру `init_stack(S)`:

```
procedure init_stack (var S: stack);
{ начальная установка стека }
  begin  S.top:=0;
  end; {init_stack}
```

К пустому стеку нельзя применять операцию `pop`, а к полному стеку нельзя применять операцию `push`. Опишем функции `full_stack(S)` и `empty_stack(S)`, проверяющие стек на переполнение и пустоту.

```
function full_stack(var S: stack):boolean;
 { возвращает  истину, если стек полон, иначе – ложь}
  begin
    full_stack:=L.top=stacksize
  end; {full_stack}

function empty_stack (var S: stack):boolean;
 { возвращает  истину, если стек пуст, иначе – ложь}
  begin
    empty_stack:=L.top=0
  end; {empty_stack}
```

В некоторых алгоритмах, использующих стек, проверка на пустоту или переполнение перед обращением к стеку не требуется, поскольку для них

доказано, что при корректных входных данных подобных ситуаций не возникает.[1] Для отладки программ, построенных по таким алгоритмам, было бы удобно, чтобы процедуры pop и push сами проверяли корректность операции со стеком и в случае ошибки сообщали о ней. Пусть процедура error с одним параметром – строкой – печатает эту строку и приводит к завершению программы. Дадим соответствующие описания push и pop .

```
procedure push (var S: stack;x:elemtype);
{ положить x в стек S с контролем переполнения}
  begin
    if S.top=stacksize then error('стек полон')
      else begin
             S.top:=S.top+1;
             S.elems[S.top]:=x
           end
  end; {push}

procedure pop (var S: stack; var x:elemtype);
{ если стек непуст, взять из стека S верхний элемент и
  присвоить его x, иначе сообщить об ошибке}
  begin
    if S.top=0 then error('стек пуст ')
      else begin
             x:=S.elems[S.top];
             S.top:=S.top-1;
           end
  end; {pop}
```

После того, как программа отлажена, проверки становятся ненужными и их следует исключить в целях повышения быстродействия программы.

Заметим, что в случаях, когда максимальный размер стека не может быть заранее определён, лучше воспользоваться реализацией на основе динамических цепочек. Тогда размер ограничивается только ресурсами вычислительной среды.

Представление очереди с помощью списка с заглавным звеном

Поскольку операции вставки и удаления осуществляются на разных концах очереди, для работы с ней удобно иметь два указателя – на заглавное и на последнее звенья. Если очередь пуста, оба указателя указывают на заглавное звено. Приведём необходимые описания.

```
type link =↑node; {указатель на звено}
elemtype = … {подходящий тип элементов очереди};
```

[1] В задаче 9 раздела «Примеры задач с решениями» описан алгоритм, использующий стек, в котором при корректных входных данных попытка взять элемент из пустого стека исключена. См. также замечание в решении задачи 10 в том же разделе.

```
node= record
      elem: elemtype; {элемент }
      next: link      {указатель на следующее звено}
      end;
queue = record
          front: link {указатель на заглавное звено};
           rear: link {указатель на последнее звено}
        end;

var Q: queue;{очередь}
```

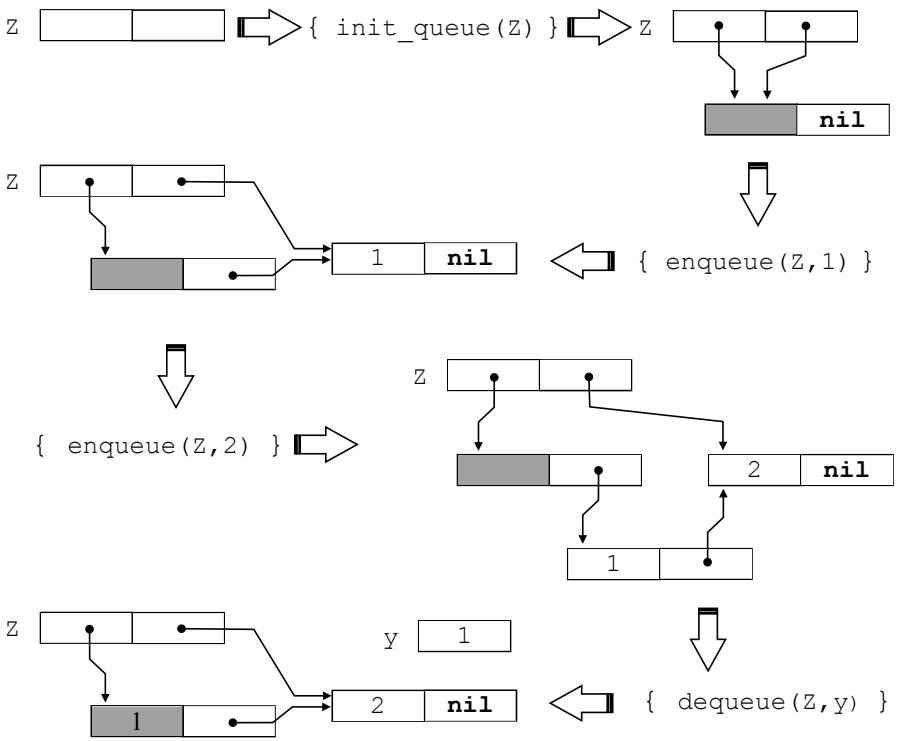


Рис. 2а

```
procedure init_queue(var Q: queue);
{ начальная установка очереди }
  begin  new(Q.front);{создание заглавного звена}
         Q.front↑.next:=nil;
         Q.rear := Q.front;
  end; {init_queue}
```

```
function empty_queue(Q: queue):boolean;
 { возвращает  истину, если очередь пуста, иначе – ложь}
  begin
    empty_queue:= Q.rear = Q.front
  end; {empty_queue}

procedure enqueue(var Q: queue; x:elemtype);
{поставить в очередь Q элемент x}
 begin
  new(Q.rear↑.next);
  Q.rear:= Q.rear↑.next;
  Q.rear↑.elem:=x;
  Q.rear↑.next:= nil
 end; {enqueue}

procedure dequeue(var Q: queue; var x:elemtype);
{взять из очереди Q элемент и присвоить x}
var p:link;
 begin
  x:= Q.front↑.next↑.elem;
  p:= Q.front;
  Q.front:= Q.front↑.next;
  dispose(p)
 end; {dequeue}
```

Пусть в основной программе описаны переменные: `Z` – очередь целых чисел, `y` – целое. На рисунке 2а показан результат последовательного применения команд `init_queue(Z)`, `enqueue(Z,1)`, `enqueue(Z,2)`, `dequeue(Z,y)`. На рисунке 2б подробно изображён процесс выполнения последней команды. Как видим, после исключения из очереди единица оказалась в поле `elem` заглавного звена, перестав быть частью очереди.

При необходимости контролировать корректность использования очереди во время отладки программы, в процедуру `dequeue` можно добавить обращение к процедуре `error` при попытке взять элемент из пустой очереди.

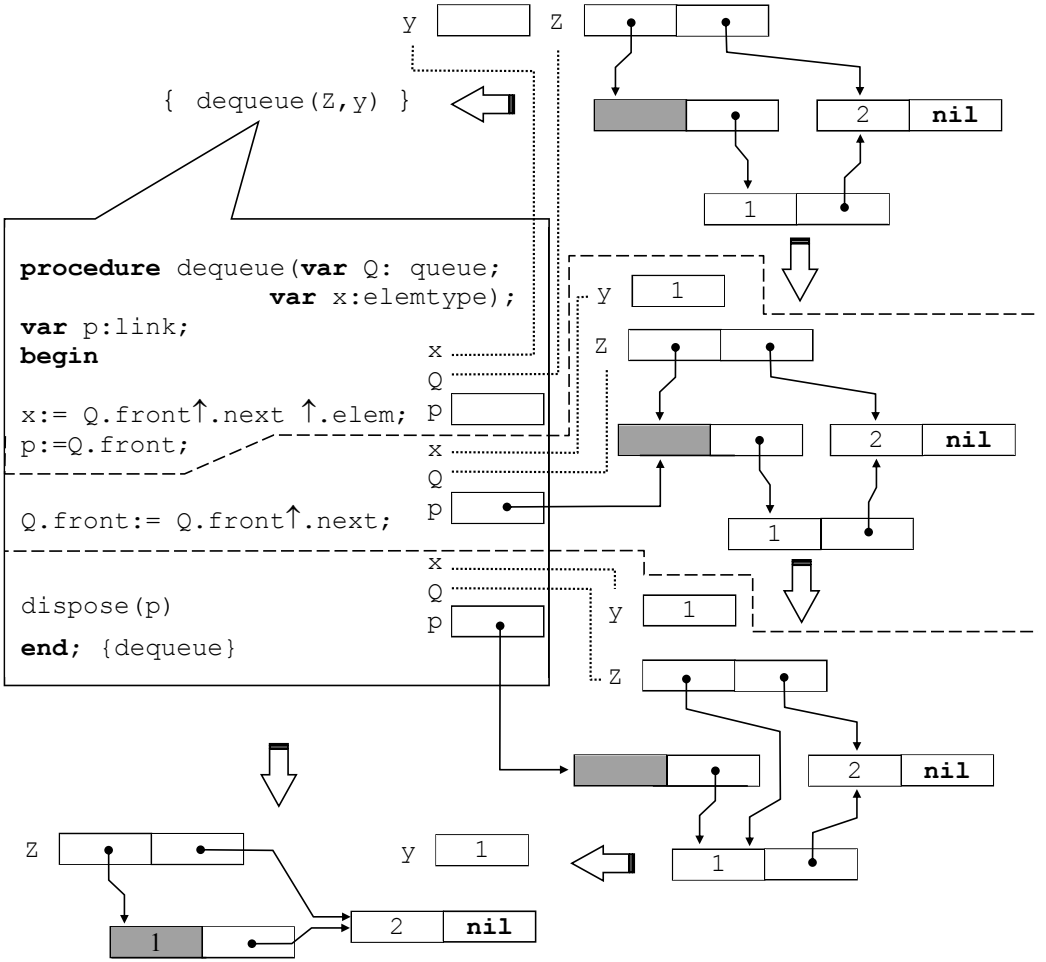


Рис. 2б

Двоичные деревья

В предыдущих разделах мы рассматривали линейные списки, элементы которых являлись значениями одного и того же типа. Теперь рассмотрим особый вид иерархических списков, называемых двоичными деревьями.

Двоичным деревом является пустой список или список, состоящий из трёх элементов: первый элемент является информационным (т.е. содержит некоторые данные) и называется *корнем* дерева; второй и третий элементы в свою очередь являются двоичными деревьями и называются соответственно *левым* и *правым* поддеревом. *Вершиной* двоичного дерева называется его корень, а также все вершины его левого и правого поддеревьев. Вершины двоичных деревьев

являются значениями одного и того же типа. Далее под термином «дерево» будем понимать двоичное дерево.

Примеры.

⟨ ⟩ – пустое дерево, в нем нет вершин.

⟨5,⟨⟩,⟨⟩⟩ – дерево, состоящее из одной вершины 5.

⟨1,⟨2,⟨⟩,⟨⟩⟩,⟨3,⟨⟩,⟨⟩⟩⟩ – дерево с вершинами 1,2,3.

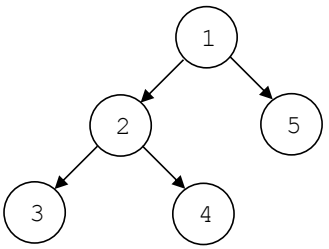
С помощью БНФ понятие двоичного дерева определяется следующим образом.

⟨двоичное дерево⟩::= **⟨ ⟩** | **⟨** ⟨корень⟩, ⟨двоичное дерево⟩,
⟨двоичное дерево⟩ **⟩**

Пусть некоторое дерево T содержит в себе дерево вида ⟨A,⟨B,…,…⟩, ⟨C,…,…⟩⟩. Тогда вершину B будем называть *левым сыном* вершины A в дереве T, а вершину C будем называть *правым сыном*. Вершину A будем называть *родителем* для вершин B и C. *Потомками* некоторой вершины являются все вершины левого и правого поддеревьев дерева, корнем которого является данная вершина. *Предками* некоторой вершины являются все корни деревьев, содержащих данную вершину. Вершина, не имеющая потомков, называется *листом*.

Для деревьев существует наглядное графическое изображение: вершины дерева изображаются в виде кружков и связываются дугами. Дуга от родителя к левому сыну направляется влево вниз, к правому сыну – вправо вниз. Дерево, не содержащее вершин (пустое дерево) можно изобразить как ∅.

Дерево ⟨1,⟨2,⟨3,⟨⟩,⟨⟩⟩,⟨4,⟨⟩,⟨⟩⟩ ⟩,⟨5,⟨⟩,⟨⟩⟩⟩ имеет следующее изображение:



Предками вершины 3 являются вершины 1 и 2. Потомками вершины 1 являются вершины 2, 3, 4, 5. Листьями в данном дереве являются вершины 3, 4, 5.

Реализация деревьев на Паскале

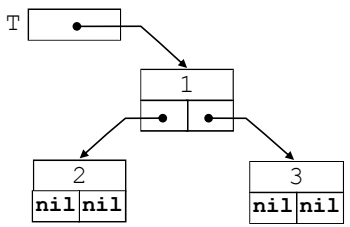
Так же как линейные списки деревья можно представлять в языке Паскаль массивами или связными структурами динамических объектов. Более удобным для обработки деревьев является представление с помощью динамических структур.

Для хранения отдельной вершины дерева создаётся динамический объект – запись из трех полей, называемая *узлом*. Первое поле (информационное) содержит элемент, соответствующий вершине дерева (значение некоторого типа). Второе поле содержит указатель на узел, соответствующий левому сыну вершины, третье поле – на узел, соответствующий правому сыну. Если у вершины нет левого (или правого) сына, то второе (или третье) поле содержит пустой указатель.

Связность узлов обеспечивается указателями. Зная указатель на корень дерева, можно добраться до любого узла. Таким образом, указатель на корень дерева задаёт всё дерево. Пустое дерево представляется пустым указателем. Приведём соответствующие описания на языке Паскаль.

```
type link=↑node; { указатель на узел }
elemtype = … {подходящий для задачи тип элементов};
node= record            {узел состоит из трёх полей:}
      elem:  elemtype; {элемент дерева (вершина)}
      left:  link;      {указатель на левый узел}
      right: link       {указатель на правый узел}
      end;
tree=link; {дерево задаётся указателем на узел}
var T: tree;{дерево}
```

Пример. Представлению дерева ⟨1,⟨2,⟨⟩,⟨⟩⟩,⟨3,⟨⟩,⟨⟩⟩⟩ с помощью связанных узлов соответствует следующее изображение (в переменной T – указатель на корень дерева):



Для данного примера выражение T означает указатель на корневой узел; T↑ означает сам корневой узел, T↑.elem – корневой элемент дерева, T↑.left – указатель на узел для левого сына корневой вершины, T↑.right – указатель на узел для правого сына. Выражение T↑.right↑.right означает пустой указатель. Заметим, что перечисленные выражения имеют и другой смысл. Так, T означает дерево с вершинами 1, 2, 3 и корнем 1. T↑.left означает дерево с одной вершиной 2, T↑.right↑.right означает пустое дерево. Построить дерево T можно с помощью следующего фрагмента программы:

```
new(T); T↑.elem:=1; new(T↑.left);new(T↑.right);
T↑.left↑.elem:=2; T↑.right↑.elem:=3;
T↑.left↑.left:=nil; T↑.left↑.right:=nil;
T↑.right↑.left:=nil; T↑.right↑.right:=nil
```

Способы обхода вершин дерева

При обработке деревьев часто встречается следующая задача: нужно обойти все вершины дерева, выполнив над каждой вершиной некоторую операцию (например, напечатать эту вершину), причем в каждой вершине нужно побывать ровно один раз. Для дерева из *n* вершин существует *n*! способов обхода. Однако наиболее часто используются следующие три дисциплины обхода. Они формулируются рекурсивно в соответствии с рекурсивным определением дерева.

Обход дерева слева направо (*inorder*)

Если дерево пусто, то ничего не делать.

Если дерево не пусто, то 1) обойти слева направо левое поддерево;
2) посетить корень;
3) обойти слева направо правое поддерево.

Обход дерева сверху вниз (*preorder*)

Если дерево пусто, то ничего не делать.

Если дерево не пусто, то 1) обойти сверху вниз левое поддерево;
2) посетить корень;
3) обойти сверху вниз правое поддерево.

Обход дерева снизу вверх (*postorder*)

Если дерево пусто, то ничего не делать.

Если дерево не пусто, то 1) обойти снизу вверх левое поддерево;
2) посетить корень;
3) обойти снизу вверх правое поддерево.

Существуют и другие названия для приведённых способов обхода. Обход сверху вниз называют КЛП – сокращение от последовательности: корень, левое поддерево, правое поддерево. Обход снизу вверх называют ЛПК, а обход слева направо – ЛКП или симметричным обходом.

Пусть процедура `print(e)` с параметром типа элемента дерева (`elemtype`) умеет печатать переданный ей параметр `e`. Для простых типов, таких как `char` или `integer`, процедура `print` будет содержать один оператор `write(e)`. Приведём процедуры, печатающие все элементы дерева в соответствии с тремя определёнными выше дисциплинами обхода.

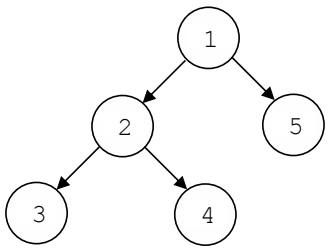
```
procedure inorder(T:tree);
begin
  if T <> nil then
    begin
      inorder(T↑.left);
```

```
      print(T↑.elem); write('␣') ;
      inorder(T↑.right);
    end
end;{inorder}

procedure preorder(T:tree);
begin
  if T <> nil then
    begin
      print(T↑.elem); write('␣');
      preorder(T↑.left);
      preorder(T↑.right);
    end
end;{preorder}

procedure postorder(T:tree);
begin
  if T <> nil then
    begin
      postorder(T↑.left);
      postorder(T↑.right);
      print(T↑.elem); write('␣');
    end
end;{postorder}
```

Рассмотрим следующее дерево:



Для данного дерева процедура `inorder` напечатает последовательность `3 2 4 1 5`. Процедура `preorder` напечатает `1 2 3 4 5`, а `postorder` напечатает `3 4 2 5 1`.

Теперь приведём нерекурсивные версии процедур обхода дерева. Проще всего реализовать обход сверху вниз или КЛП-обход (non-recursive `preoder`). Нам понадобится стек, в котором будут храниться поддеревья, точнее – указатели на корневые узлы поддеревьев. Пусть `stack` – это уже описанный тип данных (реализованный с помощью списка, например) с типом элемента[1] `tree` (или

[1] Тип элемента стека назовем `stelemtype`, так как имя `elemtype` занято под тип элемента дерева.

link, поскольку это идентичные типы) и операторами push и pop. Вначале положим в стек непустое дерево, которое нужно обойти. Затем, пока стек не пуст, будем выполнять следующие действия: взять из стека дерево, напечатать его корневую вершину, положить в стек правое поддерево (если оно не пусто), положить в стек левое поддерево (если оно не пусто), – такая последовательность данных в стеке обеспечивает на следующих итерациях обход левого поддерева, а затем – правого.

```
procedure npreorder(T:tree);
{нерекурсивная версия обхода preorder}
var S:stack;
    R:tree;
begin
  if T<> nil then begin
  init_stack(S);
  push(S,T);
  while not empty_stack(S) do
    begin
      pop(S,R);
      print(R↑.elem); write('␣');
      if R↑.right <> nil then push(S,R↑.right);
      if R↑.left <> nil then push(S,R↑.left)
    end
  end {if}
end;{npreorder}
```

Для организации нерекурсивного обхода слева направо (ЛКП-обхода) подойдёт следующий способ. Вначале положим в стек непустое дерево, затем, пока стек не пуст, будем выполнять действия: взять из стека верхний элемент, положить в стек его правое поддерево (если оно непустое), положить в стек корневую вершину (представленную соответствующим узлом дерева), положить в стек левое поддерево (при условии, что оно непустое); если же взятый из стека элемент оказался отдельной вершиной, то нужно просто напечатать ее.

Есть только одна техническая сложность – как отличить поддерево в стеке (типа tree) от отдельной вершины, представленной в этом же стеке узлом дерева (типа link), – ведь типы tree и link идентичны (эквивалентны), поскольку в стеке можно хранить только однотипные элементы. Для различения будем класть в стек над узлом, означающим отдельную вершину, специальный маркер – пустой указатель **nil**. Пустые деревья в стек не попадают, поэтому наличие маркера **nil** на вершине стека означает, что под ним находится вершина, которую нужно взять из стека и напечатать.

```
procedure ninorder(T:tree);
{нерекурсивная версия обхода inorder}
var S:stack;
    R:tree;
```

```
begin
  if T<> nil then begin
  init_stack(S);
  push(S,T);
  while not empty_stack(S) do
    begin
      pop(S,R);
      if R = nil then begin
        pop(S,R); {берём из стека вершину для печати}
        print(R↑.elem); write('␣')
      end
      else begin
        if R↑.right <> nil then push(S,R↑.right);
          {положили в стек правое поддерево}
        push(S,R); {положили в стек сам корень}
        push(S,nil); {положили в стек маркер корня}
        if R↑.left <> nil then push(S,R↑.left)
          {положили в стек левое поддерево}
      end
    end
  end {if}
end;{ninorder}
```

Аналогично реализуется и обход снизу вверх (ЛПК-обход) с учетом, что вершина дерева при таком обходе должна печататься после левого и правого поддеревьев, и поэтому помещать ее в стек (вместе с маркером **nil**) нужно раньше, чем правое и левое поддеревья.

```
procedure npostorder(T:tree);
{нерекурсивная версия обхода postorder}
var S:stack;
    R:tree;
begin
  if T<> nil then begin
  init_stack(S);
  push(S,T);
  while not empty_stack(S) do
    begin
      pop(S,R);
      if R = nil then begin
        pop(S,R); {берём из стека вершину для печати}
        print(R↑.elem); write('␣')
      end
      else begin
        push(S,R);   {сам корень}
        push(S,nil); {маркер корня}
        if R↑.right <> nil then push(S,R↑.right);{правое}
```

```
        if R↑.left <> nil then push(S,R↑.left) {левое}
      end
    end
  end {if}
end;{npostorder}
```

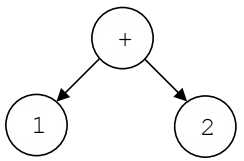
Существуют и другие реализации нерекурсивного обхода деревьев, например, без помещения левого поддерева и маркера **nil** в стек – читателю предлагается построить эти решения самостоятельно.

Представление арифметического выражения в виде дерева

Будем рассматривать арифметические выражения с операндами-цифрами, операциями +, * и круглыми скобками. Синтаксис таких выражений задается с помощью следующей БНФ.

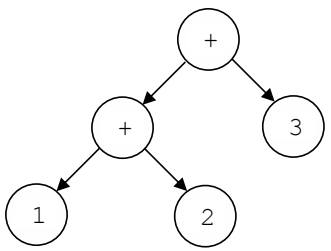
⟨выражение⟩::= ⟨слагаемое⟩ {+ ⟨слагаемое⟩ }
⟨слагаемое⟩::= ⟨множитель⟩ {* ⟨множитель⟩ }
⟨множитель⟩::= (⟨выражение⟩)|0|1|2|3|4|5|6|7|8|9

Простейшее выражение состоит из одной цифры. Такое выражение можно представить деревом, состоящим из одной вершины. Выражение 1+2 можно представить деревом, состоящим из трех вершин: в корне дерева находится операция +, левый сын корня – 1, правый – 2.

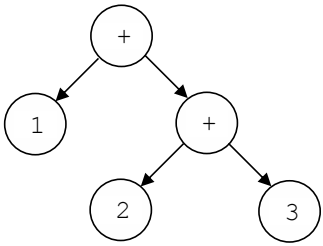


В общем случае, если есть выражение вида αΔβ, где Δ – знак операции, α и β – выражения-операнды, то нужно построить новую вершину с операцией Δ, левым поддеревом которой будет дерево выражения α, а правым поддеревом будет дерево выражения β.

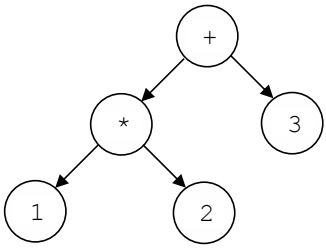
Выражение 1+2+3 в силу левой ассоциативности операции + (см. [2] стр. 15) трактуется как (1+2)+3 и поэтому дерево для него будет таким:



Выражение 1+(2+3) будет иметь следующее представление деревом:



Выражение `1*2+3` с учетом приоритета операций представляется так:



По дереву, соответствующему выражению, легко вычислить значение. Значением дерева из одной вершины является число, представленное в этой вершине. Если в корне дерева операция, нужно вычислить значение левого поддерева, значение правого поддерева и выполнить операцию с полученными операндами.

Опишем функцию, которая вычисляет значение дерева-выражения.

```
function eval(T:tree):integer;
begin
  if T↑.left=nil {and T↑.left=nil} then
    eval:= ord(T↑.elem)-ord('0')
    else
       case T↑.elem of
        '+': eval:= eval(T↑.left)+eval(T↑.right);
        '*': eval:= eval(T↑.left)*eval(T↑.right)
       end
end;{eval}
```

Если к дереву выражения применить процедуру `preoder` для обхода КЛП, получим *префиксную* запись выражения, в которой сначала идёт знак операции, а затем операнды, тоже в префиксной записи. Префиксной записью простого операнда (переменной или константы) является сам этот операнд. Префиксной записью выражения в скобках является префиксная запись этого же выражения без скобок.

Если к дереву выражения применить процедуру `postoder` для обхода ЛПК, получим *постфиксную* запись выражения, в котором сначала идут

операнды в постфиксной записи, а затем знак операции. Постфиксной записью простого операнда (переменной или константы) является сам этот операнд. Постфиксной записью выражения в скобках является постфиксная запись этого же выражения без скобок.

Например, для выражения в *инфиксной* (т.е. обычной математической) записи `(1+2)*3` префиксной записью будет `* + 1 2 3`, а постфиксная запись выглядит так: `1 2 + 3 *`.

Заметим, что префиксная и постфиксная записи вообще не содержат скобок, то есть это *бесскобочные* записи выражения. Также отметим, что операнды в префиксной и в постфиксной записи идут в том же порядке относительно друг друга, что и в инфиксной записи, только знаки операций меняют свои места.

Если представить, что дерево-выражение сделано из шаровых шарниров, то, надавив на дерево справа, оно сложится в линеечку и получится префиксная запись выражения, а если надавить слева, получится постфиксная запись.

Бесскобочные записи выражений имеют свои преимущества. Например, по префиксной записи выражения легко вычислить его значение, используя рекурсию, а по постфиксной – используя стек для промежуточных результатов.

Опишем функцию, которая вычисляет значение выражения (описанного с помощью вышеприведённой БНФ) в префиксной записи. Функция[1] считывает выражение посимвольно из строки входного файла `input`.

```
function evalprefix:integer;
var c: char; {текущий символ}
begin
  read(c);
  if ('0'<=c) and (c <= '9') then
    evalprefix:= ord(c)- ord('0')
  else
    case c of
      '+': evalprefix:= evalprefix + evalprefix;
      '*': evalprefix:= evalprefix * evalprefix
    end
end;{evalprefix}
```

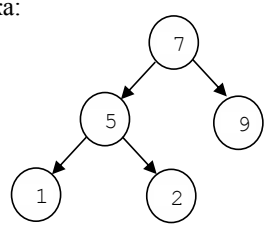
Теперь опишем функцию, вычисляющую значение выражения в постфиксной записи при помощи стека. Функция считывает выражение посимвольно из строки входного файла `input`, пока не встретит конец строки. Если считан операнд (цифра от '0' до '9'), то в целочисленный стек кладётся значение операнда – соответствующее цифре число от 0 до 9. Если считана операция, то со стека берутся два верхних элемента, являющиеся правым и левым операндами операции соответственно, выполнятся операция с этими операндами, и её результат кладётся в стек. При достижении конца строки в стеке останется единственный элемент, являющийся результатом выражения. Его нужно достать из стека и вернуть в качестве значения функции.

[1] Функция не имеет параметров и вызывается рекурсивно. В стандартном Паскале при таком вызове не нужно указывать скобки, но в языке Free Pascal их следует указать: `evalprefix()`.

```
function evalpostfix:integer;
var c: char; {текущий символ}
    op1,op2: integer; {операнды операции}
    S: stack {целочисленный стек, тип элемента - integer}
begin
  init_stack(S);
  read(c);
  while not eoln do
    begin
      if ('0'<=c) and (c <= '9') then
        push(S, ord(c)- ord('0')){операнд кладём в стек}
      else
        case c of  {выполняем операцию, результат в стек}
          '+': begin pop(S,op2); pop(S,op1);
               push(S,op1+op2)end;
          '*': begin pop(S,op2); pop(S,op1);
               push(S,op1*op2)end;
        end;
      read(c)
    end;
  pop(S,op1);{достаём из стека результат}
  evalpostfix:= op1
end; {evalpostfix}
```

В заключение раздела о двоичных деревьях отметим, что двоичные деревья могут использоваться для организации информационных таблиц с эффективными операциями поиска, вставки и удаления [3, 7, 12]. Такие таблицы представляются в форме двоичных деревьев, вершинами которых являются записи с хранимой информацией; каждая запись содержит уникальный ключ – специальное значение, по которому эту запись можно найти. Ключи могут быть целочисленными или любого другого типа с операциями сравнения на больше, меньше, равно. Если абстрагироваться от хранимой информации, ассоциированной с ключами, можно считать, что каждая вершина дерева является ключом.

Деревом поиска назовём двоичное дерево, в котором для каждой вершины её ключ больше всех ключей в левом поддереве и меньше всех ключей в правом поддереве. Пример дерева поиска:



Операции поиска, вставки и удаления в дерево поиска описаны в [3]. В [7,12] показаны способы балансировки деревьев для эффективного поиска.

Примеры задач с решениями

В задачах 1-8 используются списки без заглавного звена при следующем их описании:

```
type link=↑node; {указатель на звено}

elemtype = … {подходящий для задачи тип элементов};
node= record            {звено состоит из двух полей:}
      elem: elemtype; {элемент списка}
      next: link      {указатель на следующее звено}
      end;
list=link; {список задаётся указателем на звено}
```

Задача 1. Описать процедуру `create(L)`, которая создаёт список `L` из строки текстового файла `input`.

Решение

```
procedure create(var L:list);
{создает список из символов  строки текстового файла
                                input; elemtype=char}
var p: link;
 begin
 write('=>'); {выводим приглашение для ввода строки}
 if eoln then {пустая строка - список пуст}
              L:=nil
   else
     begin {создаем первое звено}
      new(p); read(p↑.elem);
      L:=p;
      while  not eoln do
        begin {добавляем в список следующее звено}
         new(p↑.next);  read(p↑.next↑.elem );
         p:=p↑.next
         {p указывает на добавленное звено }
        end;
      { последнее звено должно иметь указатель nil }
      p↑.next:=nil
     end ;
     readln {пропускаем маркер конца строки}
 end;
```

Задача 2. Описать процедуру `print(L)`, которая печатает все элементы списка `L`. Тип элементов – `char`.

Решение

```
procedure print(L:list);
{печатает в файл output строку из элементов списка L}
begin
 while L<>nil do
  begin write(L↑.elem);
        L:=L↑.next
  end;
 writeln
end;
```

Рекурсивное решение было приведено в разделе «Рекурсивная обработка списков».

Задача 3. Описать процедуру `exchange(L)`, которая меняет первый и последний элементы **непустого** списка `L`.

Решение

```
procedure exchange(L:list);
{ меняет местами элементы первого и последнего звеньев
  непустого списка L}
var e: elemtype;
    p: link;
begin
  p:=L; {находим последнее звено}
  while p↑.next<>nil do p:=p↑.next;
  {теперь p указывет на последнее звено;
   L указывает на первое}
  {меняем местами первый и последний элементы}
    e:=L↑.elem; L↑.elem:=p↑.elem; p↑.elem:=e
end;
```

Задача 4. Описать функцию `equal(L1,L2)`, которая проверяет на равенство списки `L1` и `L2`.

Решение

Если хотя бы один из списков пуст, то в качестве результата функции можно взять значение выражения `L1 = L2`. Действительно, если оба списка пусты (т.е. `L1 = nil` и `L2 = nil`), то они равны и выражение `L1 = L2` даёт значение «истина»; если один пуст, другой не пуст (т.е. списки не равны), то значением выражения будет «ложь». В случае, когда оба списка не пусты, нужно организовать цикл, в котором два указателя `L1` и `L2` будут синхронно «пробегать» по звеньям обоих списков, начиная с первых звеньев. Выход из цикла происходит, если достигнуто последнее звено хотя бы в одном из списков (т.е. `L1↑.next = nil` или `L2↑.next = nil`) или найдены звенья с различающимися элементами (`L1↑.elem < > L2↑.elem`). Если выход произошёл из-за неравенства элементов, то результат функции – «ложь». Если

же элементы равны, то результат зависит от того, указывают ли оба указателя L1 и L2 на последние звенья. Это можно проверить с помощью выражения L1↑.next = L2↑.next, которое принимает значение «истина» если и только если L1↑.next = **nil** и L2↑.next = **nil.**

```
function equal(L1,L2:list): boolean;
{возвращает истину, если списки L1 и L2 равны, иначе –
ложь}
begin
  if (L1 = nil) or (L2 = nil)
    then equal:= L1=L2 {истина, если и только если оба
                        списка пусты}
    else {оба списка непусты; L1 и L2 синхронно пробегают
          звенья списков}
      begin
       while (L1↑.next<>nil) and (L2↑.next<>nil)
                          and (L1↑.elem=L2↑.elem)
       {пока не достигли последнего звена в каком-либо
        из списков  и текущие элементы списков равны,
        переходим в каждом из списков к следующему звену}
          do begin L1:=L1↑.next; L2:=L2↑.next end;
       equal:=(L1↑.next=L2↑.next) and (L1↑.elem=L2↑.elem)
       {истина, если и только если L1 и L2 указывают на
        последние звенья и элементы в этих звеньях равны}
      end
end;
```

Задача 5. Описать процедуру reverse(L), которая переворачивает список L, то есть первый элемент списка становится последним, второй – предпоследним и т. д., бывший последним становится первым.

Решение

Для решения данной задачи достаточно изменить указатели так, чтобы звенья в цепочке расположились в обратном порядке. Для этого в поле next первого звена следует записать **nil**, тем самым сделав его последним; в поле next второго звена записать указатель на бывшее первое звено и далее аналогично поступать с остальными звеньями, пока не дойдём до последнего. Чтобы бывшее последним звено стало первым, надо присвоить указатель на него переменной L, представляющей список.

```
procedure reverse(var L:list);
{ переворачивает список L, т.е. изменяет указатели в этом
  списке так, чтобы его элементы оказались расположенными
  в обратном порядке }
var p,q:link;
begin
  if L<>nil {если L=nil, результатом будет пустой список}
```

```
    then
     begin p:=L; q:=L↑.next;
       L↑.next:=nil; {первое звено становится последним}
       while q<>nil {пока есть следующее звено} do
         begin L:=q; {перешли к новому звену; L – указы-
               вает на текущее звено, p – на предыдущее }
               q:=q↑.next; {запомнили указатель на хвост
                            списка }
               L↑.next:=p; {соединили текущее звено с
                            предыдущим}
               p:=L {запомнили указатель
                     на текущее звено}
         end
     end
end;
```

Задача 6. Описать процедуру `in_order(L,x)`, которая в список `L` упорядоченных по неубыванию целых чисел вставляет элемент `x`, сохраняя упорядоченность.

Решение

Будем использовать два указателя `p` и `q`. Указатель `p` будет указывать на первое звено с элементом, большим `x`, если такое существует, иначе – **`nil`**. Указатель `q` будет указывать на предыдущее звено (если оно существует).

```
procedure in_order(var L:list; x:elemtype);
{вставляет x в список L упорядоченных по неубыванию целых
 чисел, сохраняя упорядоченность(elemtype=integer)}
var p,q,t:link;
    greater: boolean;
begin
  p:=L; q:=nil; greater:=false;
 {поиск подходящего места для вставки}
  while (p<>nil) and not greater do
    if  p↑.elem>x then greater:=true;
                  else begin q:=p; p:=p↑.next end;
 {вставка перед  p}
  new(t); t↑.elem:=x; t↑.next:= p;
  if q=nil then {вставка в начало}
                L:=t;
           else  q↑.next:= t
 end;
```

Задача 7. Описать рекурсивную процедуру `count(L,e)`, которая подсчитывает число вхождений элемента `e` в список `L`.

Решение

Если список пуст, то число вхождений элемента e в список L равно нулю. Для непустого списка нужно подсчитать, сколько вхождений имеется в «хвосте» списка, и прибавить единицу или ноль в зависимости от того, совпадает ли «голова» с элементом e.

```
function count(L:list;e:elemtype):integer;
{подсчитывает число вхождений элемента e в список L}
begin
  if L = nil then count:=0
     else count:=count(L↑.next,e)+ord(L↑.elem=e)
end;
```

Задача 8. Описать рекурсивную функцию copy(L), которая строит копию списка L, возвращая указатель на первое звено построенного списка или **nil**, если L пуст.

Решение

```
function copy(L:list):list;
{возвращает указатель на копию списка L}
var head:link;
begin
if L = nil then copy:= nil
         else
           begin  {создаем  копию «головы» списка}
               new(head);
               head↑.elem:= L↑.elem;
               {к созданной «голове» присоединяем
                копию «хвоста»}
               head↑.next:= copy(L↑.next)
           end
end;
```

Задача 9. Используя стек (тип данных и операции над стеком считать уже описанными), описать процедуру pairs(f) для решения следующей задачи. В текстовом файле f находится последовательность символов, сбалансированная по круглым скобкам. Требуется для каждой пары соответствующих открывающей и закрывающей скобок вывести на экран номера их позиций в последовательности, упорядочив пары номеров в порядке возрастания номеров позиций закрывающих скобок. Например, для текста A+(45-F(X)*(B-C)) надо вывести: 8 10; 12 16; 3 17.

Решение

Опишем в процедуре переменную sym для хранения очередного считанного из файла f символа; переменную pos для хранения номера позиции

текущего символа sym, и переменную S, представляющую стек, в который будут помещаться номера позиций открывающих скобок.

Алгоритм обработки последовательности таков. Сначала счётчик позиций pos устанавливаем в ноль, создаём пустой стек (с помощью init_stack(S)) и открываем файл f для чтения. Пока не конец файла, повторяем следующие действия:

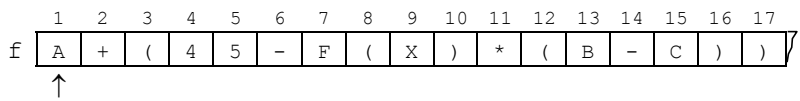
1) считываем в sym очередной символ и увеличиваем pos на единицу;
2) если текущий символ – открывающая скобка, то номер его позиции помещаем в стек (с помощью push(S, pos));
3) если текущий символ – закрывающая скобка, то это значит, что в переменной pos – номер позиции этой скобки, а на вершине стека – номер позиции соответствующей открывающей скобки; поэтому берём элемент из стека, помещая его во вспомогательную переменную i (с помощью pop(S,i)), и печатаем пару (i, pos).

В нашем алгоритме нет проверки стека на переполнение. В сбалансированном по скобкам тексте количество открывающих скобок не может превосходить половину длины текста. Поэтому размер стека должен быть не меньше, чем половина длины обрабатываемого текста, или входной текст должен иметь длину не более чем в два раза превосходящую размер стека.[1]

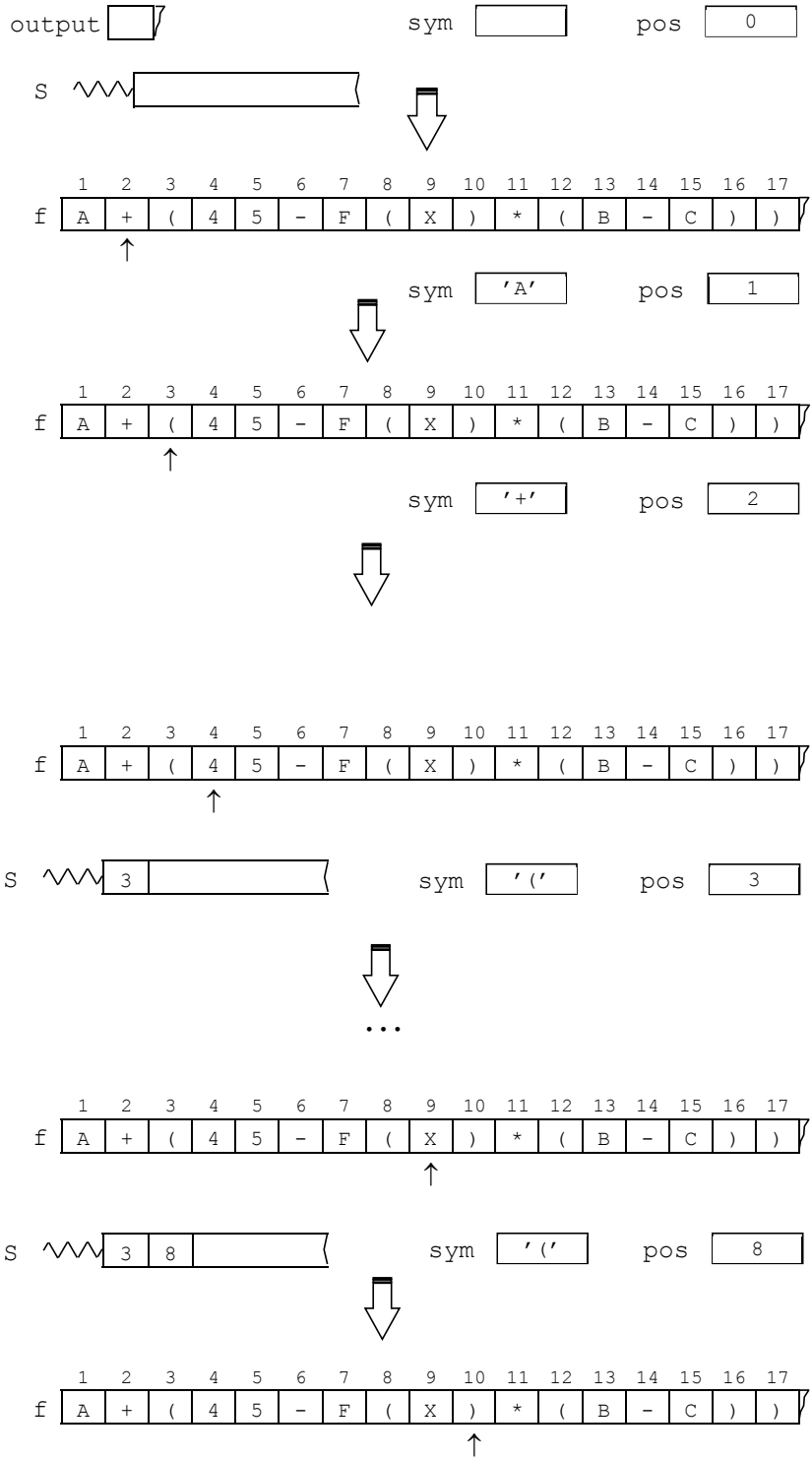
В алгоритме также нет проверки стека на пустоту перед тем, как взять элемент из стека. Эта проверка не нужна, так как в сбалансированной последовательности каждая открывающая скобка предшествует парной ей закрывающей скобке. Следовательно, номер любой открывающей скобки окажется в стеке раньше, чем произойдёт операция его извлечения на шаге (3). По достижении конца файла стек станет пустым, поскольку открывающих и закрывающих скобок в правильном входном тексте поровну.

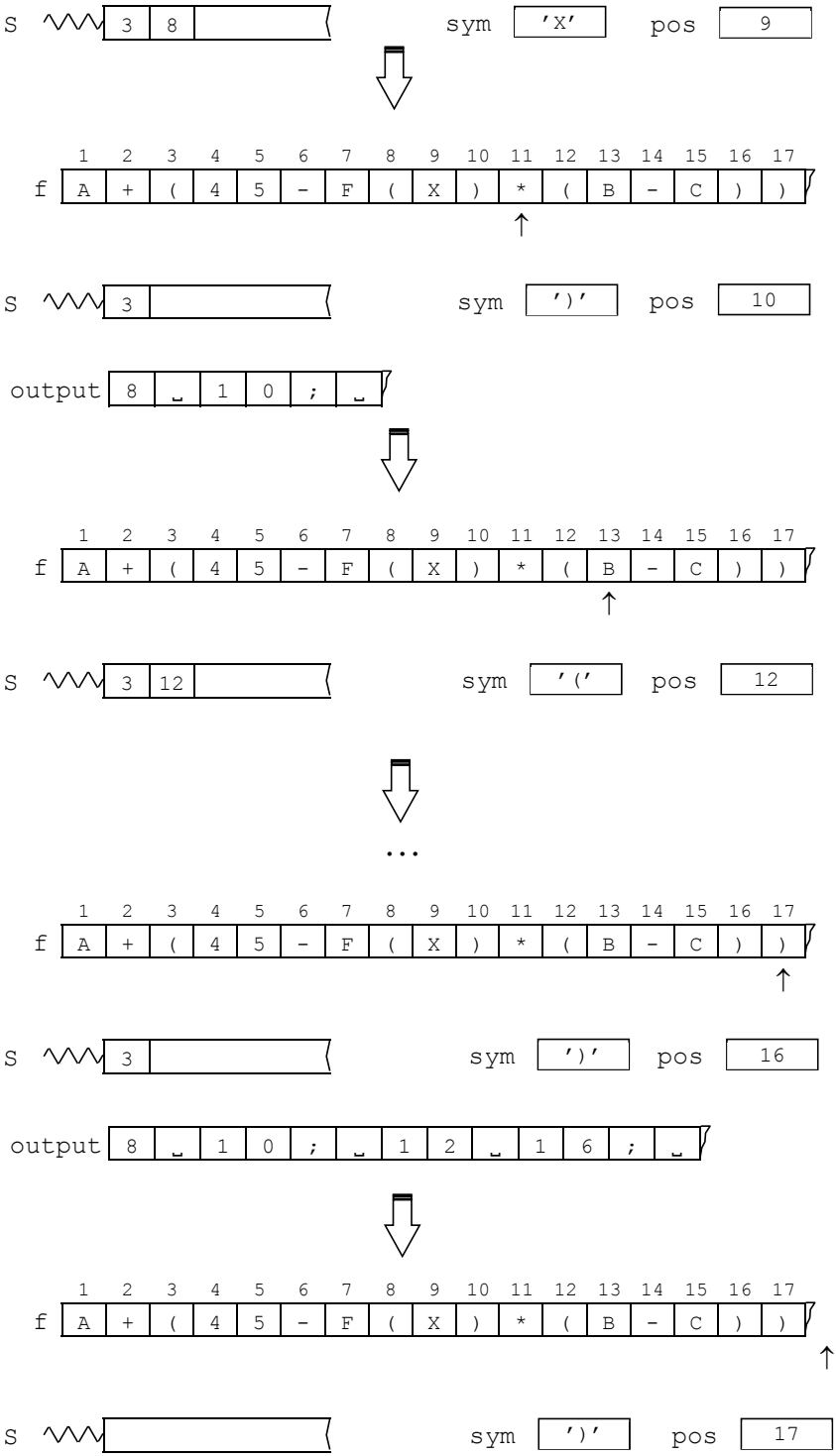
Изобразим на рисунке последовательность состояний переменных f, output, pos, sym, S при обработке текста A+(45-F(X)*(B-C)). Файловые переменные будем изображать в виде ленты, разделённой на ячейки. В одной ячейке располагается одна компонента файла. Если файл открыт для чтения, то стрелка, указывающая снизу на ячейку, отмечает компоненту, которая будет считана при очередном обращении к процедуре read. Если стрелка указывает на место за последней ячейкой, то чтение очередной компоненты невозможно, так как достигнут конец файла (функция eof в этой ситуации возвращает значение «истина»). Стек представим в виде горизонтальной трубки, открытой с правого конца (вершина стека справа). Левый конец («дно» стека) отметим ломаной линией, символизирующей «пружину» для выталкивания элементов из стека.

Перед чтением первого символа из f значение переменной sym не определено, pos=0, стек S и файл output пусты.

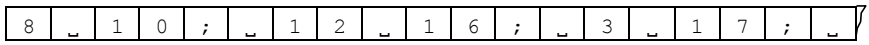


[1] Если стек представлен не массивом, а динамической цепочкой (см. задачу 10), то его размер заранее не ограничен. Стек может расти, пока у Паскаль-машины хватает ресурсов для создания нового динамического объекта – очередного звена стека.


output
sym
pos
0
S
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
f A + (4 5 - F (X) * (B - C))
sym 'A'
pos 1
sym '+'
pos 2
3
sym '('
pos 3
...
8
sym '('
pos 8


S
3
8
sym
'X'
pos
9
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
f
A + (4 5 - F (X) * (B - C))
S
3
sym
')'
pos
10
output
8 ␣ 1 0 ; ␣
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
f
A + (4 5 - F (X) * (B - C))
S
3
12
sym
'('
pos
12
...
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
f
A + (4 5 - F (X) * (B - C))
S
3
sym
')'
pos
16
output
8 ␣ 1 0 ; ␣ 1 2 ␣ 1 6 ; ␣
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
f
A + (4 5 - F (X) * (B - C))
S
sym
')'
pos
17

output



После того, как входная последовательность обработана (т.е. достигнут конец файла f), стек S пуст, в файле output результат – номера позиций парных скобок.

Заметим, что текст может быть разбит на строки, т.е. кроме обычных символов (типа char) может встречаться специальный символ – маркер конца строки. При чтении этого маркера значение переменной pos изменять не нужно, так как он не является «видимым» символом и не занимает отдельную позицию в исходной последовательности. Пропустить маркер конца строки и перейти к следующей строке можно с помощью оператора readln(f).

```
procedure pairs(f:text);
{ печатает пары номеров позиций открывающих и закрывающих
  скобок из текста  f  в порядке возрастания номеров  для
  закрывающих скобок}
var sym: char;
    pos,i: integer;
    S: stack;
begin reset(f);
      init_stack(S); pos:=0;
  if not eof(f) then
  repeat
    if eoln(f) then readln(f)
    else
      begin read(f,sym); pos:=pos+1;
        if sym = '(' then push(S, pos)
          {номер позиции открывающей скобки в стек}
        else if sym = ')' then
          {номер позиции открывающей скобки,
           соответствующей данной закрывающей, взять из
           стека и напечатать пару номеров}
          begin  pop(S,i);
            write(i,'␣',pos,';␣')
          end;
      end;
  until  eof(f);
  writeln;
end {pairs};
```

Задача 10. Программа на языке Турбо Паскаль. Программа запрашивает у пользователя имя текстового файла, в котором находится последовательность символов, сбалансированная по круглым скобкам, и выводит на экран номера парных скобок как в задаче 9.

Решение

Опишем в программе переменную `name` типа `string` и присвоим ей введённое пользователем имя внешнего файла, в котором хранится исходный текст. Опишем также переменную `g` типа `text`; с помощью процедуры `assign` свяжем файловую переменную `g` с внешним файлом `name`. Чтобы напечатать последовательность номеров парных скобок, воспользуемся процедурой `pairs` из задачи 9. Поскольку в `pairs` используется стек, мы должны реализовать в нашей программе соответствующие структуру данных и операции, т.е. описать тип `stack` и процедуры, необходимые для работы со стеком. Реализуем стек с помощью списка без заглавного звена. Процедура `init_stack(S)` делает список `S` пустым; `push(S,e)` вставляет в начало списка `S` элемент `e` – это операция «положить элемент в стек»; `pop(S,e)` удаляет из списка `S` первый элемент и присваивает его переменной `e` – это операция «вытолкнуть (взять) элемент из стека». После того, как работа с файлом закончена, его следует закрыть обращением к процедуре `close`.

```
program parentheses(input, output);
type elemtype=integer;
     link=↑node;
     node=record
            elem:elemtype;
            next:link
          end;
     stack=link;

procedure  init_stack(var S:stack);
begin
  S:=nil
end;

procedure push(var S:stack; e:elemtype);
var p:link;
begin
  new(p);
  p↑.elem:=e;
  p↑.next:=S;
  S:=p
end;

procedure pop(var S:stack; var e:elemtype);
var p:link;
begin
  p:=S;
  S:=S↑.next;
  e:=p↑.elem;
  dispose(p)
end;
```

```
procedure pairs(var f:text);
var sym  :char;
    pos,i:integer;
    S:stack;
begin
  reset(f); pos:=0; init_stack(S);
  if not eof(f) then
  repeat
    if eoln(f) then readln(f)
    else
      begin read(f,sym); pos:=pos+1;
        if      sym='(' then push(S,pos)
        else if sym=')' then
          begin pop(S,i);
                write(i,'␣',pos,';␣');
          end
      end
  until eof(f);
  writeln
end;

var name:string;
    g:text;
begin
  write('Введите имя файла:␣');
  readln(name);
  assign(g,name);
  pairs(g);
  close(g);
end.
```

Сделаем два замечания к данной программе. Во-первых, в ней использованы средства языка Турбо Паскаль [5], которых нет в стандартном Паскале: `assign`, `close`, `string`. Во-вторых, программа будет корректно работать только с файлами, сбалансированными по скобкам. Если баланса скобок нет, во время выполнения может произойти ошибка, связанная с попыткой взять элемент из пустого стека. Примером файла, подходящего для обработки данной программой, является файл, содержащий её исходный текст.

Задача 11. Описать функцию `create`, которая создаёт двоичное дерево и возвращает указатель на его корень или **`nil`**, если дерево пусто. Дерево создаётся по его записи в виде иерархического списка (как на странице 32) из строки текстового файла `input`**.** В роли открывающей и закрывающей скобок в записи дерева используются символы '<' и '>'. Предполагается, что запись дерева не содержит ошибок. Примеры записи деревьев: <> – пустое дерево,

<a,<b,<>,<>>,<c,<>,<>>> – **непустое** дерево с корнем a, левым сыном b и правым сыном c.

Решение

```
function create:tree;
{создает дерево из строки текстового файла
                      input; elemtype=char}
var t:tree; c:char;
begin
  read(c); {считываем открывающую скобку}
  read(c); {следующий после открывающей скобки символ}
  if c='>' then create:=nil {возвращаем пустое дерево}
  else
    begin {c является корневым элементом}
       new(t); t↑.elem :=c;
       read(c);{считываем запятую после корневого
                элемента}
       t↑.left:= create; {создаём левое поддерево}
       read(c); {считываем запятую после левого
                 поддерева}
       t↑.right:= create;{создаём правое поддерево}
       read(c); {считываем закрывающую скобку > }
       create:=t
    end
end; {create}
```

В [6-10] можно найти дополнительные сведения и примеры динамических структур данных. В [11] содержится широкий набор задач для самостоятельного решения.

Задания практикума на ЭВМ

В этом разделе приводятся варианты заданий практикума на ЭВМ по теме «Динамические структуры данных».

Задание 1. Представление текста в виде списка слов.

Постановка задачи

Дан текст, состоящий из непустой последовательности слов из латинских букв, разделённых запятыми, за последним словом – точка; каждое слово состоит не более, чем из 10 символов. Требуется найти k – количество слов, которые удовлетворяют условию, заданному вариантом задания. В некоторых вариантах кроме текста задаётся ещё одна буква.

Варианты

Подсчитать количество слов, которые:

1) имеют последней буквой заданную;
2) содержат заданную букву ровно два раза;
3) содержат заданную букву не менее двух раз;
4) первой и последней буквой имеют одну и ту же букву;
5) имеют длину не менее пяти букв;
6) имеют первой буквой заданную и ещё хотя бы одно её вхождение;
7) имеют последней буквой заданную и ещё хотя бы одно её вхождение;
8) содержат заданную букву, но ни первой, ни последней;
9) не имеют последней буквой заданную;
10) имеют длину не более трёх букв;
11) имеют первой и последней буквой одну и ту же заданную;
12) не имеют первой и последней буквой одну и ту же заданную.

Методические указания

1. Вводить последовательность слов следует посимвольно. Для представления слов в программе использовать строковый тип подходящей длины. Слово располагать в начальных компонентах строки, не используемые компоненты заполнять пробелами.

2. Для внутреннего представления текста использовать список с заглавным звеном. Звено списка состоит из трёх полей: первое – строка, в которой хранится слово, второе – длина данного слова, третье – указатель на следующее звено.

3. Для вставки очередного слова в список определить отдельную процедуру; для подсчёта слов, удовлетворяющих условию, описать функцию. После построения списка напечатать слова, находящиеся в звеньях списка, используя процедуру `print`.

Задание 2. Представление выражений в виде двоичного дерева.

Постановка задачи

Во входном текстовом файле (можно использовать input) задано арифметическое выражение. Для заданного выражения построить дерево. Используя подходящие способы обхода (прямой, обратный или симметричный) для дерева-выражения выполнить действия:

- вычислить и напечатать значение выражения при заданных значениях переменных;
- напечатать строку-выражение (с переменными и с числами) в указанной форме (инфиксной, префиксной или постфиксной);

- напечатать строку-выражение в указанной форме и без переменных – вместо переменных в выражении должны быть подставлены их значения.

При выводе выражения в инфиксной форме лишние скобки желательно не печатать.

Допустимые входные выражения определяются следующей грамматикой (в форме БНФ):

```
<выражение>::= <слагаемое>{<знак_+_или_-_> <слагаемое> }
<знак_+_или_-_>::= + |  -
<слагаемое>::= <множитель>  { * <множитель> }
<множитель>::=(<выражение>) | <переменная-буква> | <цифра>
```

Дерево, соответствующее заданному выражению, во внутренних вершинах содержит символы – знаки операций, а в листовых вершинах символы-цифры или символы-переменные. Можно расширить набор операций, используемых в выражении, например, добавив в него возведение в степень (обозначается символом '^') и деление нацело (обозначается '/'). При построении дерева учитывать естественный приоритет операций (у операции '^' наивысший приоритет, у '+' и '–' самый низкий) и то, что операции '–', '/' левоассоциативны, а '^' правоассоциативна. Можно упростить задачу, потребовав, чтобы во входном выражении все сложные операнды были заключены в скобки.

Варианты

Конкретное задание для студента определяется комбинацией вариантов из А, Б, В, Г.

А. Форма задания исходного арифметического выражения

1. Префиксная
2. Инфиксная
3. Постфиксная

Б. Операнды арифметических операций

1. Цифры от '0' до '9'
2. Переменные, имеющие целые значения. Каждая из них задается латинской буквой из диапазона от 'A' до 'Z'. Для работы с такими операндами можно использовать массив указателей на значения. Индексы в массиве : 'A' .. 'Z'. Указатель **nil** будет означать отсутствие соответствующей переменной в выражении
3. Цифры или переменные (комбинация Б.1 и Б.2)

В. Форма выражения (с переменными), выдаваемого на печать

1. Постфиксная
2. Префиксная
3. Инфиксная

В одном задании конкретный вариант В должен отличаться от А.

Г. Форма выражения (без переменных), выдаваемого на печать:

1. Инфиксная
2. Постфиксная
3. Префиксная

Конкретный вариант Г в задании должен отличаться от А и В. Таким образом, будет проведена работа со всеми тремя формами представления выражения, исходное выражение и два напечатанных – одно и то же выражение в трех разных формах записи.

Задание 3. Представление таблиц в виде дерева.

Постановка задачи

Дан текстовый файл, содержащий *слова* и *разделители*. Под словом понимается непустая последовательность из латинских букв и цифр. Символы, отличные от цифр и латинских букв, являются разделителями. Между соседними словами есть хотя бы один разделитель. (Разделители также могут быть перед первым словом и после последнего).

Требуется создать новый текстовый файл в котором будет содержаться таблица "встречаемости" слов исходного файла. Каждая строка таблицы содержит слово, количество его вхождений в исходный файл и "частоту" - отношение количества вхождений данного слова к общему количеству слов в исходном файле. Слова упорядочены по алфавиту (иначе говоря, расположены в лексикографическом порядке). Каждое слово встречается в таблице только один раз.

Пример. Для файла

```
program p;
{the shortest pascal program}
begin end.{program end}
```

файл-результат может выглядеть так:

```
Слово       Кол-во      Частота
begin            1                0.1
end              2                0.2
p                1                0.1
pascal           1                0.1
program          3                0.3
shortest         1                0.1
the              1                0.1
Всего:          10                1.0
```

Для внутреннего представления таблицы в программе требуется использовать определенный вид дерева, задаваемый вариантом. Кроме того, вариант

определяет, какая дополнительная информация, характеризующая работу с деревом, должна быть выведена в файл `output`. Можно усложнить задачу, задав не один исходный файл, а два: все узлы построенного по первому файлу дерева, содержащие слова из второго файла, должны быть удалены. (Удаление узла может привести к необходимости коррекции дерева, чтобы сохранить его свойства).

Варианты

А. Внутреннее представление таблиц

1. Двоичное дерево поиска (в нем слева от каждой вершины-слова должны находиться только те слова, что предшествуют этому слову по алфавиту, а справа - следующие за ним по алфавиту)
2. 2-3 -деревья (см. [6])
3. АВЛ-деревья (см. [7, 12])
4. Красно-черные деревья (см. [12])

Б. Дополнительная информация о дереве

1. Нет дополнительной информации
2. Высота дерева и количество узлов в нем
3. Количество узлов, затронутых операциями балансировки ("вращения", "переливания", "перекраски" и т.п.)
4. Комбинация вариантов 2-3

Методические указания

1. Для внутреннего представления слова можно использовать список или массив-строку. В последнем случае в программе описывается константа n ($n \geq 20$), ограничивающая длину слова. Если вводимое слово превышает установленную длину, то учитываются только его первые n символов, остальные игнорируются.

2. Структуру узлов (вершин) дерева определить с учетом особенностей выбранного вида деревьев. (В узле, например, может присутствовать информация о сбалансированности). Кроме того, в узле дерева можно хранить не само слово (ключ), а ссылку на него.

3. Описать в программе следующие процедуры или функции:
- чтение очередного слова, дополнение его справа пробелами (если для представления слова используется массив);
- поиск слова в дереве;
- вставка (удаление) слова в дерево;
- просмотр дерева и печать слов в нужном порядке;

4. Программа должна быть протестирована на различных исходных данных. Для этого подготовить несколько своих небольших внешних файлов.

5. Сравнить полученные экспериментальные данные (определяемые пунктом Б варианта задания) с теоретическими оценками из [6-7, 12].

6. Работа с файлами в языке Турбо Паскаль имеет свои особенности. В языке нет внутренних файлов, все файлы – внешние, расположенные во внешней памяти (например, на жестком диске компьютера). Перечислять внешние файлы в заголовке программы не нужно, и заголовок выглядит так:

```
program <имя программы>;
```

Поскольку названия файлов во внешней памяти (например, *C:\DATA\SOURCE.TXT*) записываются по правилам файловой системы и не являются идентификаторами в смысле языка Паскаль, то их нельзя использовать в качестве имён (файловых) переменных. В связи с этим в Турбо Паскале названию дискового файла ставят в соответствие некоторое имя (например, *t*), правильное с точки зрения языка, и далее в программе пользуются только этим именем: именно его описывают как имя файловой переменной и указывают в процедурах и функциях `reset`, `rewrite`, `read`, `write`, `eof`, `eoln` и т.п. Соответствие же между этим именем и названием дискового файла устанавливается с помощью процедуры `assign`:

```
assign(t, 'C:\DATA\SOURCE.TXT')
```

(второй параметр здесь – строка), причем обращение к этой процедуре должно быть выполнено до любых других операций над файлом. Если задана пустая строка в качестве внешнего имени, то внутреннее имя файла связывается со стандартным файлом ввода (`input` при установке режима чтения с помощью `reset`) или вывода (`output` при установке режима записи посредством `rewrite`).

Завершив работу с файлом, его следует "закрыть" с помощью процедуры `close(t)`.

С каждой выполняемой программой связан так называемый "текущий" каталог. Обычно это тот же каталог (по-другому – папка или директория), из которого программа запущена на выполнение. Если файл, с которым должна работать программа, хранится в текущем каталоге, можно не указывать его полное путевое имя, т.е. вместо `assign(t,'C:\DATA\SOURCE.TXT')` писать `assign(t, 'SOURCE.TXT')`.

Программа может изменить свой текущий каталог с помощью процедуры `ChDir(Path)`, где `Path` – путь к каталогу, который становится текущим, например, `ChDir('C:\NEW_DATA')`. Помимо строковых констант для указания соответствующих параметров можно использовать строковые переменные.

Литература

1. Pascal ISO 7185:1990. URL : http://www.pascal-central.com/docs/iso7185.pdf.
2. *Вылиток А.А.* Металингвистические формулы и синтаксические диаграммы. – М.: МГУ, МАКС Пресс, 2012. – 24 с.
3. *Абрамов В.Г., Трифонов Н.П., Трифонова Г.Н.* Введение в язык Паскаль. – М.**:** КНОРУС, 2011. – 384 с.
4. *Йенсен К*., *Вирт Н*. Паскаль: руководство для пользователя. – М.: Финансы и статистика. 1989. –255с
5. *Фаронов В.В*. Turbo Pascal 7.0 : учебный курс. – М. : КНОРУС, 2009.
6. *Ахо А., Хопкрофт Д., Ульман Д.* Структуры данных и алгоритмы. М.: Издательский дом «Вильямс», 2000. – 384с.
7. *Вирт Н.* Алгоритмы и структуры данных. – СПб.: Невский Диалект, 2001. – 352 с.
8. *Кнут Д.* Искусство программирования. Том1. М.: Издательский дом «Вильямс», 2000. – 720с.
9. *Лавров С*. Программирование. Математические основы, средства, теория. – СПб.: БХВ-Петербург, 2001. – 320 с.
10. *Любимский Э.З., Мартынюк В.В., Трифонов Н.П.* Программирование. – М.: Наука, 1980. – 608с.
11. *Пильщиков В.Н*. Язык Паскаль: упражнения и задачи. – М.: Научный мир, 2003. – 224с.
12. *Сенюкова О.В.* Сбалансированные деревья поиска. – М.: МГУ, МАКС Пресс, 2014. – 68 с.

Приложение

Пример реализации задания 1

Пусть требуется найти, сколько в тексте слов длиной пять букв и в которых заданная буква встречается не менее двух раз.

```
program sample(input, output);
const n=10;
      template='␣␣␣␣␣␣␣␣␣␣';
type  line = packed array[1..n] of char;
      size=1..10;
      link= ↑node;
      node = record
               word: line;
               length: size;
               next: link
             end;
      list=link;
var L: list;
    p: link;
    c: char;
 word: line;
  k,i: integer;
procedure insert(p: link; word: line; length: size);
{создаёт звено для слова word длины length и вставляет
его в конец списка (после звена, на которое указывает p)}
var q: link;
begin new(q);
  q↑.word:= word;
  q↑.length:=length;
  q↑.next:= nil{= p↑.next};
  p↑.next:= q
end {insert};

procedure print(p: list);
{печатает слова из списка с заглавным звеном}
begin p:=p↑.next;
  while p < > nil do
    begin write(p↑.word,'␣');
          p:=p↑.next
    end;
  writeln;
end {print};

function amount(p: list; letter: char;
                         length: size):integer;
```

```
{подсчитывает количество слов в списке, удовлетворяющих
 условию: длина равна length и буква letter входит в
 слово не менее двух раз}
var k,m,i:integer;
begin
  k:=0;   p:= p↑.next;
  while p < > nil do
    begin
      if p↑.length=length then
        begin m:=0;
          for i:=1 to length do
            if p↑.word[i]= letter then m:= m+1;
          if m>= 2 then k:= k+1
        end;
      p:=p↑.next
    end;
  amount:= k
end {amount};

begin  {построение заглавного звена списка}
  new(L); L↑.next:= nil;
  writeln('Введите текст:');
  p:=L; {p установили на начало пустого списка}
  repeat i:=0; word:= template; read(c);
    {ввод очередного слова word из файла input}
    repeat i:=i+1; word[i]:= c; read(c);
    until (c = ',') or (c='.');
    insert(p,word,i);
    p:=p↑.next; {p указывает на последнее звено}
  until c ='.';
  readln;
  print(L); {печать введённого текста}
  write('Задайте букву: ␣');
  readln(c);
{обработка списка и вывод результата}
  writeln;
  writeln('Результат: k=', amount(L,c,5));
  writeln('=======================================');
end.
```

Замечание. Идентификаторы `word` и `length` являются стандартными именами в языке Турбо Паскаль. В нашей программе они переопределены и имеют другой смысл.

Содержание

Учебно-методическое пособие

ВЫЛИТОК Алексей Александрович
МАТВЕЕВА Тамара Константиновна

ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ. ЗАДАНИЕ ПРАКТИКУМА. ЯЗЫК ПАСКАЛЬ

Учебно-методическое пособие
для студентов 1 курса

Издание второе, переработанное и дополненное

Издательство «МАКС Пресс»
Главный редактор: *Е.М. Бугачева*

Отпечатано с готового оригинал-макета
Подписано в печать 08.08.2022 г.
Формат 60 x 90 1/16. Усл.печ.л. 4,0.
Тираж 50 экз. Заказ 108.

Издательство ООО “МАКС Пресс”
Лицензия ИД N 00510 от 01.12.99 г.

119992, ГСП-2, Москва, Ленинские горы,
МГУ им. М.В. Ломоносова, 2-й учебный корпус, 527 к.
Тел. 8(495)939-3890/91. Тел./Факс 8(495)939-3891.

Отпечатано в полном соответствии с качеством
предоставленных материалов в ООО «Фотоэксперт»
109316, г. Москва, Волгоградский проспект, д. 42,
корп. 5, эт. 1, пом. I, ком. 6.3-23Н