КЕМЕРОВСКИЙ ИНСТИТУТ (филиал)
РОССИЙСКИЙ ГОСУДАРСТВЕННЫЙ ТОРГОВО-ЭКОНОМИЧЕСКИЙ УНИВЕРСИТЕТ
КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Информатика и программирование

Лебедева Т.Ф.

V Разработка документации

Основной результат этого этапа – также создание эксплуатационной документации на программный продукт:

- *описание применения* – дает общую характеристику программного изделия с указанием сферы его применения, требований к базовому программному обеспечению, комплексу технических средств;
- *руководство пользователя* - включает детальное описание функциональных возможностей и технологии работы с программным продуктом. Данный вид документации ориентирован на конечного пользователя и содержит необходимую информацию для самостоятельного освоения и нормальной работы пользователя;
- *руководство программиста* – указывает особенности установки программного продукта и его внутренней структуры – состав и назначение модулей, правила эксплуатации и обеспечения качественной работы программного продукта

7 Работа с динамическими структурами данных 112

В предыдущих лекциях мы рассматривали программирование, связанное с обработкой только статических данных. Статическими величинами называются такие, память под которые выделяется во время компиляции и сохраняется в течение всей работы программы.

Операционная система MS - DOS все адресуемое пространство делит на сегменты.

Сегмент - это пронумерованный участок памяти размером 64 К байт.

Причем сегмент может начинаться с любого физического адреса оперативной памяти. Для задания адреса объекта программы необходимо определить адрес начала сегмента и смещение относительно начала сегмента, т.е. номер байта от начала сегмента. Адрес хранится как два слова, одно из них определяет сегмент, второе – смещение: *сегмент : смещение*

Например 5E87:0058

Схема распределения памяти программ на Turbo Pascal

	Свободная память	HeapEnd
	.....	
	Куча (растет вверх)	HeapOrg
	Оверлейный буфер	
	Стек (растет вниз)	
	.....	
	Свободный стек	
Образ EXE - файла	Типизированные константы	
	.....	
	Глобальные переменные	
	Кодовый сегмент модуля System	
	Кодовый сегмент последнего модуля	
	Кодовый сегмент главной программы	
	Префикс сегмента программы (PSP)	PrefixSeg

Префикс сегмента программы (Program Segment Prefix - PSP) - это 256-ти байтовая область, создаваемая DOS при загрузке программы.

Адрес сегмента PSP хранится в переменной PrefixSeg.

Каждый модуль (и главная программа) имеет свой кодовый сегмент. Главная программа занимает первый кодовый сегмент; кодовые сегменты, которые следуют за ним, занимают модули (в порядке, обратном тому, как они следовали в операторе uses), и последний кодовый сегмент занимает библиотека времени выполнения (модуль System).

Сегмент данных содержит все глобальные переменные и затем все типизированные константы.

Размер сегмента данных не может превышать 64К.

Размер стекового сегмента не может превышать 64К; размер по умолчанию - 16К, он может быть изменен директивой компилятора $M.

Буфер оверлеев используется стандартным модулем Overlay для хранения оверлейного кода. Размер оверлейного буфера по умолчанию соответствует размеру наибольшего оверлея в программе; если в программе нет оверлеев, размер буфера оверлеев равен 0.

Раздел оперативной памяти, распределяемый статически, называется статической памятью; динамически распределяемый раздел памяти называется динамической памятью
(динамически распределяемой памятью или *кучей*).
Куча хранит динамические переменные. Куча занимает всю или часть свободной оперативной памяти, оставшейся после загрузки программы. Фактически размер кучи зависит от минимального и максимального значений кучи, которые могут быть установлены директивой компилятора **$M:**
{$M размер стека, мин.размер кучи, максим. размер кучи}
по умолчанию предполагаемая директива
{$M 16384, 0, 655360}
Использование динамических величин предоставляет программисту ряд дополнительных возможностей:

- позволяет увеличить объем обрабатываемых данных;
- если потребность в каких-то данных отпала до окончания программы, то занятую ими память можно освободить для другой информации;
- позволяет создавать динамические структуры данных.
- позволяет создавать структуры данных переменного размера

7 Работа с динамическими структурами данных 116

Объект данных обладает динамической структурой, если его размер изменяется в процессе выполнения программы или он потенциально бесконечен. Данные статической структуры – это данные, взаиморасположение и взаимосвязи элементов которых всегда остаются постоянными.

Для работы с динамическими переменными в программе должны быть выполнены следующие действия:

- Выделение памяти под динамическую переменную;
- Инициализация указателя;
- Освобождение памяти после использования динамической переменной.

Программист должен сам резервировать место, определять значение указателей, освобождать динамическую память (ДП).

Вместо любой статической переменной можно использовать динамическую, но без реальной необходимости этого делать не стоит.

7 Работа с динамическими структурами данных 117

Указатели

Для работы с динамическими программными объектами в Паскале предусмотрен ссылочный тип или тип указателей. В переменной ссылочного типа хранится ссылка на программный объект - адрес объекта.

Указатель – это переменная, которая в качестве своего значения содержит адрес байта памяти.

Объявление указателей

Указатель, связанный с некоторым определенным типом данных, называют типизированным указателем. Его описание имеет вид:

<Имя_переменной>: ^ <базовый-тип>;

Например:

Пример фрагмента программы объявления указателя

```
Type   A = array [1..100] of  integer;
   TA= ^ A ; {тип указатель на массив}
Var
   P1: ^ integer; {переменная типа указатель на целое число}
   P2: ^ real; {переменная типа указатель на вещественное число}
```

Указатель, не связанный с каким-либо конкретным типом данных, называется нетипизированным указателем. Для описания нетипизированного указателя в Паскале существует стандартный тип *pointer*. Описание такого указателя имеет вид:

<Имя-переменной>: pointer;

С помощью нетипизированных указателей удобно динамически размещать данные, структура и тип которых меняются в ходе выполнения программы.

Значения указателей – адреса переменных в памяти.

Адрес занимает четыре байта и хранится в виде двух слов, одно из которых определяет сегмент, второе – смещение.

Можно передавать значения только между указателями, связанными с одним типом данных. Указатели на различные типы данных имеют различный тип, причем эти типы несовместимы.

Пример фрагмента программы объявления указателя различных типов

```
Var  p1, p2: ^integer;
     p3: ^real;
     pp: pointer;
     ……
     p1:= p2; {допустимое действие }
     p1:= p3; {недопустимое действие}
```

Однако это ограничение не распространяется на нетипизированный указатель.

В программе допустимы будут следующие действия:

```
pp:=  p3;
p1:= pp;
```

Выделение и освобождение динамической памяти

Вся ДП рассматривается как сплошной массив байтов, который называется кучей. Существуют стандартные переменные, в которых хранятся значения адресов начала, конца и текущей границы кучи:

Heaporg – начало кучи; ***Heapend*** – конец кучи;

Heapptr – текущая граница незанятой ДП.

Выделение памяти под динамическую переменную осуществляется процедурой:

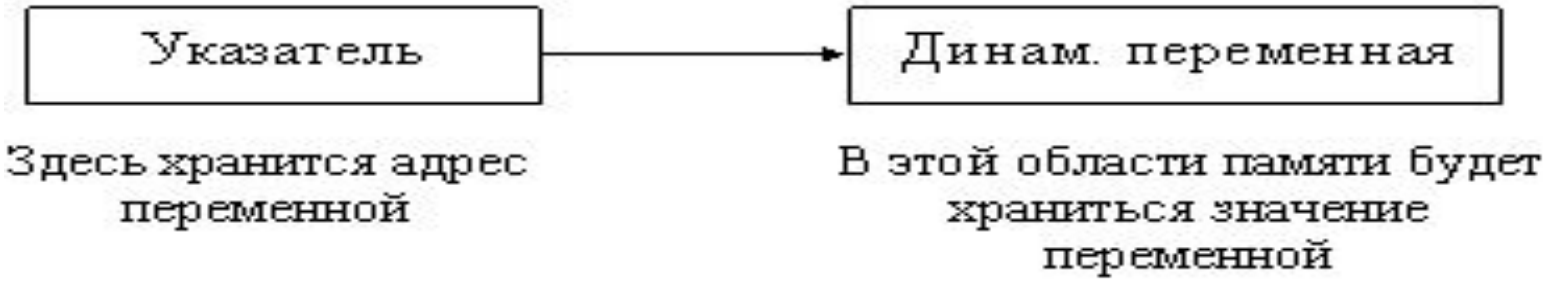
New (<переменная_типа_указатель>)

В результате обращения к этой процедуре указатель получает значение, соответствующее адресу в динамической памяти, начиная с которого можно разместить данные.

Пример фрагмента программы объявления указателя различных типов

```
Var  i,  j: ^integer;
     r:  ^real;
   begin
     new( i); {после этого указатель i   приобретает значение адреса Heapptr, а
   Heapptr смещается на 2 байта}
     ……..
     new( r) ; { r приобретает значение Heapptr, а Heapptr смещается на 6 байт}
```

Графически действие процедуры new можно изобразить так:



Освобождение динамической памяти осуществляется процедурой:

Dispose (переменная_типа_указатель)

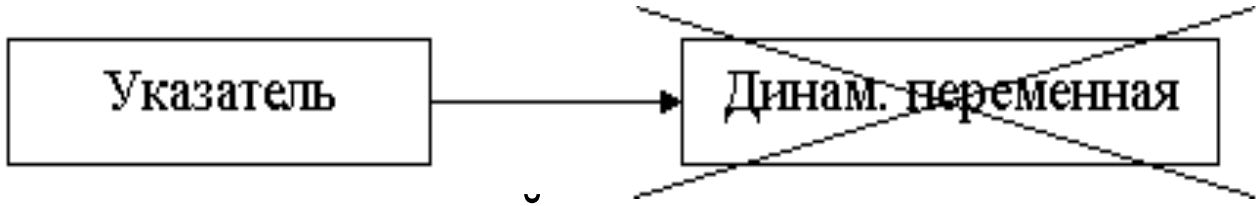
Пример фрагмента программы

Dispose (i); {возвращает в кучу 2 байта}

Dispose (r); {возвращает в кучу 6 байт}

Следует помнить, что повторное применение процедуры dispose к свободному указателю может привести к ошибке.

Процедура dispose освобождает память, занятую динамической переменной. При этом значение указателя становится неопределенным.



Присваивание значений указателю

Для инициализации указателей существует несколько возможностей.

- процедура ***new*** отводит блок памяти в области динамических переменных и сохраняет адрес этой области в указателе;
- специальная операция **@** ориентирует переменную-указатель на область памяти, содержащую уже существующую переменную. Эту операцию можно применять для ориентации на статическую и динамическую переменную.

Например,

type A = array[0..99] of char;
var X: array [0..49] of integer;
pA:^ A; {указатель на массив символов}

Объявлены переменные разных типов: массив из 50 целых чисел и указатель на массив символов. Чтобы указатель pA указывал на массив X, надо присвоить ему адрес ***X*** ***pA:= @ X;***

Существует единственная константа ссылочного типа *nil*, которая обозначает «пустой» адрес. Ее можно присваивать любому указателю.

Переменной-указателю можно присвоить значение другого указателя того же типа. Используя указательный тип pointer как промежуточный, можно присвоить значение одного указателя другому при несовпадении типов.

Операции с указателями

Для указателей определены только операции присваивания и проверки на равенство и неравенство. В Паскале запрещаются любые арифметические операции с указателями, их ввод-вывод и сравнение на больше-меньше.

Еще раз повторим правила присваивания указателей:

1. любому указателю можно присвоить стандартную константу *nil,* которая означает, что указатель не ссылается на какую-либо конкретную ячейку памяти;
2. указатели стандартного типа pointer совместимы с указателями любого типа;
3. указателю на конкретный тип данных можно присвоить только значение указателя того же или стандартного типа данных.
4. Указатели можно сравнивать на равенство и неравенство, например:
5. ***If p1=p2 then …..***

   If p1<>p2 then *…..*

В Паскале определены стандартные функции для работы с указателями:

addr(x) – тип результата pointer, возвращает адрес x (аналогично операции @), где x – имя переменной или подпрограммы);

seg(x) – тип результата word, возвращает адрес сегмента для x;

ofs(x) – тип результата word, возвращает смещение для x;

ptr(seg, ofs) – тип результата pointer, по заданному сегменту и смещению формирует адрес типа pointer.

Присваивание значений динамическим переменным

После того, как динамическая переменная объявлена, ей можно присваивать значения, изменять их, использовать в выражениях и т.д.

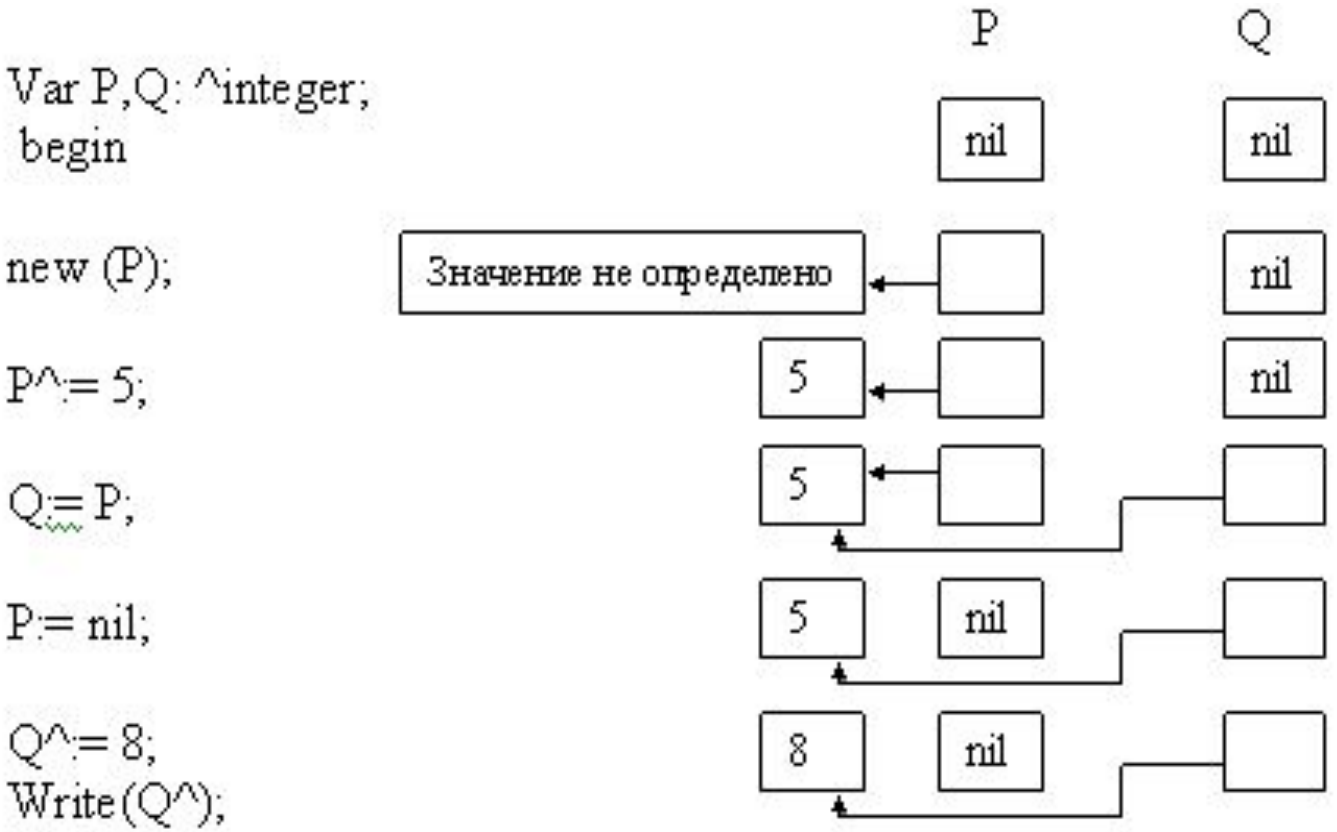
Для этого используют следующее обращение: <*переменная_указатель*>^.

Такое обращение называется операция *разадресации (разъименования).* Таким образом происходит обращение к значению, на которое указывает указатель, т.е. к данным. Если же за <переменной_указателем> значок ^ не стоит, то имеется в виду адрес, по которому расположены данные.

Динамически размещенные данные можно использовать в любом месте программы, где допустимо использование выражений соответствующего типа.

Например: *R^:= sqr(R^) + I^ -17; q^:= 2; inc(q^); writeln(q^) :*

Рассмотрим пример работы с указателями:



Серия последовательных обращений к New и Dispose обычно приводит к фрагментации памяти - память разбивается на небольшие фрагменты с чередованием свободных и занятых участков.

Дпя уменьшения явления фрагментации используют специальные процедуры.

Процедура ***Mark (Var p:pointer)*** запоминает значение HeapPtr в указателе р, полученном в качестве параметра.

Процедура ***Release (Var p:pointer)*** - освобождает весь фрагмент памяти, начиная с адреса р, зафиксированного в указателе процедуры Mark. Например:

new(pl); new(p2); mark(p);
new(p3); new(p4);
release(p);.,.

Динамическую память можно выделять фрагментами, указывая их размер.

Процедура ***GetMem (Var p:pointer; size:word)*** - запрашивает у системы память размера, указанного в параметре size (запрашиваемый объем не должен превышать 64КБ), и помещает адрес выделенного системой фрагмента в переменную типа pointer с именем р.

Функция ***SizeOf(x): word-*** возвращает длину указанного объекта х в байтах.

Процедура ***FreeMem (p:pointer; size:word)*** - освобождает область памяти, вьделенную процедурой GetMem.


Данные динамической структуры
Файлы
Текстовые
Типизированные
Нетипизированные
Несвязанные динам. структуры
Классифицируются аналогично данным статической структуры
Связанные динам. структуры
Линейной структуры
Односвязные
Очередь
Стек
Дек
Список
Многосвязные
Многосвязный линейный список
Кольцевой структуры
Односвязный кольцевой список
Многосвязный кольцевой список
Разветвленной структуры
Деревья
Графы
Двоичные (бинарные)
Разветвленные

Динамические структуры данных

Переменные типа «указатель» обычно используют при реализации динамических переменных, в том числе и динамических структур данных.

Динамические структуры данных могут быть организованы линейно и в виде дерева и в виде сети.

Линейная динамическая структура представляет собой изменяемую последовательность элементов. Частными случаями таких структур являются:

• *стеки*, в которых разрешено добавлять элементы только в конец и удалять только последние элементы (LIFO – last in first out);

• *очереди*, в которых добавление элементов осуществляется в конец, а удаление - из начала (FIFO – first in first out);

• *деки*, которые допускают добавление и удаление элементов и с начала, и с конца.

Списком называют структуру, в которой помимо данных хранятся также адреса элементов. Элемент списка состоит из двух частей: *информационной*, содержащей данные, и *адресной*, где хранятся указатели на следующие элементы.

В зависимости от количества полей в адресной части и порядка связывания элементов различают:

• *линейные односвязные списки* - единственное адресное поле содержит адрес следующего элемента, если следующий элемент отсутствует, то в адресное поле заносят константу nil ;

• *кольцевые односвязные списки* - единственное адресное поле содержит адрес следующего элемента, а последний элемент ссылается на первый;

• *линейные двусвязные списки* - каждый элемент содержит адреса предыдущего и последующих элементов, соответственно первый элемент в качестве адреса предыдущего, а последний - в качестве адреса следующего элемента содержат nil ;

• *кольцевые двусвязные списки* - каждый элемент содержит адреса предыдущего и последующих элементов, причем первый элемент в качестве предьщущего содержит адрес последнего элемента, а последний элемент в качестве следующего - адрес первого элемента

В древовидной структуре каждый элемент (вершина) ссылается на один или более элементов следующего уровня

Исходные установки

В начале программы работы с линейными динамическими структурами необходимо описать элемент и его тип:

```
Type tpel=^element; {тип «указатель на элемент»}
element= record
num:integer; {число}
p:tpel; {указатель на следующий элемент}
end;
```

В Паскале существует основное правило: перед использованием какого-либо объекта он должен быть описан.

Исключение сделано лишь для указателей, которые могут ссылаться на еще не объявленный тип.

В статической памяти описываем переменную-указатель списка и несколько переменных-указателей, используемых при выполнении операций со списком:

```
Var  first, {указатель списка - адрес первого элемента списка}
n, f, q:tpel; {вспомогательные указатели}
```

Исходное состояние «список пуст»:

```
first :=nil;
```

Пример1 Разработать программу, которая строит список по типу стека из целых чисел, вводимых с клавиатуры. Количество чисел не известно, но отлично от нуля. Конец ввода - по комбинации клавиш CTRL-Z (конец файла на устройстве ввода). Обычно построение списка по типу стека выполняется в два этапа: в список заносится первый элемент, а затем организуется цикл добавления элементов перед первым:

```
Program stek;
Type tpel=^element; {тип «указатель на элемент»}
Element = record
num : integer; {число}
p: tpel; {указатель на следующий элемент}
end;
Var   first, q, f:tpel;
a:integer;
```

```
begin
ReadLn(a);
new(first); {запрашиваем память под элемент}
first ^.num:=a; {заносим число в информационное поле}
first ^.p:=nil; {записываем nil в поле, «адрес следующего»}
while not EOF do
    begin
      ReadLn(a);
       new(q); {запрашиваем память под элемент}
      q^.num:=a; {заносим число в информационное поле}
      q^.p:=first; {в поле «адрес следующего» переписываем
адрес первого элемента}
         first:=q; {в указатель списка заносим адрес нового
элемента}
  end; ...
```

```
{удаление стека с выводом значений его элементов}
while   first <> nil do
   begin
          a:=q^.num;  {заносим информационное поле в переменную a}
      q:= first; {в поле «адрес следующего» переписываем адрес первого элемента}
      first:=q^.p;
      Dispose(q);
      Writeln('a=', a);
  end; ...
```

Пример 2 Разработать программу, которая строит список по типу очереди из целых чисел, вводимых с клавиатуры. Количество чисел не известно, но отлично от нуля. Конец ввода - по комбинации CTRL-Z (конец файла на устройстве ввода). При построении списка по типу очереди сначала мы заносим в стек первый элемент, а затем организуем цикл добавления элементов после последнего, приняв во внимание, что nil необходимо разместить в адресном поле только последнего элемента:

```
ReadLn(a);
new(first); {запрашиваем память под элемент}
first ^.num:=a; {заносим число в информационное поле}
f:=first; {f - текущий элемент, после которого добавляется  следующий}
while not EOF do
    begin
      ReadLn(a);
      new(q); {запрашиваем память под элемент}
      q^.num:-a; {заносим число в информационное поле}
      f^.p:=q; {в поле «адрес следующего» предыдущего элемента
заносим адрес нового элемента}
      f:= f^.p; {теперь новый элемент стал последним}
   end;
q^.p:=nil; (в поле «адрес следующего» последнего элемента записываем nil}
```

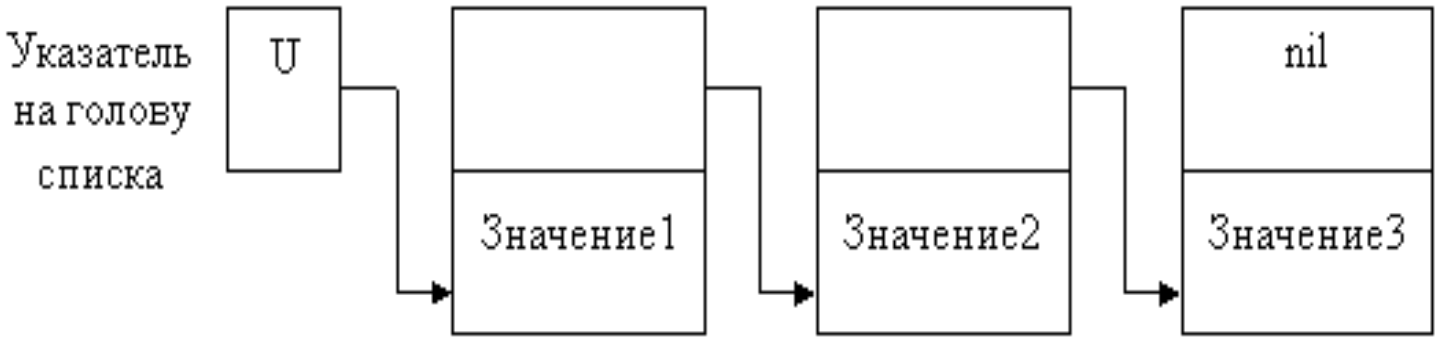
Динамические структуры

Линейные односвязные списки (однонаправленные цепочки).

Списком называется структура данных, каждый элемент которой посредством указателя связывается со следующим элементом.

Каждый элемент связанного списка, во-первых, хранит какую-либо информацию, во-вторых, указывает на следующий за ним элемент. Так как элемент списка хранит разнотипные части (хранимая информация и указатель), то его естественно представить записью, в которой в одном поле располагается объект, а в другом – указатель на следующую запись такого же типа. Такая запись называется звеном, а структура из таких записей называется списком или цепочкой.

Лишь на самый первый элемент списка (голову) имеется отдельный указатель. Последний элемент списка никуда не указывает.



Описание списка

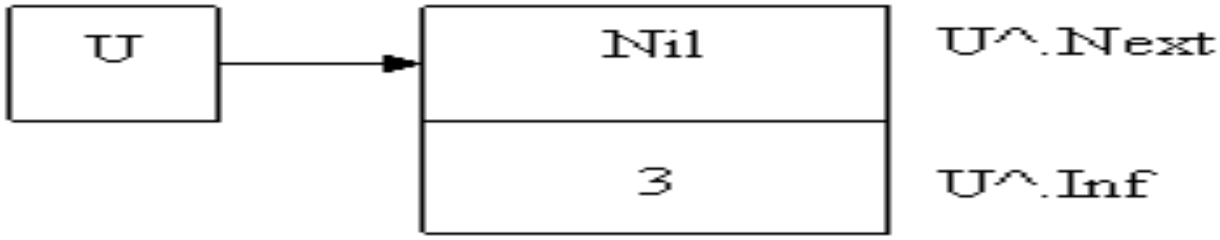
Пример описания списка

```
Type ukazat= ^ S;
  S= record
     Inf: integer;
     Next: ukazat;
     End;
Var  u: ukazat;
```

Формирование списка

Создадим первый элемент списка:

```
New (u); {выделяем место в памяти}
u^. Next:= nil; {указатель пуст}
u^. Inf:=3;
```



Продолжим формирование списка. Для этого нужно добавить элемент либо в конец списка, либо в голову.

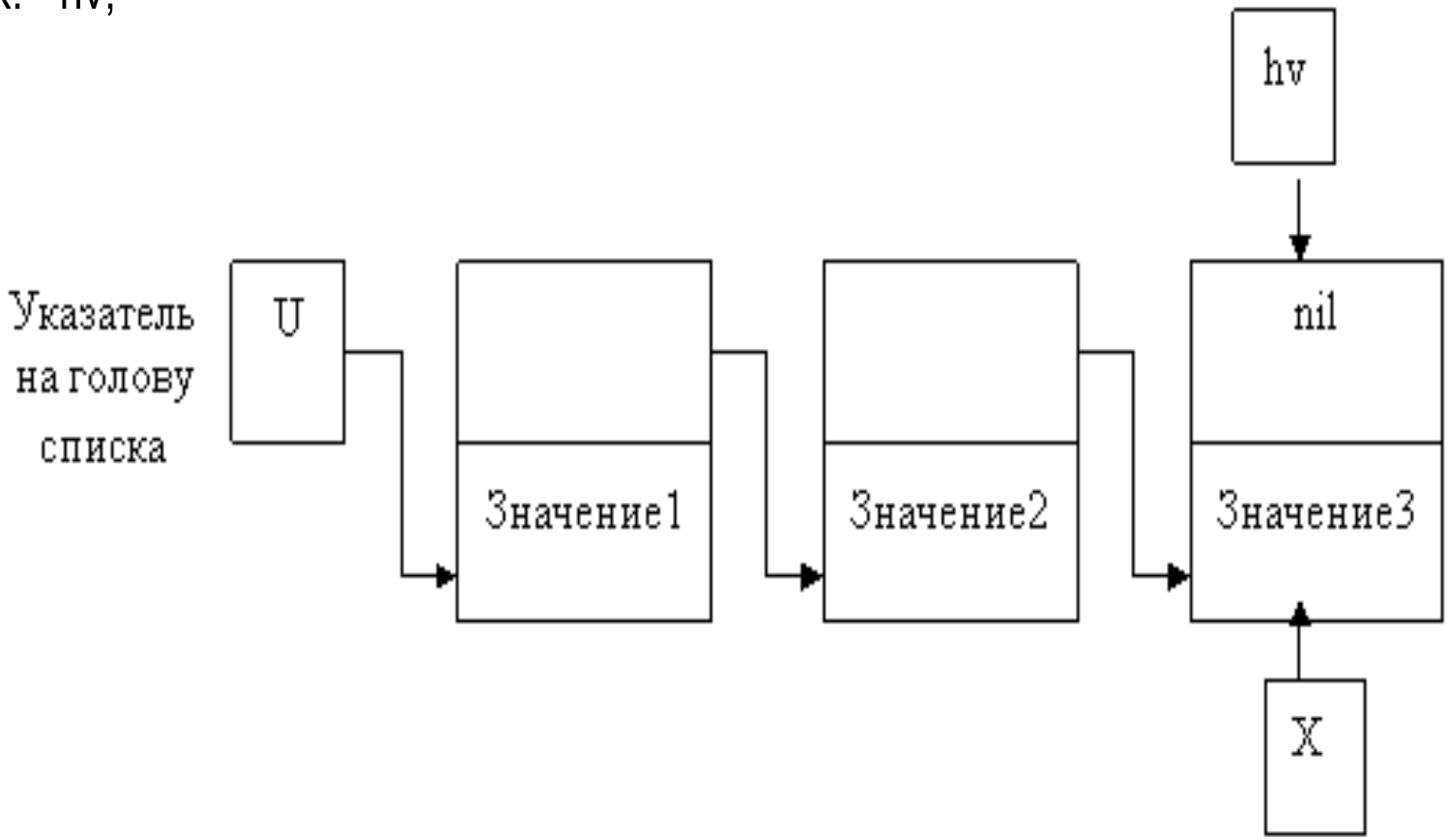
А) Добавим элемент в голову списка. Для этого необходимо выполнить последовательность действий:

- получить память для нового элемента;
- поместить туда информацию;
- присоединить элемент к голове списка.

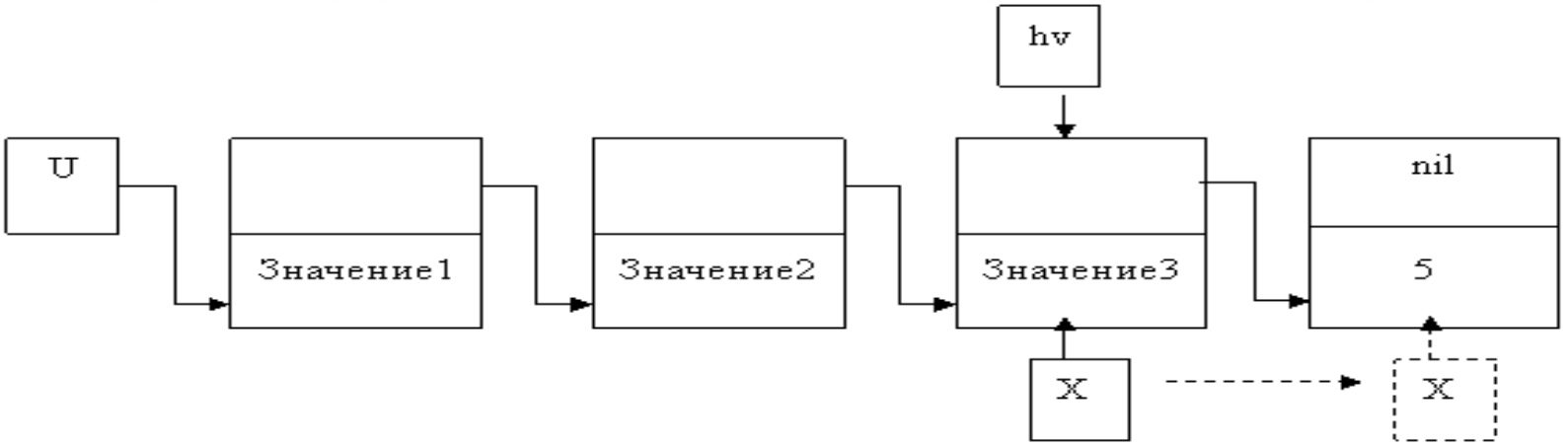
```
New(x);
Readln(x^.Inf);
x^. Next:= u;
u:= x;
```

Б) Добавление элемента в конец списка. Для этого введем вспомогательную переменную, которая будет хранить адрес последнего элемента. Пусть это будет указатель с именем hv (хвост).

x:= hv;



New(x^. next); {выделяем память для следующего элемента}



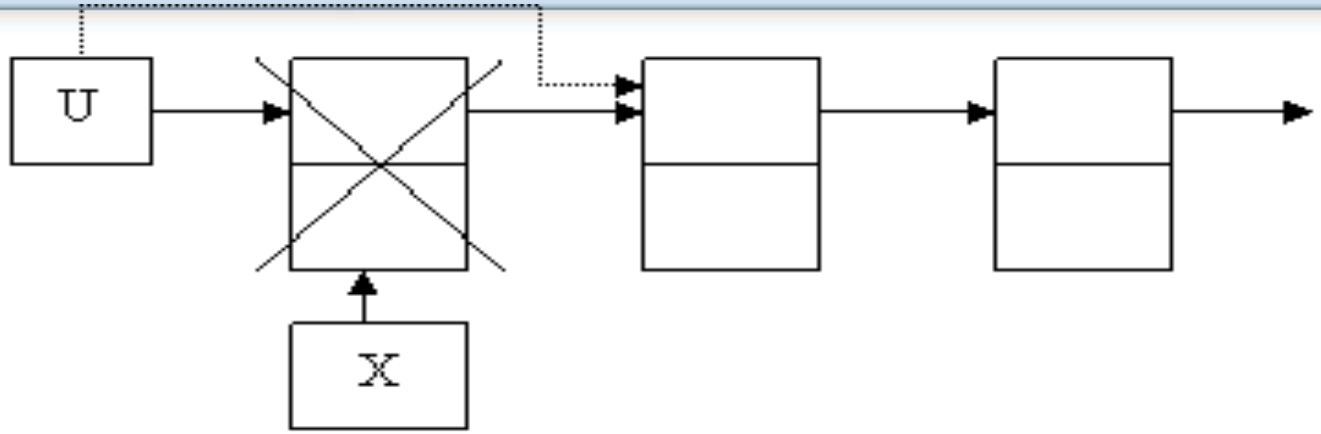
x:= x^.next;
x^.next:= nil;
x^. inf:= 5;
hv:=x;

Просмотр списка

```
While u<> nil do
Begin
WriteIn (u^.inf);
u:= u^.next;
end;
```

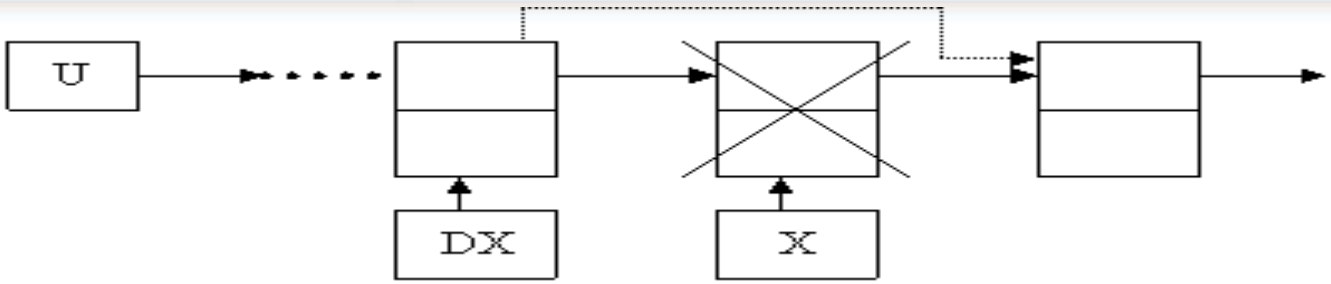
Удаление элемента из списка

А)Удаление первого элемента. Для этого во вспомогательном указателе запомним первый элемент, указатель на голову списка переключим на следующий элемент списка и освободим область динамической памяти, на которую указывает вспомогательный указатель.



x:= u;
u:= u^.next;
dispose(x);

Б) Удаление элемента из середины списка. Для этого нужно знать адреса удаляемого элемента и элемента, стоящего перед ним. Допустим, что digit – это значение удаляемого элемента.




```
x:= u;
while ( x<> nil) and ( x^. inf<> digit) do
begin
dx:= x;
x:= x^.next;
end;
dx:= x^.next:
dispose(x);
```

В)Удаление из конца списка. Для этого нужно найти предпоследний элемент.

```
x:= u; dx:= u;
while x^.next<> nil do
begin
dx:= x; x:= x^.next;
end;
dx^.next:= nil;
dispose(x);
```

Просмотр списка.

Важно уметь перебирать элементы списка, выполняя над ними какую-либо операцию. Пусть необходимо найти сумму элементов списка.

```
summa:= 0;
x:= u;
while x<> nil do
begin
summa:= summa+ x^.inf;
x:= x^.next;
end;
```

Выделим типовые операции над списками:

1. добавление элемента в начало списка;
2. удаление элемента из начала списка;
3. добавление элемента в произвольное место списка, отличное от начала (например, после элемента, указатель на которое задан);
4. удаление элемента из произвольного места списка, отличного от начала (например, после элемента, указатель на которое задан);
5. проверка, пуст ли список;
6. очистка списка;
7. печать списка.

Бинарные деревья

В математике бинарным (двоичным) деревом называют конечное множество вершин, которое либо пусто, либо состоит из корня и не более чем двух непересекающихся бинарных деревьев, называемых левым и правым поддеревьями данного корня.

Таким образом, каждая вершина бинарного дерева может включать одно или два поддерева или не включать поддеревьев вовсе. Первое поддерево обычно называют левым, а второе - правым. Соответственно ветвь, исходящую из вершины и ведущую в корень левого поддерева, называют левой, а ветвь, ведущую в корень правого поддерева - правой.

Вершины, из которых не выходит ни одной ветви, называют листьями.

В программах для реализации бинарных деревьев используют n-связные списки. С вершинами бинарного дерева обычно связывают записи, хранящие некоторую информацию.

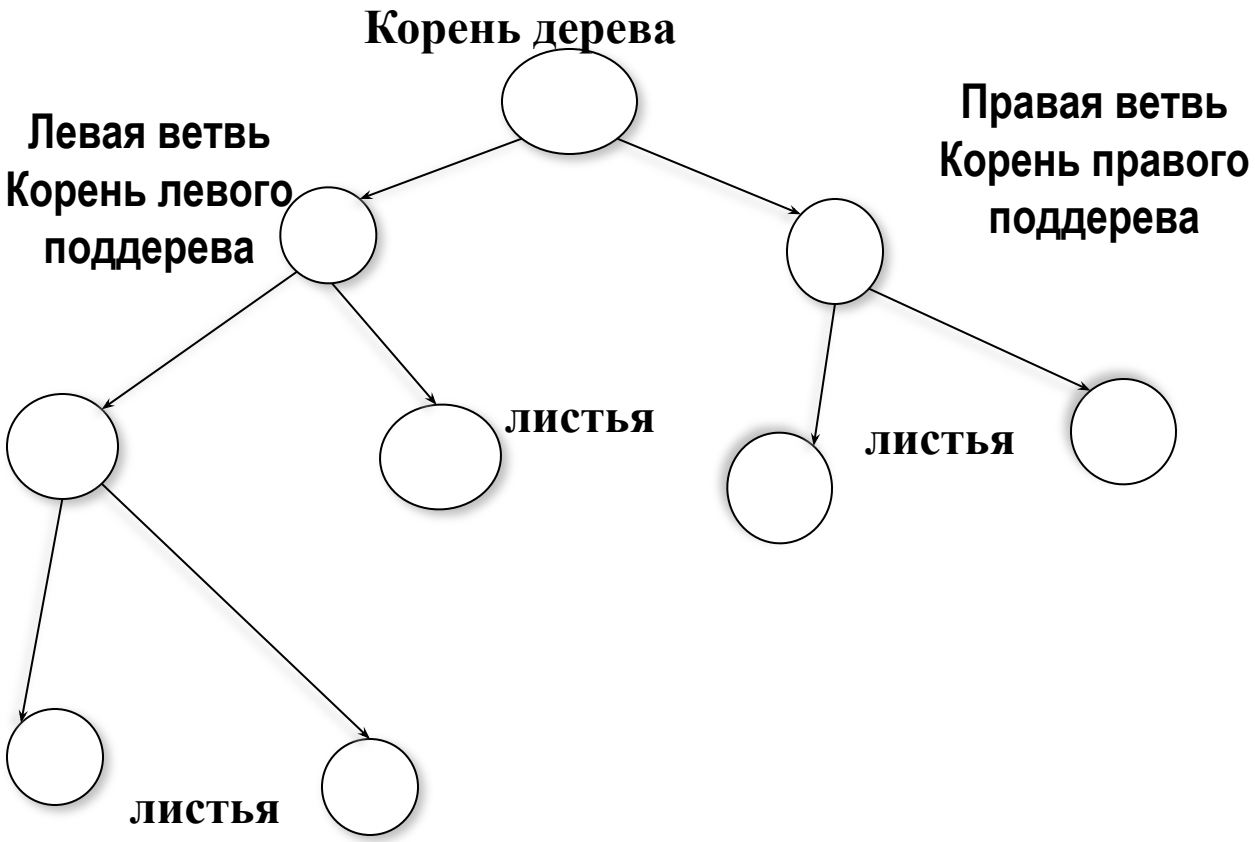


Рис. 1 Пример бинарного дерева

Построение дерева выполняется следующим образом.

Если дерево пусто, то первая же вершина становится корнем дерева. Добавление остальных вершин регламентируется в зависимости от условия задачи: в соответствии с заданной дисциплиной построения дерева отыскивается подходящая вершина, к которой и подсоединяется новая вершина.

Достаточно часто используют регулярные бинарные деревья с разными законами построения. Примером могут служить сортированные бинарные деревья, построение которых осуществляется по правилу:

- ключевое поле левого поддерева всегда должно содержать значение меньше, чем в корне,
- а ключевое поле правого поддерева - значение больше или равное значению в корне.

Рассмотрим основные операции с сортированными бинарными деревьями.

Исходные установки. В начале программы необходимо описать элемент и его тип:

```
Type   top_ptr = ^top; {тип «указатель на вершину»}
 top=record
     value:  integer; {число}
      left, {указатель на левое поддерево}
      right:top_ptr; {указатель на правое поддерево}
end;
```

Описываем указатель корня дерева и несколько указателей, используемых при выполнении операций с деревом:

```
Var    root, {указатель структуры - адрес корня дерева}
pass,  next, q : top_ptr; {вспомогательные указатели}
Исходное состояние - «пустое дерево»:
root:=nil;
```

1.Построение дерева.

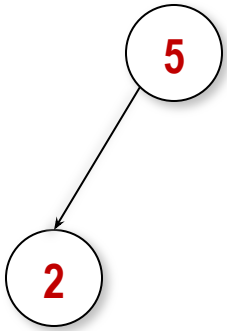
Дерево строится в соответствии с главным правилом.

Например, пусть дана последовательность целых чисел {5, 2, 8, 7, 2, 9, 1, 5}.

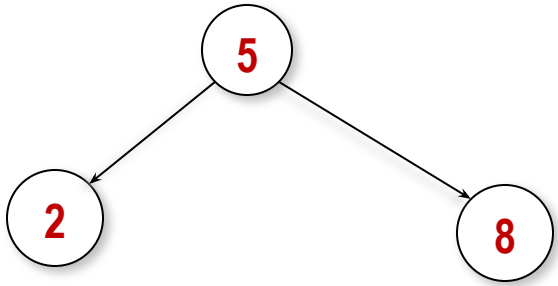
1. Первое число 5 будет записано в корень дерева (рис. 2, а).
2. Второе число 2 меньше значения в корне дерева, следовательно, оно будет записано в левое поддерево (рис. 2, б).
3. Следующее число 8 больше значения в корне, соответственно оно будет записано в правое поддерево (рис. 2, в).
4. Следующее число 7 больше, чем значение в корне дерева, значит, оно должно быть записано в правое поддерево, но правое поддерево уже построено.
5. Сравниваем 7 со значением в корне правого поддерева - числом 8. Так как добавляемое значение меньше значения в корне правого поддерева, то добавляем левое поддерево уже к этому корню (рис. 2, г).

Полностью сформированное бинарное дерево представлено на рис. 3.

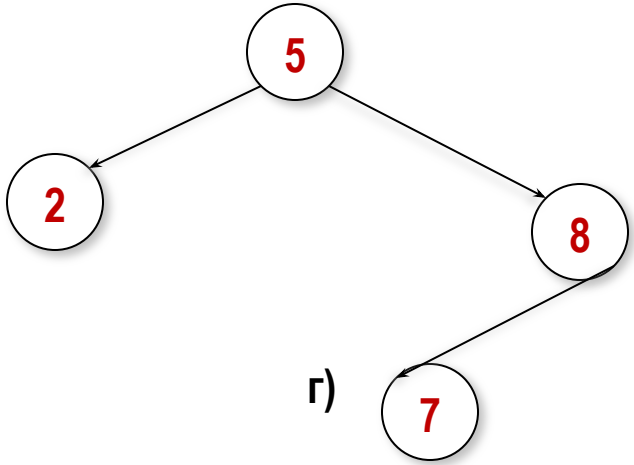
а)



б)



в)



г)

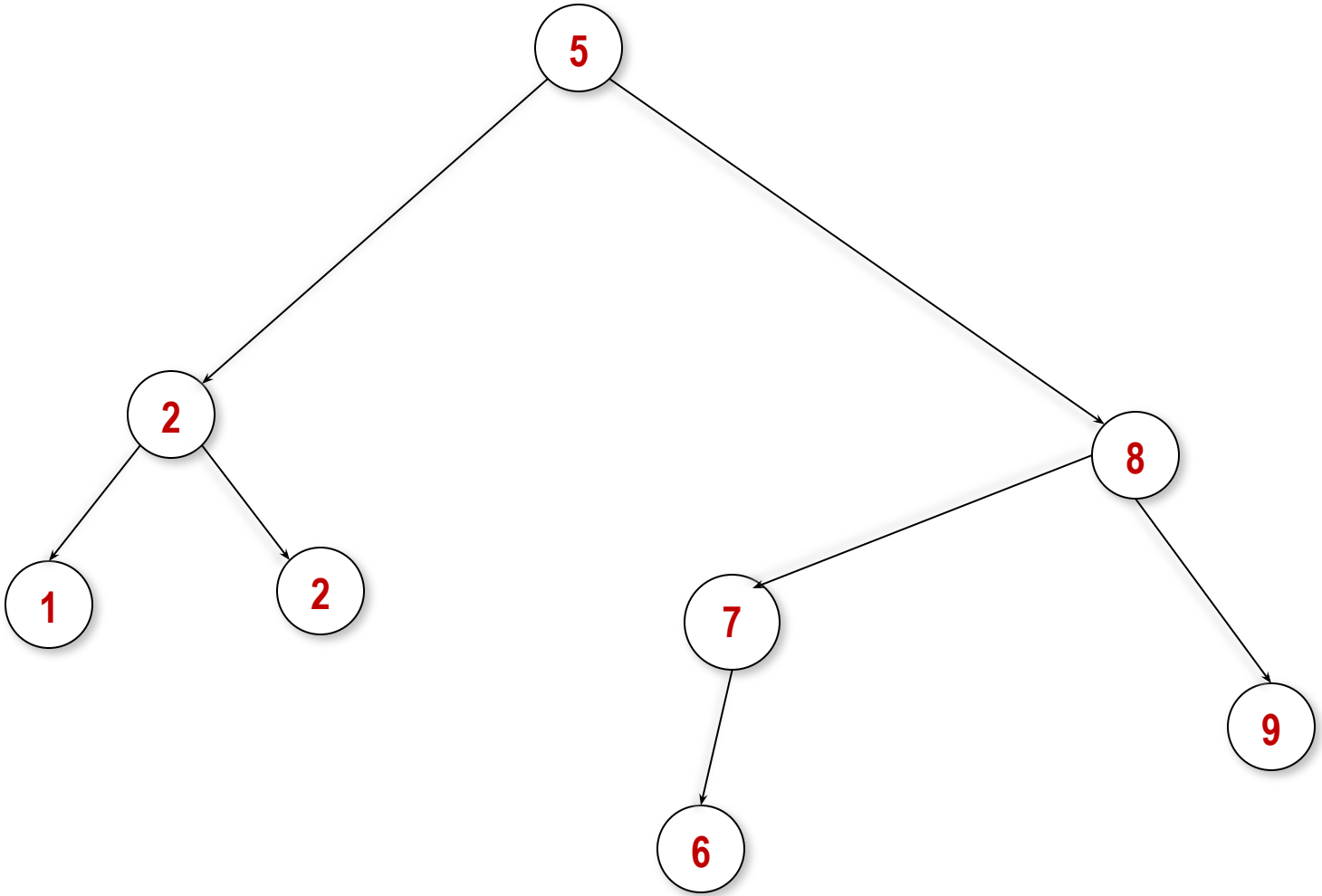


Рис. 3. Полностью сортированное бинарное дерево

Фрагмент программы, реализующий добавление вершины к дереву, состоит из трех частей: создания вершины, поиска корня, к которому можно добавить поддерево, придерживаясь основного правила, и, непосредственно, добавления вершины:

{создание новой вершины}

```
new(q); {выделяем память для нового элемента}
with q^ do {заносим значения}
   begin
     value:=n;
     left:=nil;
     right :=nil;
   end;
```

{поиск корня для добавляемой вершины}

```
pass:=root; {начинаем с корня бинарного дерева}
while pass <> nil do {пока не найдено свободное место}
  begin next:=pass; {сохраняем адрес корня-кандидата}
  if q^.value<pass^.value then pass: =pass^.left {влево}
else  pass:= pass^,right; {вправо}
  end;
```

{добавление вершины}

```
if q^.value<next^.value then {если значение меньше
корня} next^.left:=q {добавляем левое поддерево}
else next^.right:=q; {добавляем правое поддерево}
```

Используя рекурсивность определения дерева, можно построить рекурсивную процедуру добавления вершин к дереву. В качестве параметров эта процедура будет получать указатель на корень дерева и указатель на добавляемый элемент.

```
Procedure Add(Var r: top_ptr; pass:top_ptr);
begin
If r = nil then r:=pass
{если место свободно, то добавляем}
        else {иначе идем налево или направо}
       if (pass^.value < r^.value) then Add(r^.left ,pass)
else Add(r^.right, pass);
end;...
```

2.Поиск вершины в сортированном бинарном дереве. Поиск в сортированном бинарном дереве осуществляется следующим образом:

1)Вначале значение ключа поиска сравнивается со значением в корне.

2)Если значение ключа в искомой вершине меньше, чем в корневой, то поиск переходит в левую ветвь. Если больше или равно - то в правую ветвь. И так в каждой следующей вершине до тех пор, пока не отыщется искомая.

3.Удаление вершины с указанным ключом

Удалению вершины с указанным ключом предшествует ее поиск (см. выше). Непосредственное удаление вершины реализуется в зависимости от того, какая вершина удаляется:

- удаляемая вершина не содержит поддеревьев (лист) - удаляем ссылку на вершину из корня соответствующего поддерева;
- удаляемая вершина содержит одну ветвь : для удаления необходимо скорректировать соответствующую ссылку в корне, заменив адрес удаляемой вершины адресом вершины, из нее выходящей;

- удаляемая вершина содержит две ветви: в этом случае нужно найти подходящую вершину, которую можно вставить на место удаляемой, причем эта подходящая вершина должна легко перемещаться. Такая вершина всегда существует: это либо самый правый элемент левого поддерева, либо самый левый элемент правого поддерева удаляемой вершины

4.Сортировка с использованием дерева.

Так как дерево формируется по определенным выше правилам, то сортировка по возрастанию осуществляется обходом дерева «слева направо». Обход начинается с самого нижнего левого листа или, если такого листа нет, корня. Вывод значений осуществляется в следующем порядке: сначала выводится значение самого нижнего левого поддерева, затем корня, затем самого нижнего левого поддерева правого поддерева и т.д.(рис.4)

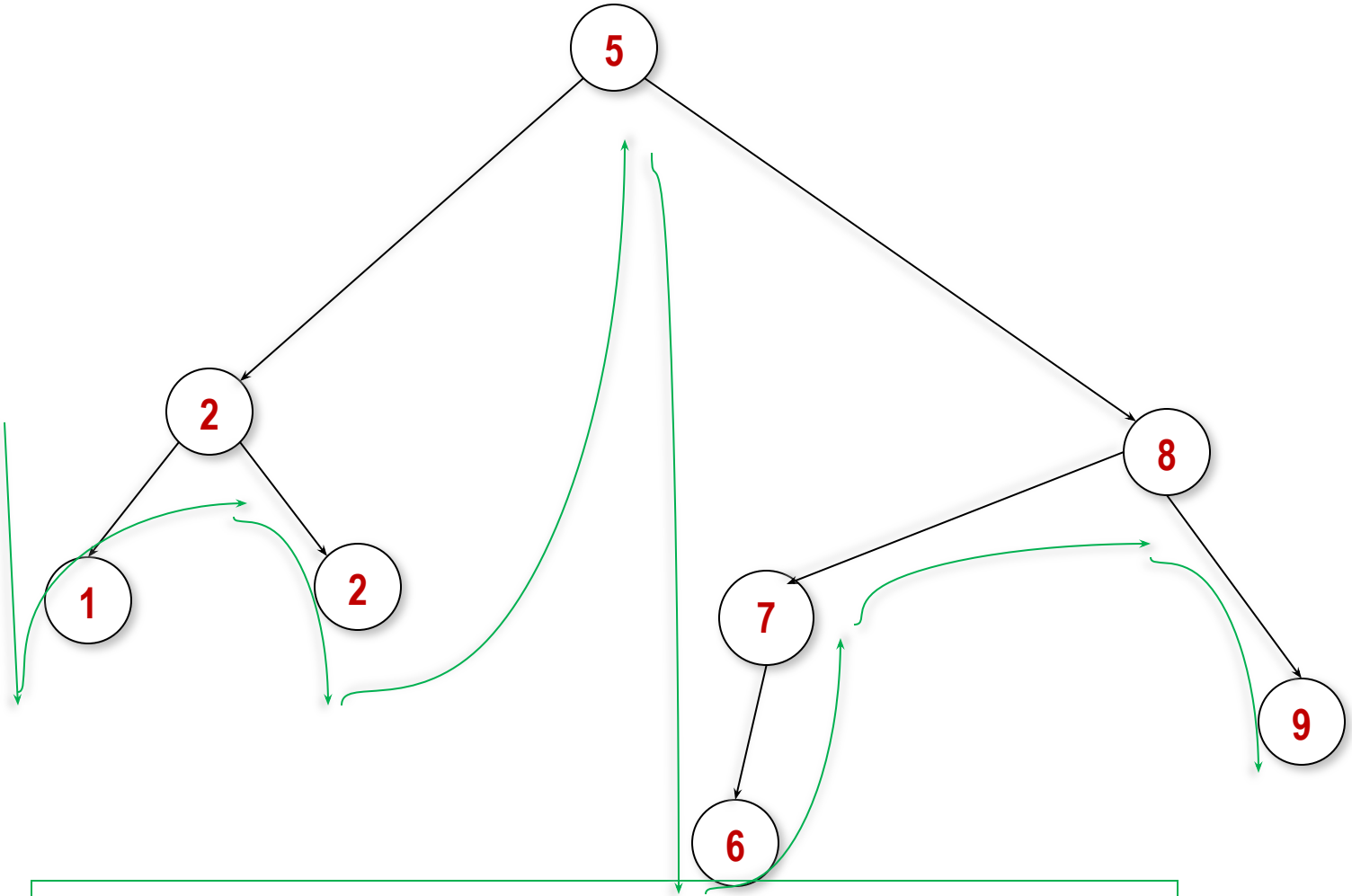


Рис. 4. Обход дерева «слева направо»

Вопросы?