Глава 13. Динамические структуры данных

Динамические структуры данных – это структуры данных, память под которые выделяется и освобождается программистом по мере необходимости. Динамические структуры данных в процессе существования в памяти могут изменять не только число составляющих их элементов, но и характер связей между элементами.

Википедия

В изменениях мы находим своё предназначение.
Гераклит Эфесский, V век до н.э.

В стандарте Паскаля динамическими структурами данных являются только файлы, которые, однако, расположены на внешней памяти ЭВМ и Паскаль-программе не принадлежат. В языке Free Pascal упоминались динамические массивы, которые, впрочем, могли увеличивать свой размер только «скачком», порождая новую переменную бóльшего размера, а затем освобождая память старой переменной.

В то же время, как мы уже говорили, есть много задач, которые должны вводить и обрабатывать данные переменного объёма. Для хранения этих данных и используются различные динамические структуры, такие, как очереди, списки, деревья, таблицы и т.д. Таких типов данных в самом Паскале нет, но в других языках они есть. Например, в языке Lisp есть списки, во многих языках есть таблицы и т.д. Правда, их реализация чаще всего скрыта внутри языков, и как они устроены на *внутреннем уровне* неизвестно. В то же время для образовательных целей, конечно, хорошо знать, как там всё сделано «под капотом».

Исходя из этого, мы будем реализовывать динамические структуры данных на Паскале. Для каждой такой структуры мы сначала определим, как будем хранить её компоненты в переменных Паскаля (это так называемая структура хранения). Затем мы реализуем операции над динамической структурой в виде подпрограмм. Такой подход позволит оценить эффективность реализации динамической структуры данных как по занимаемой памяти, так и по быстродействию. С другой стороны понятно, что «как-то так же» эти структуры должны быть реализованы и в тех языках, где они есть (чудес не бывает).

Например, в языке PascalABC.NET можно использовать очередь, реализованную в виде так называемой *коллекции* системы программирования Microsoft.NET, которая имеет богатый набор разных стандартных библиотек. Все операции над очередью выполняются как вызовы соответствующих процедур и функций из этих библиотек, так что читатель о внутреннем устройстве очереди ничего и не узнает. Всё как в мультфильме «Вовка в тридевятом царстве»: «Вы что, за меня и программы писать будете?»

Каждую динамическую структуру данных мы сначала будем описывать на абстрактном уровне, фиксируя её главные черты, а потом делать реализацию на конкретном языке программирования. Понятно, что эту реализацию можно сделать по-разному и придётся оценить эффективность каждой такой реализации.

Сначала мы рассмотрим простейшие, *линейные* динамические структуры данных.

13.1. Очереди

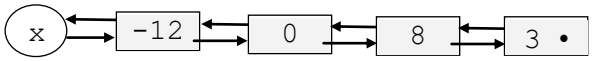
Очередь – абстрактный упорядоченный тип данных с дисциплиной доступа к элементам «первый пришёл – первый вышел» (FIFO –First In, First Out). Добавление элемента возможно лишь в конец очереди, выборка – только из начала очереди, при этом выбранный элемент из очереди удаляется.

Википедия

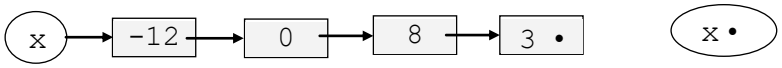
Итак, абстрактная очередь – это линейная упорядоченная динамическая структура данных, состоящая из элементов, число элементов есть длина очереди. Каждый элемент содержит некоторые данные (очередь введённых с клавиатуры символов, очередь студентов в буфете и т.д.), а также ссылки на другие элементы этой очерели. Очередь может быть и пустой, а также увеличивать или уменьшать свою длину. Удаляться элементы могут только из начала очереди, а добавляться только в конец. Остальные свойства абстрактной очереди остаются неопределёнными:

1. Могут ли элементы очереди быть разного типа и, следовательно, разного размера? Вот стоит очередь студентов в буфете, кажется что все её элементы одинаковы (по поведению в очереди). Однако потом замечаешь, что к одному студенту часто подходят его приятели и встают перед ним. Да это же в очереди стоит не один студент, а группа студентов! Для простоты будем в дальнейшем предполагать, что все элементы конкретной очереди одного типа, в примерах будем рассматривать очередь целых чисел.
2. Могут ли элементы очереди меняться местами или можно ли уйти из средины очереди (в магазинной очереди это вполне возможно)? Это, конечно, будет не совсем «нормальная» очередь, однако можно написать специальные операции, которые будут это делать.
3. Может ли одна очередь одновременно стоять к двум кассирам (или двум продавцам)? В жизни это встречается (стоящий в очереди первым отправляется к освободившемуся продавцу). Отметим, что сейчас в большинстве учреждений, где более одного «менеджера», организована именно такая «электронная» очередь. Впрочем, в «электронной» очереди обычно стоят элементы разных типов (разные посетители пришли по разным делам и должны попасть к разным «менеджерам», да и очередь ли это вообще? 🙄).
4. Может ли очередь увеличиваться сразу на несколько элементов? По аналогии с магазином, покупателей могут запускать с улицы порциями (например, по 10 человек). Вспомним, что так же вела себя очередь символов из потока `input`, она увеличивалась сразу на длину очередной введённой строки символов.

Отдельно рассмотрим вопрос упорядоченности. Вообще говоря, каждый элемент очереди знает, кто стоит перед ним, и кто за ним (будем, как обычно, рисовать это стрелочками). Для первого элемента (перед ним никого нет) нарисуем стрелку на «продавца» (обозначим его `x`, он знает первого в очереди), а для последнего (после него никого нет) будем вместо стрелочки рисовать жирную точку:



Такая «строгая» упорядоченность, однако, бывает далеко не всегда. Находясь в очереди в магазине, мы, скорее всего, помним только, за кем мы стоим, но не помнит, кто стоит за нами. От этого, однако, очередь не «рассыпается», значит информация о том, кто стоит за нами избыточна, её можно не знать, но при желании восстановить. Действительно, досточно найти «крайнего» (или «последнего») и идти по очереди от конца к началу, пока не достигнем нужного человека. Здесь, однако, нехорошо, что надо знать последнего в очереди (на начало очереди указывает продавец `x`). Можно, однако, изменить очередь так, чтобы каждый элемент знал только, кто стоит за ним, при этом получим такую картинку (отдельно изобразим пустую очередь, там один продавец `x`):



Обладая известной фантазией, иногда можно представить себе такой список в виде каравана верблюдов. К хвосту первого верблюда привязан второй, к хвосту второго – третий и т.д., а первого верблюда ведёт караванщик `x`. И, вообще говоря, караван может быть и пустым 😕.

Вот теперь мы терпеливо стоим в очереди и ждём, когда продавец скажет: «Проходите на обслуживание» 🙂. Попробуем описать такую очередь на Паскале. Нам надо задать типы данных для трёх сущностей: 1) элемента очереди (`Elem`), 2) стрелки (`Link`) и 3) всей очереди (`Queue`):

```
type Link=↑Elem; { хранит стрелку → на элемент}
     Elem=record next: Link; n: integer end;
     Queue=Link; { тип для всей очереди }
var x,y: Queue; { две очереди }
```

В нашем случае мы определили всю очередь `Queue` просто как ссылку `Link` на (первый) элемент (это идентичные типы), но концептуально это разные вещи, не будем их путать. Кроме того, вскоре мы сделаем другую реализацию очереди, где это будут уже разные типы.

С стандарте Паскаля переменные-очереди x и y стазу после порождения будут иметь неопределённые значения. Для того, чтобы указать, что в начальный момент очереди пусты, присвоим им пустые ссылки:

```
x:=nil; y:=nil
```

Заметим, что все динамические структуры данных (файлы, списки, динамические массивы и т.д.) всегда порождаются *пустыми*, а лишь затем могут расти и уменьшаться.

Рассмотрим теперь реализацию *операций* над очередями. Часто необходимо узнать длину очереди (количество элементов в ней). По прагматике надо реализовать функцию, возвращающую одно целое число – длину очереди. У функции один параметр – очередь, он не меняется и маленький по размеру (одна ссылка), значить, надо передавать его по значению. Для пустой очереди, естественно, возвращаем ноль. Можно предложить такую реализацию этой операции (будем использовать язык Free Pascal, важные для понимания места отмечены значками ❶, ❷ и т.д.):

```
function Len(x: Queue): integer;
  var L: integer=0; ❶ { для длины очереди }
begin { ❶ L:=0 }
  while x<>nil do begin
   ❷ inc(L); ❸ x:=x↑.next; { к след. элементу }
  end;
  Len:=L
end;
```

Язык Free Pascal позволяет порождать переменные с заданными значениями ❶, тогда можно не писать в теле функции оператор `L:=0`.[1] Использован цикл с предусловием, так как очередь может быть пустой, тогда тело цикла не должно выполниться ни одного раза. Вместо оператора `L:=L+1` использована более компактная запись ❷ `inc(L)` для инкремента (увеличения на единицу) значения длины L.

Параметр x передан по значению, это копия фактического параметра, её можно спокойно менять, не боясь испортить оригинал, что мы и делаем в операторе ❸. Параметр x является своеобразным «менеджером», который «бежит» вдоль очереди, считая её элементы.

К сожалению, хотя параметр x и передан по значению, но это не гарантирует, что внутри подпрограммы очередь нельзя изменить. Например, после оператора `x↑.n:=0` значение первого стоящего в очереди целого числа изменяется на нулевое. Так происходит потому, что при передаче параметра по значению снимается копия не со всей очереди, а только со ссылки на начало этой очереди. Так что здесь Паскаль не может в полной мере помочь нам в безопасной обработке таких данных, так как он фактически ничего «не знает» ни о каких очередях 😕.

И ещё один тонкий момент: переменные в левой и правой части оператора ❸ совместимы по присваиванию, хотя и имеют разные типы. Переменная x имеет тип Queue, а переменная `x↑.next` – тип Link, нас спасает только то, что эти типы идентичны: `Queue=Link`.

Заметим, что для вычисления длины очереди мы использовали оператор цикла. Это самый эффективный метод, однако в некоторых языках нет оператора цикла, и приходится использовать рекурсию, например:

```
function Len(x: Queue): integer;
begin Len:=0;
  if x<>nil then Len:=1+Len(x↑.next)
end;
```

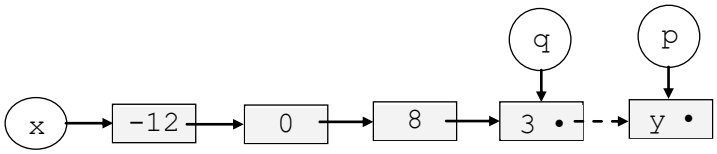
Следующей реализуем операцию постановки в конец очереди нового элемента. Дана (возможно, пустая) очередь и целое число, которое надо вставить в новый элемент и поставить этот элемент в конец очереди. Ясно, что по прагматике у нас намечается процедура:

---

[1] Как мы уже говорили, наша переменная L *автоматического* класса памяти, поэтому даже в языке Free Pascal по описанию `var L: integer;` она порождается не с *нулевым*, а с *неопределённым* значением.

```
procedure Enqueue(❶ var x: Queue; y: integer);
  var q: Link;       { для поиска конца очереди }
      p: Link;       { для порождения нового элемента }
begin
  new(p);            { новый элемент p↑ }
  p↑.n:=y;           { число y в последний элемент }
  p↑.next:=nil;      { сделать элемент последний }
  if x=nil then      { очередь была пуста ? }
    x:=p             { очередь изменилась! }
  else begin         { очередь не пуста, x не меняется }
    q:=x;            { q на начало очереди }
    while q↑.next<>nil do
      ❷ q:=q↑.next; { q↑ на последний элемент }
   ❸ q↑.next:=p      { новый элемент в конец }
  end
end;
```

Ссылка на начало очереди `x` редко (только для пустой очереди), но меняется, поэтому ❶ **`var`** обязателен! Включение нового элемента в пустую и не пустую очередь делается по-разному, поэтому в процедуре использован условный оператор. Для не пустой очереди «менеджер» `q` должен сначала найти и встать на последний элемент, в конце цикла в точке ❷ получается такая картинка:



Завершающий оператор ❸ присоединяет новый элемент в конец очереди.

Последней реализуем операцию удаления из очереди первого элемента. По аналогии с очередью в магазине, важно осознать, что удаление элемента состоит из двух этапов: *удаление* его из очереди и *обслуживание* этого элемента. Важно понять, что удаление из очереди должно *предшествовать* этапу обслуживания. Действительно, из нашего «магазинного» опыта понятно, что, если два продавца могут брать покупателей из одной очереди, то надо сначала взять (удалить) из очереди первого покупателя, а лишь потом обслужить его. В противном случае другой продавец вынужден будет пропустить первого и взять на обслуживание из очереди второго покупателя, что противоречит самой сути очереди.

Итак, у нас получается такое осознание задачи. Продавец (вызывая нашу операцию) спрашивает: «Кто следующий» и уводит первого покупателя на обслуживание (предварительно удаляя его из очереди). И лишь после обслуживания покупатель удаляется из магазина (т.е. наша операция сама процедуру `dispose` не вызывает, это будет делать продавец!). Таким образом, операция получает как параметр очередь, изменяет её, удаляя первый элемент (значит, **`var x`**!), и возвращает ссылку (`link`) на удалённый из очереди элемент. Ссылка маленькая по размеру, поэтому пишем функцию. Для пустой очереди наша функция будет, естественно, возвращать ссылку **`nil`**.

```
function Dequeue(var x: Queue): Link;
begin
  Dequeue:=x; { ссылка на первый элемент или nil }
  if x<>nil then { очередь не пуста }
     x:=x↑.next { удалить первый элемент }
end;
```

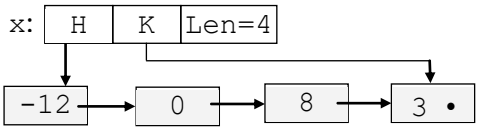
К сожалению, наша функция будет с побочным эффектом, так как она изменяет внешний объект – очередь, т.е. у нашей операции два результата: ссылка на первый элемент очереди и ссылка на саму изменённую очередь. В своё оправдание мы можем сказать, что во «внешнем мире» мы изменяем только объект, переданный нам по ссылке. Альтернативой является написание процедуры с двумя параметрами, принимаемыми по ссылке, что менее красиво.

Оценим теперь сложность реализованных операций. Под сложностью будем понимать количество базовых действий алгоритма (за действие можно взять сравнение двух величин между собой или операцию присваивания). Ясно, что это количество зависит от длины входных данных, в нашем случае от длины очереди (как обычно, обозначим эту длину N).

В двух первых операциях количество действий пропорционально длине очереди, в математике это обозначается как O(N) (читается «O большое от N»). Сложность третьей операции не зависит от длины очереди, это обозначается как O(1). Много это или мало? Конечно, всё зависит от типичной длины очереди, одно дело небольшой магазинчик, и совсем другое крупный гипермаркет. После некоторого периода использования может выясниться, что время работы первых двух операций недопустимо большой, их надо модифицировать (оптимизировать).

При модификации наших операций, однако, возникает принципиальная трудность. Дело в том, что к моменту модификации уже может существовать много программ, которые пользуются (вызывают) эти операции, найти и изменить все эти вызовы чрезвычайно трудоёмкая работа. Единственным выходом является сохранить неизменным синтаксис (т.е. внешний вид) и семантику вызовов операций, тогда все программы, использующие эти операции, менять не надо. В этом нам поможет инкапсуляция (сокрытие) внутренней реализации всех подпрограмм.

Итак, новая реализация очереди должна обеспечивать максимальную скорость всех операций O(1), для этого сначала надо изменить тип Queue. Теперь переменная этого типа должна хранить не только ссылку на *начало* очереди (H), но и ссылку на её *конец* (K), а также текущую *длину* очереди (Len):



Теперь получится такое описание типов:

```
type Link=↑Elem; { стрелка }
     Elem=record next: Link; n: integer end;   { элемент }
     Queue=record H,K: Link; Len: integer end;{ вся очередь }
var x,y: Queue;
```

Инициировать очередь (сделать её пустой) теперь более сложно:

```
x.H:=nil; ❶ x.K:=nil; x.Len:=0
```

Для инициализации очереди можно написать специальную процедуру, например:

```
procedure InitQueue(var x: Queue);
begin x.H:=nil; ❶ x.K:=nil; x.Len:=0 end;
```

Дальше мы увидим, что присваивание `❶ x.K:=nil` при инициализации этой очереди можно опустить. В языке Free Pascal можно также порождать очередь-запись с уже заданным значением, например:

```
var x: Queue=(H: nil; K: nil; Len: 0);
```

Давайте изменим операции для работы с очередью, заголовки подпрограмм при этом останутся неизменными. Сначала операция определение длины очереди:

```
function Len(x: Queue): integer;
begin Len:=x.Len end;
```

Теперь изменим операцию постановки нового элемента в конец очереди:

```
procedure Enqueue(var x: Queue; y: integer);
  var p: Link; { для порождения нового элемента }
begin
  new(p);                  { новый элемент p↑ }
  p↑.n:=y;                 { число y в последний элемент }
  p↑.next:=nil;            { сделать элемент последний }
  inc(x.Len);              { увеличить длину очереди }
  if x.H=nil
```

```
    then x.H:=p          { очередь пуста }
    else x.K↑.next:=p;   { новый элемент в конец }
  x.K:=p                 { K на последний элемент }
end;
```

Как видим, никакого цикла больше нет. И, наконец, операция удаления элемента из очереди:

```
function Dequeue(var x: Queue): Link;
begin
  FromQueue:=x.H;   { ссылка на элемент или nil }
  if x.H<>nil then begin  { очередь не пуста }
    x.H:=x.H↑.next;  { удалить первый элемент }
    x.Len:=x.Len-1   { уменьшить длину очереди }
  end
end;
```

Обратите внимание, что для пустой очереди значение поля `x.K` не анализируется, поэтому при удалении из очереди последнего элемента этому полю можно не присваивать **nil**. Вот теперь наши операции достигли максимума скорости обработки очереди O(1).

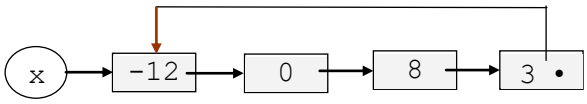
В программировании для хранения и обработки используются самые разнообразные очереди. Вспомним буфер ввода потока `input`, который является очередью символов из очередной введённой строки. Это образно можно сравнить с магазином, покупателей в который запускают порциями (не больше размера торгового зала). Далее, пока все покупатели в этой очереди не будут обслужены, новых в магазин не запускают. это очередь ограниченного размера, поэтому для её реализации лучше всего подходит обычный массив символов.

13.1.1. Кольцевая очередь

Кольцевые дороги
Никуда не ведут.

Рок-группа
«Смысловые галлюцинации»

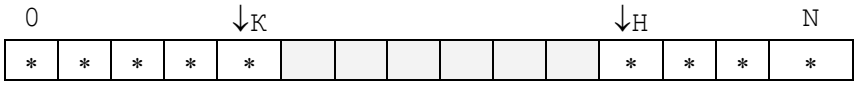
Иногда в программировании используются *кольцевые* очереди. При этом для односторонней очереди последний элемент ссылается на первый:



Для двусторонней очереди дополнительно первый элемент ссылается на последний.

Кроме того, часто кольцевой называют одностороннюю очередь ограниченного размера, для хранения которой можно использовать массив с заранее известным числом элементов. Это псевдодинамическая структура данных, так как память выделяется заранее для максимального числа элементов, а потом «заполняется по кольцу». Отметим, что так часто устроена запись с видеокамеры на устройство памяти.

Например, в некоторый момент очередь в массиве может иметь такой вид (`H` и `K` соответственно индексы начала и конца очереди, длина массива `N`, свободное место массива закрашено:



Видно, как очередь начинается с индекса `H`, затем продолжается «по кольцу» массива и заканчивается по индексу `K` (свободная часть очереди закрашена). Попробуем описать такую кольцевую очередь целых чисел. Пусть максимальный объём очереди `N>0`:

```
const N=1000;
type RingQueue=record
       H,K: 0..N-1;  { индексы начала и конца }
       Q: array[0..N-1] of integer;
     end;
```

```
var x,y: RingQueue;
```

Для пустой очереди x будем предполагать `x.H=-1`. Операцию определения длины такой очереди можно описать так:

```
function Len(var x: RingQueue): integer;
begin
  if x.H=-1 then Len:=0 else
  if x.H<=x.K then Len:=x.K-x.H+1
              else Len:=N+x.K-x.H+1
end;
```

Объясните, почему мы передали параметр x типа RingQueue по ссылке, хотя операция определения длины не меняет очередь.

Напишем теперь операцию постановки в очередь x нового элемента (числа y). Реализуем операцию в виде функции, которая возвращает значение **true** для успешно выполненной операции, и значение **false**, когда очередь переполнена и операция не выполнена.

```
function ToRingQueue(var x: RingQueue; y: integer): boolean;
  var z: integer;
begin
  z:=(x.K+1) mod N; ToRingQueue:=true;
  if x.H=-1 then begin { очередь пуста }
     x.H:=0; x.K:=0; x.Q[0]:=y
  end else
  if z<>x.H then begin
     x.K:=z; x.Q[z]:=y
  end else ToRingQueue:=false { очередь полна }
end;
```

Самостоятельно разработайте и напишите операцию удаления элемента из такой кольцевой очереди.

Обобщение очереди является структура данных дека (deq – Double Ended Queue, т.е очередь с двумя концами). Включение новых и исключения старых элементов можно производить на любом конце деки.

13.2. Стеки

Стек (stack) – абстрактный тип данных с дисциплиной доступа к элементам «последний пришёл – первый вышел» (LIFO – Last In, First Out). Добавление элемента возможно лишь в начало стека, выборка тоже только из начала стека, при этом выбранный элемент из стека удаляется.

Википедия

Стэк отличается от очереди только тем, что добавление и удаление элементов производится только с одного конца (вершины) стека. Эта структура данных часто используется в различных алгоритмах. Мы уже сталкивались с ней при рассмотрении порождения и уничтожения локальных переменных в подпрограммах. Стек настолько важен, что в большинстве компьютеров он реализован аппаратно. Это означает, что есть машинные команды для записи в стек и чтения данных из стека, таким образом, программно (как мы, например, делали для очереди), стек реализуется очень редко. Исходя их этого, подробное изучение этой структуры данных переносится в курс по архитектуре ЭВМ.

13.3. Списки

Связный список – динамическая структура данных, состоящая из узлов, со-

держащих данные и ссылки на следующий и/или предыдущий узел списка.

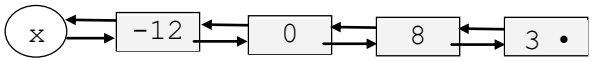
Википедия.

Порой мы видим многое, но не замечаем главного.

Конфуций, V век до н.э.

Абстрактный список (list) – это линейная упорядоченная динамическая структура данных, состоящая из узлов (или звеньев). Каждый узел списка хранит некоторые данные и не менее одной связи с другими узлами списка. В памяти компьютера связи реализуются в виде ссылок (указателей). Список может быть пустым (не содержать ни одного узла), а также увеличивать или уменьшать свою длину. Удаляться и добавляться узлы могут в любое место списка (в начало, в конец или в внутрь списка). Как обычно, остальные свойства списка зависят от реализации. Например, узлы списка могут быть разного типа (так называемые *гетерогенные* списки или кортежи) и даже, в свою очередь, быть списками (такие списки называются *иерархическими*).[i] [см. сноску в конце главы]

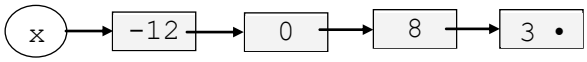
Для простоты будем предполагать, что данные во всех узлах списка одного типа, в примерах будут использоваться списки целых чисел. Внешне список очень похож на очередь, например, список



(-12,0,8,3) можно изобразить так:

В жизни многие очереди на самом деле списки: клиенты могут уходить из средины очереди, приходить и вставать в очередь перед знакомым, а также браться на обслуживание «по знакомству» из средины очереди. Как видно, список есть обобщение очереди и стека, в котором мы отказываемся от дисциплин обслуживания FIFO и LIFO.

Как и для очередей, ссылку вперёд можно удалить, она избыточна:



Первый вид списков называются двусвязными (а также двусторонними или двунаправленными, всё это синонимы), а вторые соответственно односвязными, односторонними или однонаправленными. По двусвязному списку можно двигаться в обе стороны, а по односвязному – только от начала к концу. С другой стороны, узел двусвязного списка занимает в памяти больше места, так как содержит дополнительную ссылку.

Существуют языки, которые «знают» о списках и могут их описывать, хранить и обрабатывать. Например, в одном из старейших языков программирования Lisp (LISP – LISt Processing language, язык обработки списков) приведённый выше список может записываться программе так:

```
(-12 . (0 . (8 . (3 . ()))))
```

Как видно, непустой список в языке Lisp (и почти во всех языках, имеющих такой тип данных) определяется рекурсивно, он состоит из первого элемента и «хвоста» списка, который, в свою очередь, является списком. Это связано с тем, что в функциональных языках, как правило, нет оператора цикла, и приходится строить алгоритмы с использованием рекурсии.

Символ «точка» используется вместо стрелочки (правда, точка ссылается не на следующий элемент списка, как в наших примерах, а на список-хвост). Пустые скобки `()` используются вместо ссылки **`nil`**.[1] Впрочем, в этом языке допускается и не такая «каноническая», а привычная «сокращённая» форма записи (которой, конечно, все и пользуются):

```
(-12 0 8 3)
```

Мы будем рассматривать простейший односвязный список с таким описанием в Паскале:

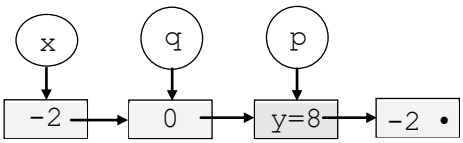
```
type Link=↑Node; { Link это стрелка на узел Node }
     Node=record next: Link; n: integer end;
     List=Link; { весь список }
var x,y: List;
```

---

[1] Юмористическая расшифровка названия языка LISP – Lots of Irritating Superfluous Parentheses (много раздражающих лишних скобок).

Операции над списками реализуются в Паскале в виде подпрограмм. Несколько бóльшую трудность в односторонних списках представляет операция удаления узла по сравнению с операцией вставки нового узла. Пусть, например, надо удалить из списка первый узел, содержащий число y, а если таких узлов в списке нет, то «ничего не делать». Реализуем эту операцию в виде процедуры:

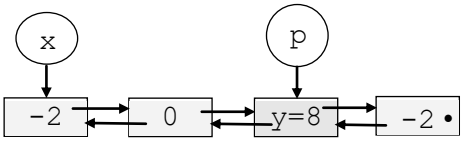
```
procedure DelNumber(var x: List; y: integer);
          { Удалить узел с числом y }
  var p,q: Link;
begin p:=x; { p «побежит» вдоль списка }
  while (p<>nil) and (p↑.n<>y) do begin
    q:=p; p:=p↑.next
              { q отстаёт от p на одну позицию }
  end; { p↑ на удаляемый узел, q↑ на предыдущий узел }
```




```
  if p<>nil then begin { нашли }
    if p=x then begin
{ удаляемый узел первый, x меняется ! }
      x:=x↑.next; dispose(p)
    end else begin
{ удаляемый узел не первый, x не меняется ! }
      q↑.next:=p↑.next; dispose(p)
    end
  end
end;
```

Рассмотрим, как та же операция реализуется для двустороннего списка:

```
type Link=↑Node;
     Node=record n: integer; next,prev: Link end;
     List=Link; { весь список }
var x,y: List;
procedure DelNumber(var x: List; y: integer);
  var p,q: Link;
begin p:=x; { p «побежит» вдоль списка }
  while (p<>nil) and (p↑.n<>y) do p:=p↑.next;
{ будет такая картина, p↑ на удаляемый узел }
```




```
  if p<>nil then begin { нашли y }
    if p=x then begin { удаляемый узел первый }
      x:=x↑.next; x↑.prev:=nil; dispose(p)
    end else begin { удаляемый узел не первый }
      p↑.prev↑.next:=p↑.next;
      p↑.next↑.prev:=p↑.prev; dispose(p)
    end
end;
```

Как видим, принципиально мало что изменилось, только приходится при удалении узла менять по две ссылки (вперёд и назад).

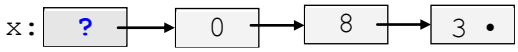
Заметим, что описанный нами выше однонаправленным список «официально» называется однонаправленным списком целых чисел без заглавного звена. Список с заглавным звеном описывается немного по другому. Вместо

```
type List=Link; { весь список }
```

используется описание

```
type List=Node; { весь список }
var x: List;
```

Другими словами, ссылку на начало списка хранит (фиктивное) заглавное звено x, в котором поле для хранения целого числа не используется (**?**), а поле ссылки указывает на первое звено списка:



Списки с заглавным звеном позволяют строить алгоритмы работы с ними более «элегантно». Реализуем предыдущую задачу удаления звена для такого списка на языке Ftee Pascal:

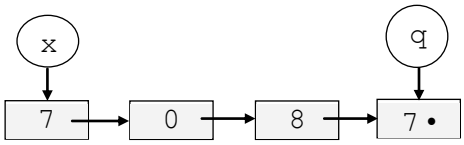
```
procedure DelNumber(var x: List; y: integer);
  var p,q: Link;
begin
  p:=@x; { p ссылка на x, на стандарте Паскаля так нельзя }
  while (p↑.next<>nil) and (p↑.next↑.n<>y) do
    p:=p↑.next; { p↑ на предыдущий узел }
  if p↑.next<>nil then begin { нашли }
    q:=p↑.next; p↑.next:=p↑.next↑.next;
    dispose(q)
  end
end;
```

Как видим, при наличии заглавного звена не надо отдельно рассматривать случай, когда удаляется первое звено списка. Далее будем рассматривать только списки без заглавного звена.

Теперь рассмотрим такую задачу. Дан односвязный список целый чисел. Если числа в первом и последнем узлах совпадают, то «удвоить» последний узел списка, иначе оставить список неизменным. Под «удвоением» узла будем понимать добавление сразу после него (или сразу перед ним) нового (new) узла с таким же числом. Особым случаем является пустой список, сделаем спецификацию, что он остаётся пустым (ничего не делаем). Особым является также случай из одного узла, но отдельно в программе его можно не рассматривать, он охватывается общим алгоритмом.

Никакого скалярного или ссылочного результата операция не имеет, значит нужно написать процедуру. Параметром процедуры является список (т.е. ссылка на начало списка), он нигогда не изменяется, его нужно передавать только по значению. Можно предложить такую реализацию этой операции:

```
procedure DoubleLastIfFirstEquelLast(x: List);
  var q: Link;
begin
  if x<>nil then begin { список не пуст }
    q:=x; { q↑ на начало списка, цикл while }
    while q↑.next<>nil do q:=q↑.next;
  { после цикла будет q↑ на последний узел }
```




```
      if x↑.n=q↑.n then begin
{ числа равны, новый узел добавляем после последнего }
        new(q↑.next); q:=q↑.next;
        q↑.next:=nil; q↑.n:=x↑.n
      end
  end { список не пуст }
```

```
end;
```

Разберём ещё одну, более сложную задачу. Дан односвязный список целых чисел, необходимо все минимальные числа удалить из списка, а все максимальные – продублировать. Осознание задачи: удалить, значит освободить память (dispose), а продублировать значит породить (new) и вставить сразу после (или сразу перед) ещё один узел с таким же числом.

Сделаем естественную спецификацию, что пустой список остаётся пустым. Далее, особым является случай, когда минимум совпадает с максимумом, т.е. все числа одинаковые (частный случай – список из одного узла). В этом случае будем, например, «минималистами», т.е. удалим *все* узлы списка.

Ясно, что список придётся просмотреть два раза, сначала найти минимум и максимум, а на втором проходе удалять и дублировать. При дублировании примем решение вставлять новый узел после узла с максимальным числом, тогда все узлы будут обрабатываться одинаково. А вот при удалении такой приём не сработает: при удалении из начала списка будет меняться ссылочная переменная x, указывающая на это начало (значит, **var** x), а при удалении из средины списка x не меняется. Следовательно, при втором проходе надо предусмотреть два случая.

Для реализации этой операции можно предложить такую процедуру:

```
procedure DelMinDoubleMax(var x: List);
  var min,max: integer; p,q: Link;
begin
if x<>nil then begin { список не пуст }
  min:=x↑.n; max:=min;
  p:=x↑.next; { p на второй узел }
❶ while p<>nil do begin { первый проход }
    if p↑.n<min then min:=p↑.n;
    if p↑.n>max then max:=p↑.n;
    p:=p↑.next
  end;
{ второй проход, первый этап, удаление min из начала списка}
  while (x<>nil) and ❷(x↑.n=min) do begin
    p:=x; ❸ x:=↑.next; dispose(p)
  end; { конец первого этапа }
{ второй проход, второй этап, первое число <> min }
  p:=x; { x больше менять нельзя }
{ p на узел, который можно дублировать, но не удалять }
  while p<>nil do begin
    if (p↑.next<>nil) and ❷(p↑.next↑.n=min) then
    begin { узел, следующий за текущим p, исключить }
      q:=p↑.next; { q на удаляемый узел }
      p↑.next:=q↑.next; dispose(q)
    end else
    if p↑.n=max then begin { дублировать }
      new(q); q↑.next:=p↑.next;
      q↑.n:=p↑.n; p↑.next:=q;
      p:=q↑.next { p за дублированный узел }
    end
    else p:=p↑.next { движение по списку }
  end { while }
end
end;
```

Отметим тонкие моменты. В точке ❶ нужен цикл с предусловием, так как в списке может быть только один элемент. В точках ❷ мы предполагаем, что включён механизм сокращённого вычисления логических выражений {$B-}, о котором говорилось в разделе 4.1. Когда включён режим

{$B+} будет ошибка, так как для значения `x=nil` будет вычисляться значение `x↑.n`, что повлечёт ошибку времени выполнения «попытка пойти по пустой ссылке» (null reference). В точке ❷ учащиеся часто меняют последовательность операторов (сначала удаляют, потом двигаются), что недопустимо.

В дополнение изучения работы со списками рассмотрим последнюю «хитрую» задачу. Необходимо инвертировать заданный односторонний список целых чисел, т.е. последнее число списка сделать первым, предпоследнее вторым и т.д. Ясно, что надо реализовать эту операцию в виде процедуры, причем параметр-список надо передавать по ссылке, так как он будет меняться (правда, не всегда, сами перечислите случаи, когда параметр меняться не будет). Ниже показано возможное решение, инвертирование производится за один проход. По списку согласованно двигаются три указателя p, q и r, которые «перебрасывают» ссылки на противоположные. Попытайтесь разобраться, как работает эта процедура.

```
procedure Invert(var x: List);
  var p,q,r: next;
begin
 ❶ if x<>nil then begin { не пуст }
    p:=x; q:=p↑.next; p↑.next:=nil;
    while q<>nil do begin
      r:=q↑.next; q↑.next:=p;
      p:=q; q:=r
    end;
    x:=p
  end
end;
```

В точке ❶ показана типичная обработка пустой структуры данных, когда процедура «ничего не делает». В языке Free Pascal (и во многих других) популярна другая форма обработки пустой структуры данных с оператором exit, который является неявным оператором **goto** перехода на завершающий **end**:

```
begin
  if x=nil then exit; { пусто – на выход }
{
  обработка непустой структуры данных
}
end;
```

В некоторые языки программирования (Lisp, Python и другие) изначально встроен тип список, и они умеют делать над списками часто используемые (но, конечно, далеко не все) операции. Чаще всего эти операции реализованы встроенными функциями. Скажем, в языке Lisp реализованы иерархические списки, когда элементом списка может быть снова список, списки задаются в круглых скобках. Например, вот встроенная функция, возвращающая длину списка:

```
(length '(a (3 b) 3 c)) ⇒ 4
```

А вот функции, возвращающие начальный узел (car) и «хвост» (cdr) списка:

```
(car '(a (2 b) (3 c) d) ⇒ (a)
(cdr '(a (2 b) (3 c) d) ⇒ ((2 b) (3 c) d)
```

При необходимости реализовать такие функции, например, на Паскале для нашего списка целых чисел встают трудные вопросы осознания и спецификации. Например, что значит «вернуть хвост списка»? Это ссылка на копию «хвоста» или ссылка на «хвост» исходного списка? Всё это тонкости работы функций на языке Lisp, которые мы, конечно, здесь рассматривать не будем.

В качестве хорошего упражнения напишите *рекурсивную* логическую функцию, сравнивающую два списка целых чисел:

```
function EQ(x,y: List): boolean;
```

13.4. Деревья

Дерево это абстрактная иерархическая структура данных в виде набор узлов с не ориентированными связями, не содержащими циклов.

Википедия

Если у тебя нет листьев, ствола или корней, то как ты можешь продолжать называть себя деревом?

Артур Голден. «Мемуары гейши»

Абстрактная структура данных граф определяется как набор (возможно пустой) вершин (узлов) и набор неориентированных рёбер (связей), соединяющих вершины. Отметим, что обычно неориентированная связь реализуется в программах двумя ссылками (прямой и обратной). Абстрактная структура данных дерево определяется как связный ациклический (т.е. не имеющий циклов, замкнутых путей) граф, один из узлов которого выделен как корень дерева (иногда всё это называется корневым деревом). Вершины дерева принято называть узлами, а рёбра – связями.

Отметим, что многие «деревья» не соответствуют этому определению. Например, всем известные генеалогические деревья «нашими» деревьями не являются, так как человек может иметь общего предка по нескольким восходящим путям, т.е. образуется замкнутый путь.

Итак, и списки, и деревья являются наборами узлов и связей между ними. В то же время это *принципиально разные* структуры данных, одни линейные, другие иерархические. Здесь всё дело в сути связи между узлами. Для линейных структур это связь «равного с равным», а для иерархических это связь «главный-подчинённый» (предок-потомок, отец-сын и т.п.). Правда, в отличие от генеалогического дерева, у сына только отец, но нет матери (однополое размножение 🙂).

Таким образом, в деревьях определено предпочтительное направление движения от предка к потомку (прямая ссылка считается главной, а обратная второстепенной). В зависимости от максимально возможного количества потомков у каждого узла производится классификация деревьев. Когда число потомков не более двух, это двоичные деревья (binary trees), не более трёх – троичные и т.д. При большом числе потомков (десятки и сотни) деревья называются сильно ветвящимися. В большинстве задач используются либо двоичные, либо сильно ветвящиеся деревья. Например, дерево, хранящее словарь, в каждом узле может быть столько потомков, сколько букв в алфавите.

Хорошим примером сильно ветвящегося дерева является файловая система. Листья такого дерева являются файлами, а остальные узлы – каталогами (директориями). Каждый каталог содержит ссылки на все свои узлы-подкаталоги и файлы-листья. В свою очередь, каждый файл «знает», в каком каталоге он находится, а каждый каталог – в каком родительском каталоге он расположен (по связям можно ходить сверху вниз и снизу вверх).

Правда, развитые файловые системы (например, широко распространённая NTFS), допускают нарушение чисто древовидной структуры. Вместо ссылки на файл-лист, каталог может содержать так называемую символическую ссылку (Symbolic Link), указывающую на какой-то узел (файл или каталог), расположенный в любом месте файловой системы. Таким образом, в файловой системе могут возникать замкнутые пути-циклы.

Потомки каждого узла считаются упорядоченными, например, у двоичного дерева это левый и правый потомок (когда мы смотрим на рисунок дерева). «Главный» узел, у которого нет предка, естественно, называется корнем дерева. Узлы, у которых нет ни одного потомка, называются листьями дерева. Дерево рекурсивная структура, из каждого узла налево и направо тоже «растут» деревья (называемые левым и правым поддеревом, возможно пустые). Для программиста дерево определяется ссылкой на его корень (или **`nil`** для пустого дерева).

Для простоты далее мы будем работать только с двоичными деревьями целых чисел, т.е. считать, что в каждом узле, кроме ссылок, хранится одно целое число. Традиционно деревья изображаются растущими от корня вниз, на рис. 13.1 показан пример такого дерева.

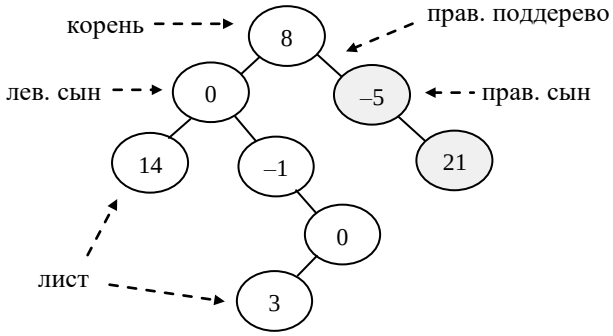


Рис. 13.1. Пример двоичного дерева целых чисел.

По аналогии с двухсторонними списками, связи между узлами дерева тоже двусторонние: узел ссылается на своего потомка, а потомок на родителя, т.е. «на самом деле» это две стрелки (ну, или одна двусторонняя), но на рисунках это стрелки обычно не изображаются.

Можно заметить, что связь от потомка к родителю является избыточной, её всегда можно восстановить, спускаясь по дереву от корня к нужному узлу. Эту связь явно реализуют, если нужно часто подниматься по древовидной структуре, хорошим примером является файловая система, где надо часто переходить из каталога в его родительский каталог.

Существуют способы записать дерево в виде линейной структуры (строки символов). Например, двоичное дерево можно изобразить в виде своеобразной формулы со скобками:

(<корень> (<левое поддерево>) (<правое поддерево>))

Тогда показанное на рис. 13.1 дерево можно записать так:

(8 (0 (14) (-1 () (0 (3) ()))) (-5 () (21)))

Заметим, что пустые поддеревья у не листьев заданы пустыми скобками, а у листа его пустые поддеревья не изображаются.

Реализуем двоичное дерево целых чисел на Паскале. Избыточную ссылку на родительский узел создавать не будем. Итак, опишем типы узла, стрелки и всего дерева:

```
type
  Link=↑Node; { Link это стрелка на Node }
  Node=record L,R: Link; n: integer end;
  BT=Link; { BT – всё дерево, ссылка на корень }
var x,y: BT;
```

Операции над деревьями, как обычно, реализуются в виде подпрограмм. В качестве первого примера рассмотрим операцию NumNodes, возвращающую число узлов дерева. Реализуем эту операцию в виде функции, которая получает по значению параметр-двоичное дерево:

```
function NumNodes(x: BT): integer;
begin
  if x=nil
    then NumNodes:=0  { пустое дерево }
    else NumNodes:=1+❶ NumNodes(❸x↑.L)
                    +❷ NumNodes(❸x↑.R)
end;
```

Здесь также важно понять, что решить эту задачу с помощью циклов трудно (мы сделаем это на примере нашей первой операции подсчёта числа узлов дерева). Это и означает, что деревья являются существенно рекурсивной структурой данных. Для обработки таких данных программисту нужно выработать в себе рекурсивный стиль мышления. В нашей задаче это такое рассуждение. Когда дерево пустое, операция сразу возвращает ноль. Для непустого дерево в нём есть по крайней мере одни узел – корень дерево. Таким образом, функция должна возвратить единицу (корень), прибавив к этой

единице число узлов из левого поддерева, которое возвращает вызов функции ❶ и число узлов из правого поддерева, которое возвращает вызов функции ❷.

Отметим тонкий момент: в точках ❸ тип фактического параметра `Link`, а тип формального параметра `BT`. Остаётся, однако, совместимость по присваиванию, так как параметр передаётся по значению, а эти типы объявлены идентичными: `BT=Link`.

Исследуем вопрос, какова будет глубина рекурсии нашей операции, т.е. какое максимальное количество раз функция вызовет сама себя, прежде, чем начнёт возвращать результат. Для этого определим понятие высоты дерева – число узлов в самой длинной ветви от корня до листьев. Для примера на рис. 13.1 высота дерева равна пяти (высота пустого дерева равна нулю).

Несколько определений для двоичного дерева.

Дерево называется полным, если в нём нет узлов, имеющих только одного потомка.

Дерево называется идеально сбалансированным, если для каждого узла число вершин в левом и правом поддеревьях отличаются не более, чем на единицу. Дерево называется сбалансированным (по высоте), если у каждого узла высота левого поддерева отличается от высоты правого поддерева не более, чем на единицу. В основном сбалансированность важна в так называемых деревьях двоичного поиска (Binary Seach Tree), о которых будет рассказано в следующей главе.

Разберёмся теперь, как высота дерева зависит от количества узлов `N` в этом дереве. Здесь можно отметить две крайности. Во-первых, всё дерево может выродиться в одну длинную ветвь, практически превратившись в список, тогда высота дерева просто равна числу узлов. Во-вторых, это может быть полное дерево, в этом случае на каждом новом слое число узлов удваивается, и высота дерева будет пропорциональна двоичному логарифму от числа узлов $O(\log_2(N))$.

Далее заметим, что для сбалансированных деревьев с числом узлов `N` наша функция будет всего вызываться не `N`, а `2N` раз. Так происходит потому, что из каждого листа мы делаем по два «лишних» обращения в пустые поддеревья, а для сбалансированных деревьев число листьев примерно равно числу остальных узлов! Можно так исправить эту ситуацию (это ещё одна оптимизация алгоритма):

```
function NumNodes(x: BT): integer;
  var NL,NR: integer;
begin
  NumNodes:=0 { пустое дерево }
  if x<>nil then begin
    NL:=0; NR:=0;
    if x↑.L<>nil then NL:=NumNodes(x↑.L);
    if x↑.R<>nil then NR:=NumNodes(x↑.R);
    NumNodes:=1+NL+NR
  end
end;
```

Немного усложнив алгоритм, мы вдвое уменьшили число вызовов функции. В сноске в конце раздела рассказано, как на языке Free Pascal замерить скорость работы программы, а также получить текущие дату и время. [ii] [см. сноску в конце главы]

Попытаемся теперь написать предусловие нашей функции, когда она даст правильный ответ? В 32-битном режиме работы ЭВМ под хранения данных программе отводится немного менее 2 млрд. байт, а под каждый узел процедура `new` берёт 16 байт (у нас длина узла 10 байт, но меньше 16 процедура `new` не выдаёт 🙁). Итого можно создать дерево из примерно 125 млн. узлов, следовательно, с хранением результата функции справиться тип `longint` (до 2 млрд.).

С глубиной рекурсии дело обстоит хуже. Как уже упоминалось, при каждом вызове подпрограммы порождается несколько служебных переменных, так что потребляется около 50 байт памяти, отведённой под хранение локальных переменных. Всего такой памяти совсем немного, около 1 млн. байт, так что глубина рекурсии не может превышать примерно 20000. Сбалансированное дерево из 100 млн. узлов имеет высоту $\log_2 10^7 \approx 27$, но *не сбалансированное* может иметь высоту вплоть до 10^7, так что этого надо опасаться. Итак, предусловие нашей функции: `Pred={H<20000}`.

Напишем теперь функцию подсчёта числа улов в двоичном дереве без рекурсии, только с помощью цикла. К сожалению, теперь нам будет нужно более «интеллектуальное» дерево. В каждый узел

придётся добавить ссылку вверх (на родительский узел) и служебную переменную k (стандартно во всех узлах дерева k=0):

```
type Link=↑Node;
  Node=record
    n: integer; k: integer { 0,1,2-нахождение в узле }
    L,R,U: Link;
  end;
  BT=Link;
```

```
function NumNodes(x: BT): integer;
  var S: integer;
begin
  if x=nil then NumNodes:=-1 { Дерево пусто } else begin
    S:=0;
    repeat
      case x↑.k of
        0: begin S:=S+1; x↑.k:=x↑.k+1;
             if x↑.L<>nil then x:=x↑.L
           end;
        1: begin x↑.k:=x↑.k+1;
             if x↑.R<>nil then x:=x↑.R
           end;
        2: begin x↑.k:=0; x:=x↑.U end;
      end
    until x=nil;
    NumNodes:=S
  end
end;
```

Самостоятельно разберитесь, как работает эта функция. Отметим, что в её работе использован дополнительный объём памяти (переменная k в каждом узле), пропорциональный числу узлов. Можно модифицировать этот алгоритм, использовав объём (глобальной) дополнительной памяти, пропорциональной высоте дерева. Отметим, что фактически мы промоделировали в цикле схему работы рекурсивного алгоритма.

Теперь напишем функцию, вычисляющую высоту дерева:

```
function H(x: BT): integer;
  var HL,HR: integer;
begin
  H:=0; { пустое дерево }
  if x<>nil then begin
    HL:=0; HR:=0;
    if x↑.L<>nil then HL:=H(x↑.L);
    if x↑.R<>nil then HR:=H(x↑.R);
    if HL<HR then HL:=HR; { HL=max(HL,HR) }
    H:=1+HL
  end
end;
```

К сожалению, у этой функции тоже предусловие Pred={H<20000}, так что использовать её для *контроля* деревьев с «плохой» высотой не получиться 🐷. [iii] [см. сноску в конце главы]

> Напишите функцию, которая для «хороших» деревьев с высотой H<20000 возвращает их высоту, а для «плохих» возвращает −1.

Решим теперь задачу нахождения максимального числа, хранящегося в двоичном дереве. Ясно, что надо написать функцию, получающую двоичное дерево как параметр по значению (объясните, почему?). Сделаем спецификацию для пустого дерева. Конечно, можно в этом случае возвращать ноль, однако, такой ответ будет и для непустого дерева, содержащего, например, только не положительные числа.

Так как любое целое число может быть максимумом, то возникает проблема, что наша операция должна возвратить в «специальном» случае пустого дерева. Многие учащиеся в этом случае говорят: «Давайте возвратим ноль, но напечатаем "ПУСТОЕ ДЕРЕВО"». Чтобы понять абсурдность такого решения, достаточно спросить, а кто будет это читать? Понятно, что нашу функцию может вызывать другая подпрограмма, которая «неграмотная» (читать из `output` не умеет ☺), однако на основе результата функции эта подпрограмма должна принять решение о дальнейшем ходе вычислений.

Для программы прочитать данные, выведенные в собственный поток `output` в большинстве случаев (например, при записи на экран, на принтер, да и в файл) очень трудно (обычно необходимо программировать на Ассемблере). На старых ЭВМ можно было читать символы, выведенные на экран, из так называемого видеобуфера, но сейчас это так просто не работает. Специфическое решение этой задачи в виде создания особого «трубопровода» (pipeline) между так называемыми параллельными процессами одной программы Вы узнаете из курса по операционным системам.

Так как простого решения этой проблемы нет, будем для пустого дерева, например, возвращать самое маленькое число типа `integer` (для большинства компьютеров это `-Maxint-1`):

```
function MaxNode(x: BT): integer;
  const Minint=-Maxint-1;
  var Max,M: integer;
begin Max:=Minint;
  if x<>nil then begin { не пустое дерево }
    M:=MaxNode(x↑.L);    { max в левом поддереве }
    if M>Max then Max:=M;
    M:=MaxNode(x↑.R);    { max в правом поддереве }
    if M>Max then Max:=M;
    M:=x↑.n;             { число в корне }
    if M>Max then Max:=M
  end;
  MaxNode:=Max
end;
```

Итак, функция возвращает максимальное из трёх значений: числа в корне `x↑.n` и максимумов из левого и правого поддеревьев.

Задания для самостоятельной работы.

1). Исправьте функцию для случая, когда для пустого дерева она должна возвращать ноль. Учтите, что простой замены присваивания `Max:=-Maxint-1` на `Max:=0` недостаточно.

2). Наша функция «зря» ходит в пустые поддеревья, вдвое увеличивая свою вычислительную сложность, исправьте это.

3). Решите эту задачу, написав другую функцию с заголовком

```
function MaxNode(x: BT): Link;
```

Пусть эта функция возвращает ссылку на узел, содержащий максимальное число дерева, или **nil** для пустого дерева (так мы сигнализируем, что максимального числа в дереве НЕТ).

13.3.1. Уничтожение и копирование деревьев

Уничтожение одного есть рождение другого.

Аристотель, IV век до н.э.

Когда надобность в построенных динамических структурах данных пропадает, их нужно уничтожать, чтобы освободить память для порождение других переменных. Уничтожение двоичного дерева типично рекурсивный процесс: сначала надо уничтожить левое поддерево, потом правое, а потом и соединяющий их корень. Ясно, что для этого необходима процедура с одним параметром, переданным обязательно по ссылке.

```
procedure DestroyBT(var x: BT);
  var p,q: Link;
begin
```

```
if x<>nil then begin { дерево не пусто }
  DestroyBT(x↑.L); DestroyBT(x↑.R);
  dispose(x); x:=nil { не забыть! }
end
end;
```

У этой операции такие же ограничения на высоту переданного ей дерева, как и у операции подсчёта числа узлов дерева.

Объясните, что будет делать эта процедура, если ей передавать параметр по значению.

Рассмотрим теперь операцию копирования. Для совместимых по присваиванию переменных копирование значения производит оператор присваивания, например:

```
var c,d: array[1..10] of real;
    a,b: integer; x,y: BT;
begin a:=b; c:=d; x:=y;
```

Здесь, конечно, важно понимать, что оператор `x:=y` копирует не сами двоичные деревья, а только ссылки на них. При рассмотрении очередей мы образно сравнивали такой оператор присваивания со случаем, когда одну очередь стали обслуживать сразу два продавца. А вот «настоящее» копирование двоичных деревьев приходится реализовывать в виде отдельной операции.

Первое, что приходит в голову, это реализовать такое копирование с помощью процедуры, например, с таким заголовком:

```
procedure CopyBT(x: BT; var y: BT);
{ Не забыть сначала DestroyBT(y) }
```

Однако, когда начинаешь делать спецификацию такой операции, возникает трудность. Ясно, что для `x=nil` надо присвоить `y:=nil`, но что делать, если было `y<>nil` ? Придётся, чтобы не было утечки памяти, сначала уничтожить y, но кто должен это делать: сама процедура или тот, кто её вызвал? Из этих соображений можно реализовать операцию копирования в виде функции:

```
function CopyBT(x: BT): BT;
{ Надо сначала DestroyBT(y), потом y:=CopyBT(x) }
  var p: BT;
begin
  CopyBT:=x;              { для пустого дерева }
  if x<>nil then begin
    new(p); p↑.n:=x↑.n;  { копия корня }
    p↑.L:=CopyBT(x↑.L); { копия левого поддерева }
    p↑.R:=CopyBT(x↑.r); { копия правого поддерева }
    CopyBT:=p
  end
end;
```

Как видим, для непустого дерева наша функция сначала копирует корень, а потом вызывает сама себя, чтобы скопировать левое и правое поддеревья, и «подвесить» их к скопированному корню.

Задание для самостоятельной работы. К сожалению, наша операция тратит много времени на копирования пустых поддеревьев, постарайтесь это исправить.

В качестве последней рассмотрим задачу нахождения суммы чисел, расположенных в *листьях троичного* дерева целых чисел. Это дерево можно описать так:

```
type Link=↑Node; { Link это стрелка }
     Node=record L,M,R: Link; n: integer end;
     TT=Link; { Ternary Tree – всё троичное дерево }
```

Для упрощения задачи сделаем спецификацию, когда и пустое дерево, и дерево с нулевой суммой чисел в листьях, будут возвращать ответ ноль. В этом случае получится такая функция:

```
function SumLeafs(x: TT): integer;
  var SL,SM,SR: integer;
```

```
begin
  SumLeafs:=0;                    { для пустого дерева }
  if x<>nil then begin
  ❶ if (x↑.L=nil) and (x↑.M=nil) and (x↑.R=nil)
{ ❶ if (x↑.L=x↑.M) and (x↑.M=x↑.R)
    так тоже лист, но менее понятно 🙂 }
    then SumLeafs:=x↑.n;  { лист ! }
    else begin
  ❷   SumLeafs:=SumLeafs(x↑.L)+
                SumLeafs(x↑.M)+SumLeafs(x↑.R)
{ ❷ то же самое, более длинно, но и более эффективно
      SL:=0; SM:=0; SR:=0;
      if x↑.L<>nil then SL:=SumLeafs(x↑.L);
      if x↑.M<>nil then SM:=SumLeafs(x↑.M);
      if x↑.R<>nil then SR:=SumLeafs(x↑.R);
      SumLeafs:=SL+SM+SR
}
    end
  end
end;
```

Значком ❶ помечены два способа проверки того, что узел является листом, первый очевидный (все три потомка равны **nil**), второй «хитрый», так как только ссылки **nil** будут равны между собой. Аналогично значком ❷ помечены два способа рекурсивного обхода (без проверки и с проверкой пустого поддерева). Обязательно разберитесь, как всё работает.

Ну, а далее из деревьев можно выращивать леса́ 🙂. Вот, например, как определяется абстрактная структура данных случайный лес (random forest), она широко используется в машинном обучении (machine learning). «Случайный лес – это множество решающих деревьев. В задаче регрессии их ответы усредняются, а в задаче классификации принимается решение голосованием по большинству. Все деревья строятся независимо друг от друга» 🙄.

Вопросы и упражнения

Мы тогда постигаем, когда не можем, но делаем. Ибо не можем часто относительно, нам лишь кажется, что не можем.

Н.К. Рерих

1. Какие структуры данных называются динамическими?
2. В чём трудность решения задач, которые вводят данные переменного и неизвестного заранее размера?
3. Как описывается абстрактный тип данных «очередь»?
4. Что такое дисциплина обслуживания и как она определяется для очереди?
5. Может ли очередь увеличиваться сразу на несколько элементов?
6. Какая информация об очереди является избыточной и может быть удалена?
7. Какая очередь называется кольцевой?
8. Какая дисциплина обслуживания у абстрактной структуры данных стек?
9. Чем список отличается от очереди?
10. Чем абстрактная структура данных дерево принципиально отличается от списка?
11. Можно ли обойти все узлы дерева в цикле?
12. Какие связи в дереве можно считать избыточными?
13. Что такое глубина рекурсии?

[i] Для продвинутых читателей. Математически списки являются конечной последовательностью некоторых элементов, поэтому можно дать списку строгое (математическое) определение, мы этого делать не будем.

В программировании списки входят в более общий класс структур данных под названием «коллекции» (collection). Коллекцией называется абстрактная структура данных, содержащая в себе некоторое количество объектов (заметим, что здесь не требуется их упорядоченности). Из коллекции можно получить содержащийся там объект (сам объект при этом остаётся в коллекции), изъять или, наоборот, поместить объект в коллекцию. Всё остальное надо доопределить:

1) Одного ли типа объекты в коллекции?
2) При получении объекта из коллекции выдаётся ссылка на этот объект или его копия?
3) При помещении объекта в коллекцию, если он там уже есть, то туда добавляется второй экземпляр, или замещается существующий объект?
4) Упорядочены ли объекты в коллекции?
5) Может ли коллекция менять свой размер?

и т.д.

Ясно, что под это определение подходит и массив, и вектор, и строка, и множесто, и очередь и далее по списку 🙂. Несмотря на такую общность в определении, есть языки программирования (X#, Java и другие), которые включают в себя такую структуру данных. Разумеется, при описании коллекции она должна как-то конкретизироваться.

[ii] Для продвинутых читателей. Можно достаточно точно измерить время, потраченное компьютером на выполнение некоторого участка программы. Для этого надо получить количество тактов работы, потраченных процессором на выполнение данного участка. Вы уже, вероятно, знаете о тактовой частоте, с которой работает процессор, например, 3.2 ГГц означает, что 3.2 миллиарда раз в секунду на схемы процессора приходят электрические импульсы, заставляя его совершать некоторые элементарные операции.

Счётчик таких тактов для современных процессоров фирмы Intel хранится в специальном регистре длиной 64 бита, значение этого регистра можно считать в переменную типа `int64` с помощью стандартной процедуры языка Free Pascal

```
QueryPerformanceCounter(Tiks)
```

Вызвав эту процедуру в начале и в конце участка программы, скорость работы которого мы хотим замерить, можно получить число импульсов, затраченных процессором на выполнение этого участка программы. Далее, получив с помощью процедуры

```
QueryPerformanceFrequency(Frec)
```

тактовую частоту процессора в килогерцах, можно вычислить число секунд, потраченное на выполнение участка программы. Перечисленные процедуры находятся в модуле `Windows`.

Рассмотрим пример работы с матрицей:

```
uses Windows;
const N=10000;
var i,j: integer; Fr,Tic1,Tic2: int64;
    Mat: array[1..N,1..N] of real;
begin
  QueryPerformanceFrequency(Fr); { частота ЦП }
  Write('Тактовая частота=',Fr/1e+9:4:2,'ГГц. ');
{ ввод матрицы }
  QueryPerformanceCounter(Tic1);
  for i:=1 to N do
    for j:=1 to N do
        Mat[i,j]:=sin(i+j);
  QueryPerformanceCounter(Tic2);
  Writeln('Время=',(Tic2-Tic1)/Fr:5:3,' сек')
end.
```

На моём компьютере получилось

```
Частота=3.33 Ггц. Время=4.331 сек
```

Интересно, что, если поменять `Mat[i,j]:=sin(i+j)` на оператор `Mat[j,i]:=sin(i+j)`, (т.е. обходить матрицу не по строкам, а по столбцам), то будет такое время

```
Время=6.264 сек
```

Видно, что чтение памяти «подряд» много эффективнее, чем с шагом `N*sizeof(real)`. Можно попытаться «оптимизировать» программу, проходя индексами только то верхнему треугольнику матрицы, например, используя такой цикл

```
for i:=1 to N do begin
  Mat[i,i]:=sin(i+i);
  for j:=1 to N-1 do begin r:=sin(i+j);
    Mat[i,j]:=r; Mat[j,i]:=r
  end
end
```

Тогда получается такой результат

```
Время=3.980 сек.
```

Причина такого странного поведения процессора будет изучаться в курсе по архитектурам ЭВМ. Отметим, что, когда особая точность не нужна, можно использовать функцию без параметров `GetTickCount64`, которая выдаёт в формате `qword` число миллисекунд, «натикавшее» с момента включения компьютера.

Для работы с датой и временем в модуле `Sysutils` есть полезные функции без параметров `Time`, `Date` и `Now` (`Date+Time`). Они выдают время и дату в особом «временно́м» формате `TDateTime`, кто не желает вникать в этот формат, можно просто воспользоваться функцией преобразования его в строку символов, например:

```
uses Sysutils;
begin
  writeln('Дата=',DateToStr(Date));
  writeln('Время=',TimeToStr(Time));
  writeln('Дата и
время=',DateTimeToStr(Date))
end.
```

[iii] Для продвинутых читателей. Итак, при вызове процедур и функций мы берём память из стека (автоматический класс памяти), а при выходе из процедур и функций мы возвращаем эту память обратно в стек. При порождении динамической переменной (например, узла дерева, это динамический класс памяти), мы берём память из кучи по `new`, а при уничтожении переменной по `dispose` мы возвращаем эту память назад в кучу.

Проблема невозможности сделать большую глубину рекурсии кроется в том, что стек относительно маленький (в Windows обычно 1 Мб), а куча большая (в 32-битном режиме до 2 Гб). Таким образом, единственное решение заключается в ручном моделировании работы со стеком, причём стек переносится в кучу! Впрочем, иногда бывает выгоднее использовать не стек, а очередь. При этом рекурсивный алгоритм преобразуется в алгоритм с циклами. Такие алгоритмы разработаны для многих задач обработки деревьев и графов. Например, сам Н. Вирт разработал такой не рекурсивный алгоритм поиска пути в графе.

С другой стороны, модный и сложный язык Haskell совсем не содержит циклов, а только рекурсию 😕.