Простые структуры данных

Базовые типы данных:

- целые числа (integer);
- числа с плавающей точкой (real, float);
- логические значения (boolean);
- символы (char);
- строки (string).

(в нотация Паскаля)

Типы данных в языке Python:

- int – целые числа;
- float – вещественные числа;
- bool – логические значения, True (истин) или False (ложь);
- str – символ или символьная строка.

Массив. Фундаментальность массива, как структуры данных, заключается в его прямом соответствии системам памяти почти на всех компьютерах. Для извлечения содержимого слова из памяти машинный язык требует указания адреса. Таким образом, всю память компьютера можно рассматривать как однородный массив, где адреса памяти (номер ячейки) есть не что иное, как индексы элементов массива.

Массив – это переменные одного типа, расположенные в памяти рядом (в соседних ячейках) и имеющие общее имя. Каждый элемент массива имеет уникальный номер.

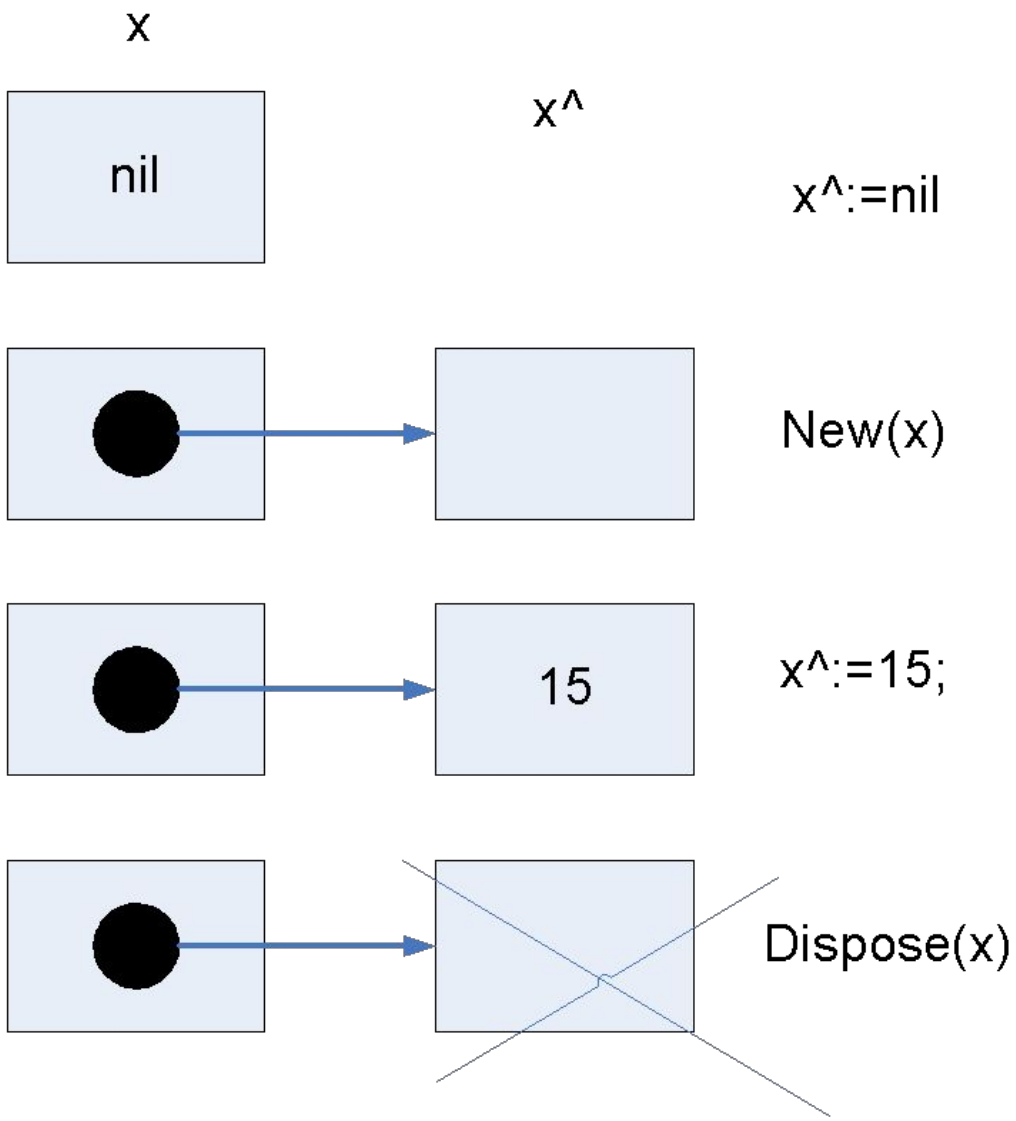
В языке Python такой структуры как «массив» нет. Вместо этого используют списки (list).
Список в Python – это совокупность элементов, каждый из которых имеет свой номер (индекс). Нумерация всегда начинается с нуля. Список – это динамическая структура.

Указатель (переменные ссылочного типа). В нотации языка программирования Паскаль.

```
Type <имя_типа>=^<базовый тип>;
Var <имя_переменной>: <имя_типа>; или
<имя_переменной>: ^ <базовый тип>;
```

Пример.

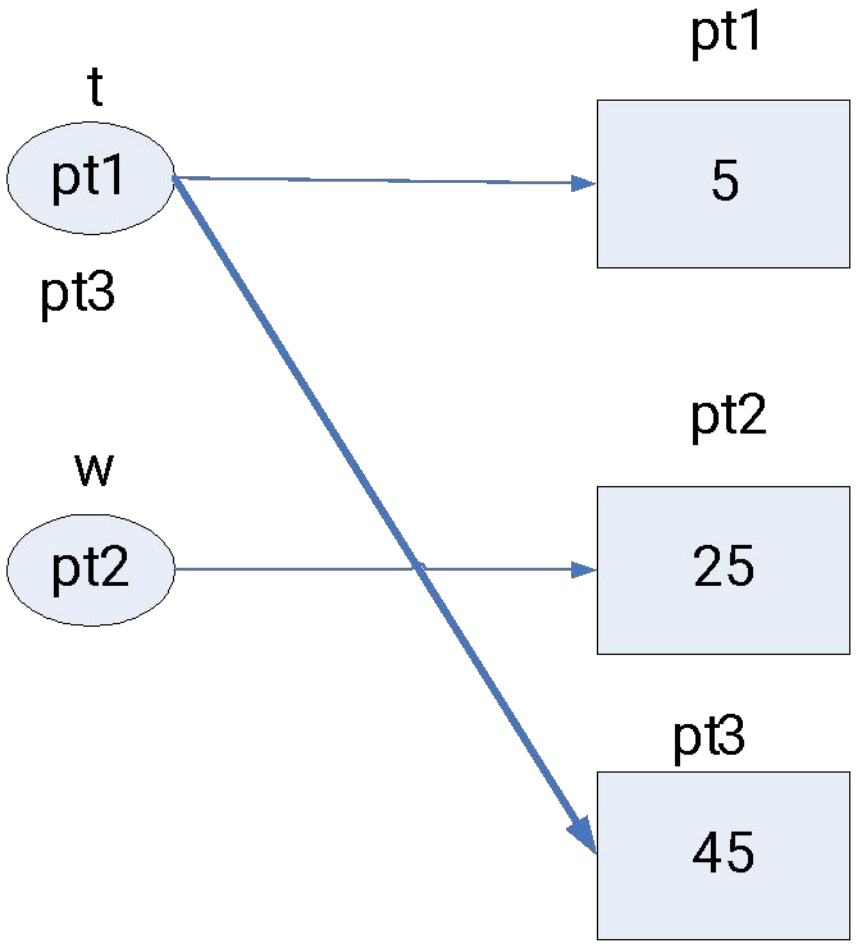
```
Type ss=^Integer;
Var x, y: ss;{Указатели на переменные
целого типа.}
a:   ^Real;{Указатель   на   переменную
вещественного типа.}
```



Основные операторы для работы с указателями

На языке Python

```
t = 5
w = 25
t = 45
```

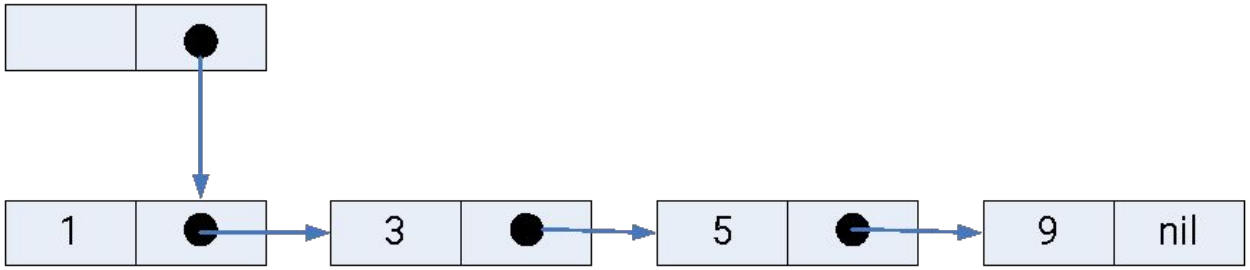


Связный список. Базовая структура данных, в которой каждый элемент содержит информацию, необходимую для получения следующего элемента. Основное преимущество связных списков перед массивами заключается в возможности эффективного изменения расположения элементов. За эту гибкость приходится жертвовать скоростью доступа к произвольному элементу списка, поскольку единственный способ получения элемента состоит в отслеживании связей от начала списка.

Определение. Связный список – это набор элементов, связи между которыми устанавливается с помощью указателей.

Пример

Линейный упорядоченный список из четырех элементов. Целые числа (ключи) вводились в такой последовательности: 3, 5, 1, 9.



Описание элемента списка имеет вид:

```
Type pt=^elem;{Указатель на элемент
списка.}
     elem=Record
     data:Integer;{Поле данных (ключ).}
     next:pt;{Указатель на следующий
   элемент списка.}
    End;
Var first:pt;{Указатель на первый элемент
списка.}
```

Основные операции с элементами списка:

- вывод элементов списка;
- вставка элемента в список (упорядоченный список);
- удаление элемента из списка.

Вывод элементов списка

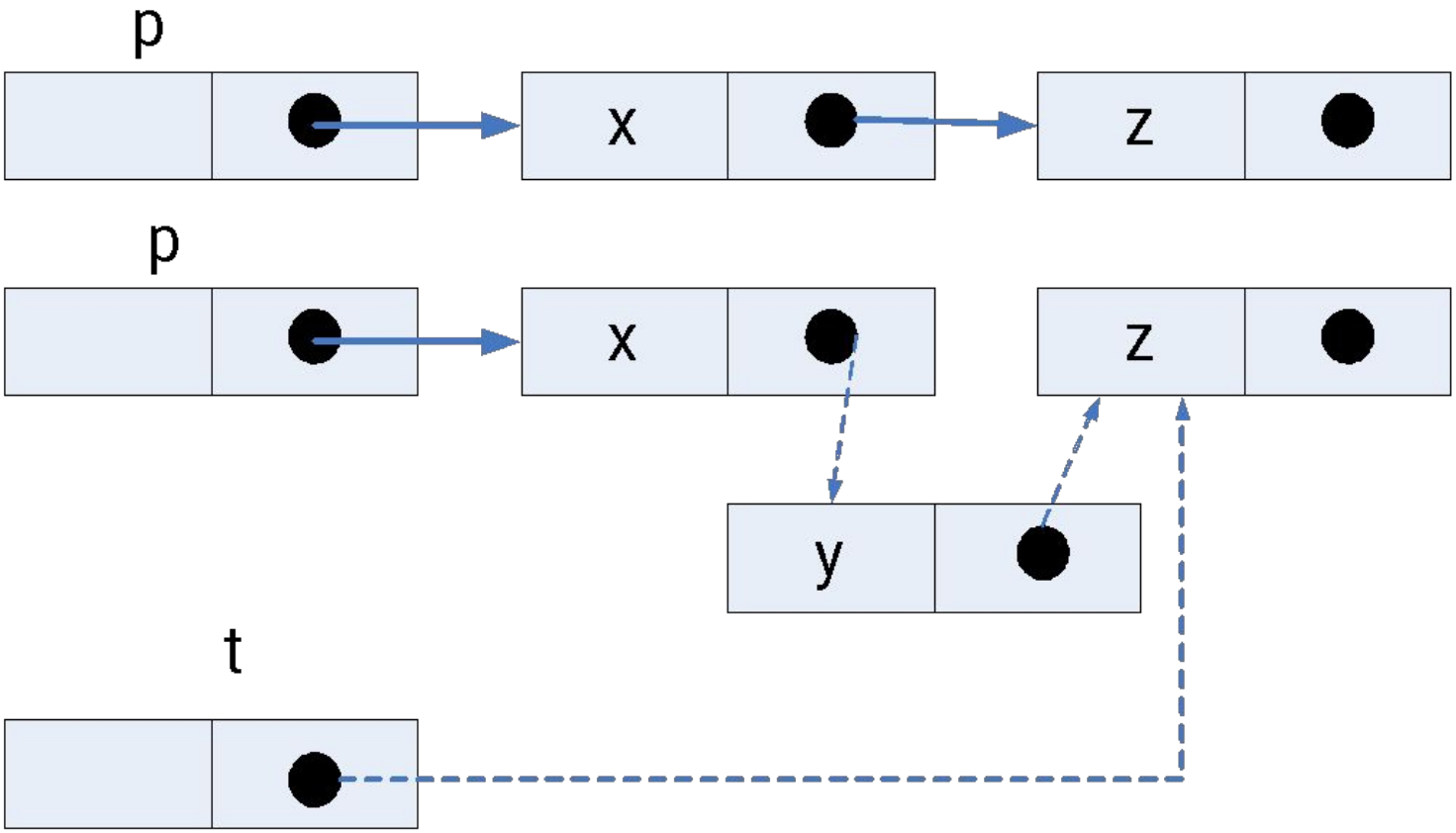
```
Procedure Print(t:pt);
 Begin
  While t<>nil Do Begin
   Write(t^.data,' ');
   t:=t^.next;
  End;
 End;
```

Ее рекурсивная реализация:

```
Procedure Print(t:pt);
 Begin
  If t<>nil Then Begin
   Write(t^.data,' ');
   Print(t^.next);
  End;
 End;
```

Вставка элемента в список

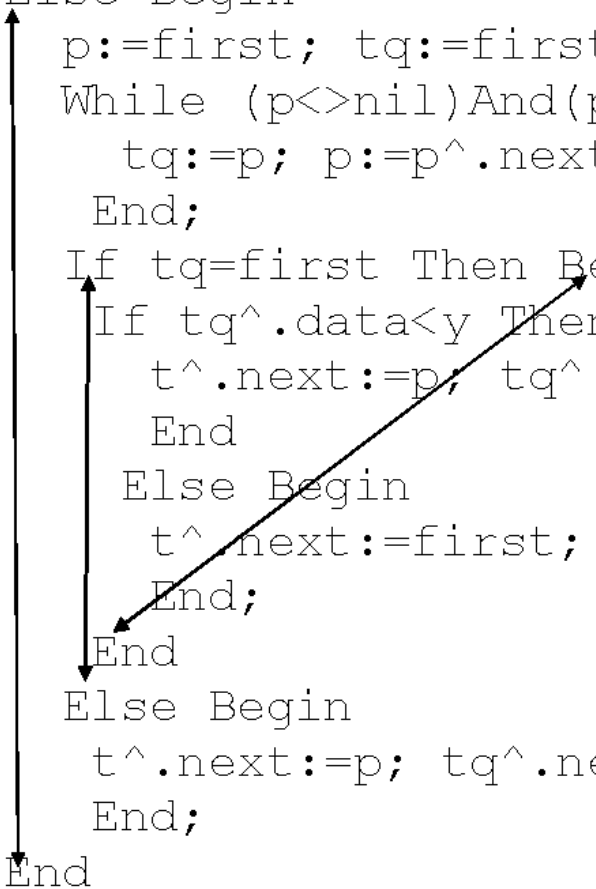
Дано значение указателя p на элемент списка, после которого требуется вставить элемент y. Список до вставки элемента и после вставки.



Логика

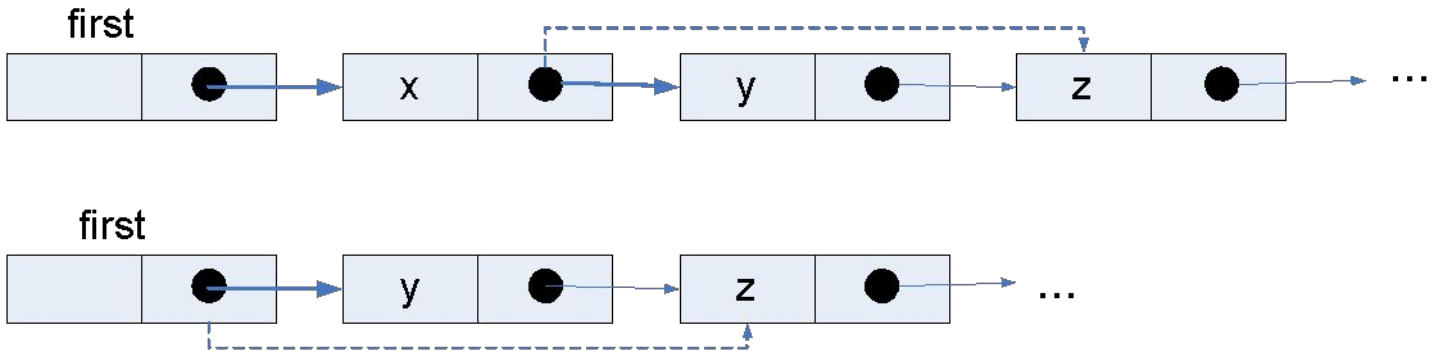
```
процедура insertList(first,y)
  создать новый элемент (указатель t)
  если список пуст (first=nil):
    вставить первый элемент
  иначе:
    p←first,tq←first
    пока список не пройден (p<>nil) и
элемент списка<y:
      tq←p, p←следующий элемент списка
    если tq=first:
      если первый элемент списка <y:
        вставить элемент первым в списке
      иначе: вставить элемент после первого
элемента списка
    иначе: вставить элемент после элемента
tq
```

```
Procedure Ins_List(Var first:pt;y:Integer);
 Var t,p,tq:pt;
 Begin
  New(t);t^.data:=y;t^.next:=nil;
  If first=nil Then first:=t
   Else Begin
     p:=first; tq:=first;
     While (p<>nil)And(p^.data<y)  Do Begin
       tq:=p; p:=p^.next;
      End;
     If tq=first Then Begin
      If tq^.data<y Then Begin
        t^.next:=p; tq^.next:=t;
        End
      Else Begin
        t^.next:=first; first:=t;
        End;
     End
     Else Begin
      t^.next:=p; tq^.next:=t;
      End;
   End
 End;
```



Удаление элемента из списка

Действия при удалении элемента из списка сводятся к его поиску, а затем к переадресации от предшествующего удаляемому элементу, к элементу следующим за удаляемы. Единственная сложность заключается в том, чтобы предусмотреть случай удаления первого элемента списка – изменяется значение переменной `first`. Удаляется элемент с ключом y.



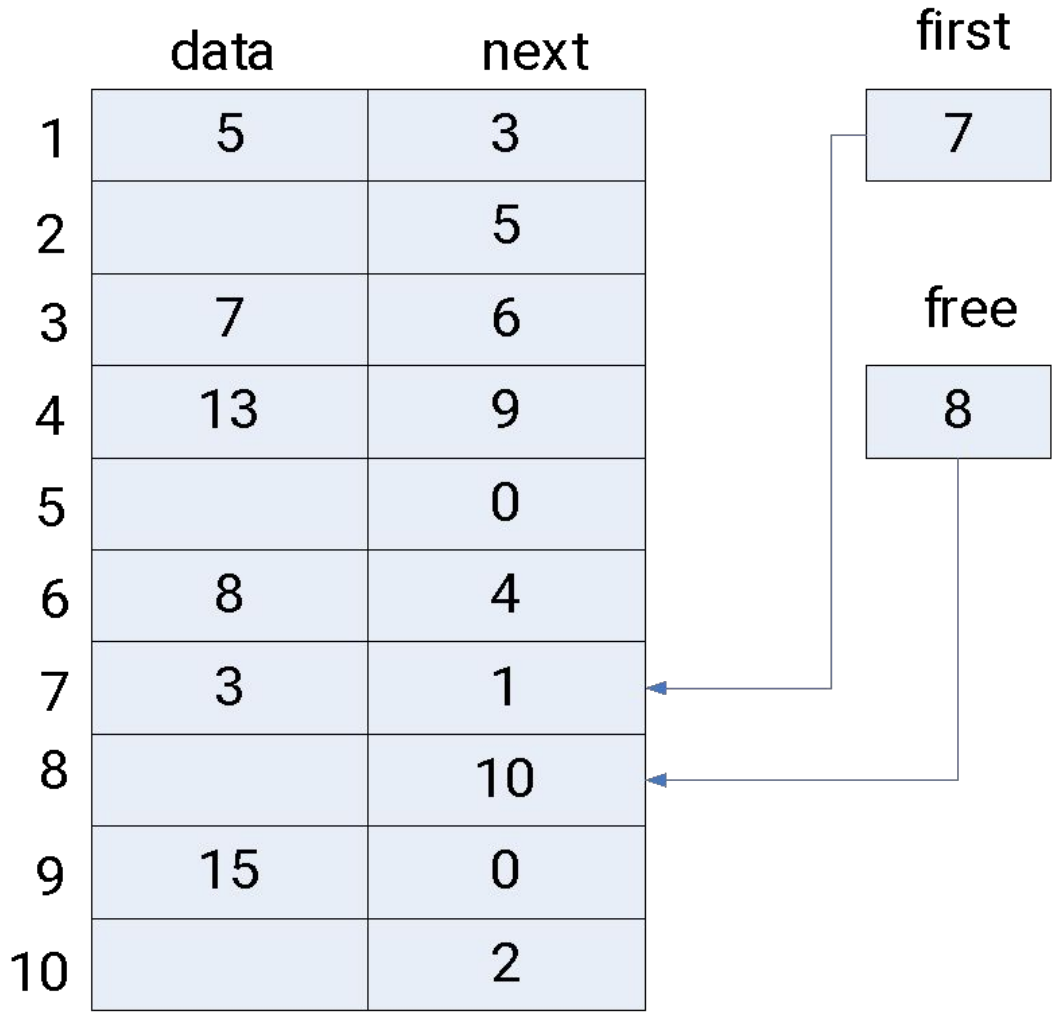
Логика

```
процедура deleteList(first,y)
  t←first
  пока список не пройден:
   если ключ элемента = y:
    если t=first:
     удалить первый элемент
    иначе:
     удалить элемент (t)
   иначе:
    перейти к следующему элементу списка
```

В процедуре Del_List из списка удаляются все элементы, равные y.

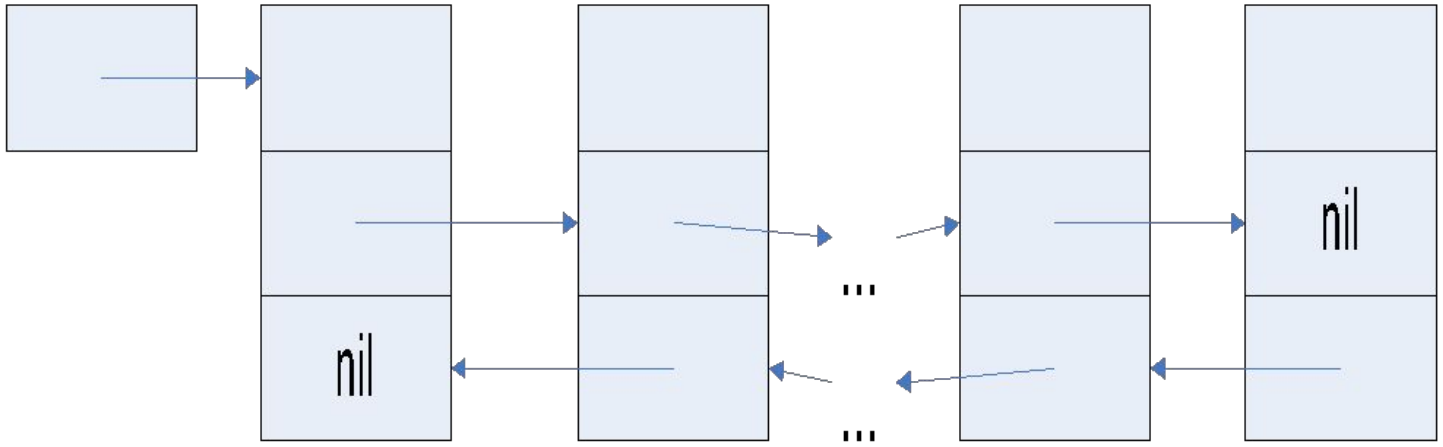
```
Procedure Del_List(Var first:pt; y:Integer);
 Var t,x,dx:pt;
 Begin
  t:=first;
  While t<>nil Do
   If t^.data=y Then
    If t=first Then Begin
      x:=first;first:=first^.next;
      Dispose(x);
      t:=first;
      End
     Else Begin
       x:=t; t:=t^.next; dx^.next:=t;
       Dispose(x);
      End
   Else Begin dx:=t;:=t^.next;End
 End;
```

Реализация линейного списка с использованием массивов



Пример представления списка с использованием массивов

Двусвязные списки



Двусвязный список

Описание элемента списка в этом случае имеет вид:

```
Type pt=^elem;{Указатель на элемент списка.}
     elem=Record
     data:Integer;{Поле данных (ключ).}
     next, prev:pt;{Указатели на следующий и
  предшествующий элементы списка.}
     End;
Var head:pt;{Указатель на первый элемент
списка.}
```

Абстрактные типы данных

Определение. Абстрактный тип данных (АТД) — это тип данных (набор значений и совокупность операций для этих значений), доступ к которому осуществляется только через интерфейс.

Стек

Стек магазинного типа — это АТД, который включает две основные операции: вставить, или затолкнуть (push) новый элемент и удалить, или вытолкнуть (pop) элемент, вставленный последним. Принцип: последним пришел, первым ушел (last-in, first-out, сокращенно LIFO).

Реализация стека через линейный список

```
Type pt=^elem;
     elem=Record
      data: Integer;
      next: pt;
     End;
Var head:pt;
```

Если стек пуст, то значение указателя `head` равно `nil`. Эта проверка реализуется функцией `Empty` с результатом логического типа.

```
Function Empty(head:pt):Boolean;
 Begin
  If head=nil Then Empty:=False
   Else Empty:= True;
 End;
```

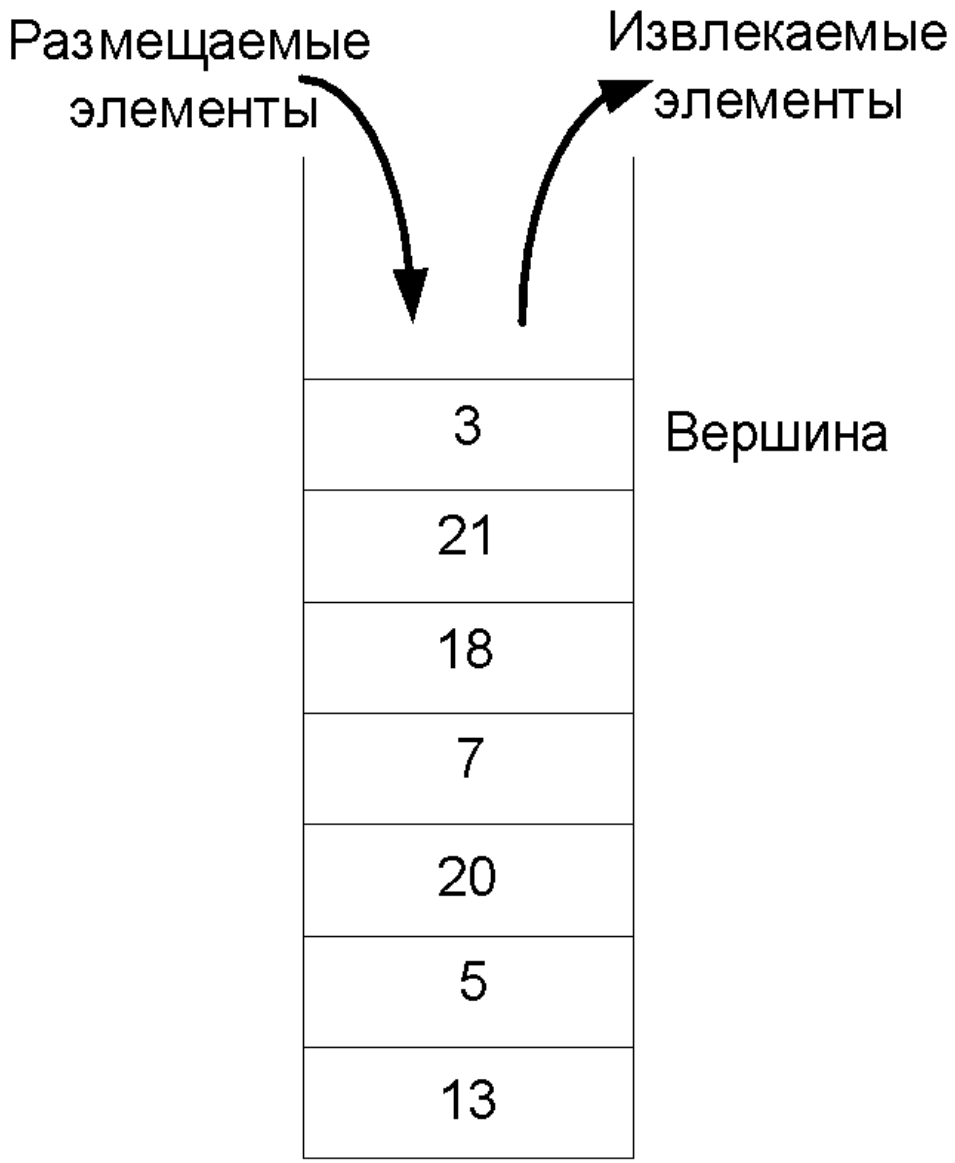
Процедура записи элемента в стек должна содержать два параметра: первый определяет указатель на начало стека, второй – записываемое в стек значение. Запись в стек производится аналогично вставке нового элемента в начало списка.

```
Procedure Push(Var head:pt; x:Integer);
 Var t:pt;
 Begin
  New(t);
  t^.data:=x;
  t^.next:=head;
  head:=t;
 End;
```

Извлечение элемента из непустого стека. В результате выполнения этой операции некоторой переменной x должно быть присвоено значение первого элемента стека и изменено значение указателя на начало списка.

```
Procedure Pop(Var head:pt; Var x:Integer);
  Var t:pt;
  Begin
   x:=head^.data;
   t:=head;
   head:=head^.next;
   Dispose(t);
  End;
```

Реализация стека с использованием массива



Очередь

Очередь – это упорядоченный набор элементов, в котором извлечение элементов происходит с одного его конца, а добавление новых элементов – с другого (аббревиатура *FIFO – first-in-first-out*: первым вошел – первым вышел). Основные операции с очередью:

- запись элемента в очередь, если это возможно, то есть нет переполнения структуры данных, выбранной для хранения элементов очереди;
- чтение элемента из очереди, естественно, если очередь не пуста.

```
Type pt=^elem;
     elem=Record
      data: Integer;
      next: pt;
     End;
Var head, tail:pt;
```

Если очередь пуста (нет элементов), то значение указателей head и tail равно nil. Эта проверка реализуется функцией Empty с результатом логического типа.

```
Function Empty(head:pt):Boolean;
 Begin
  If head=nil Then Empty:=False
   Else Empty:=True;
 End;
```

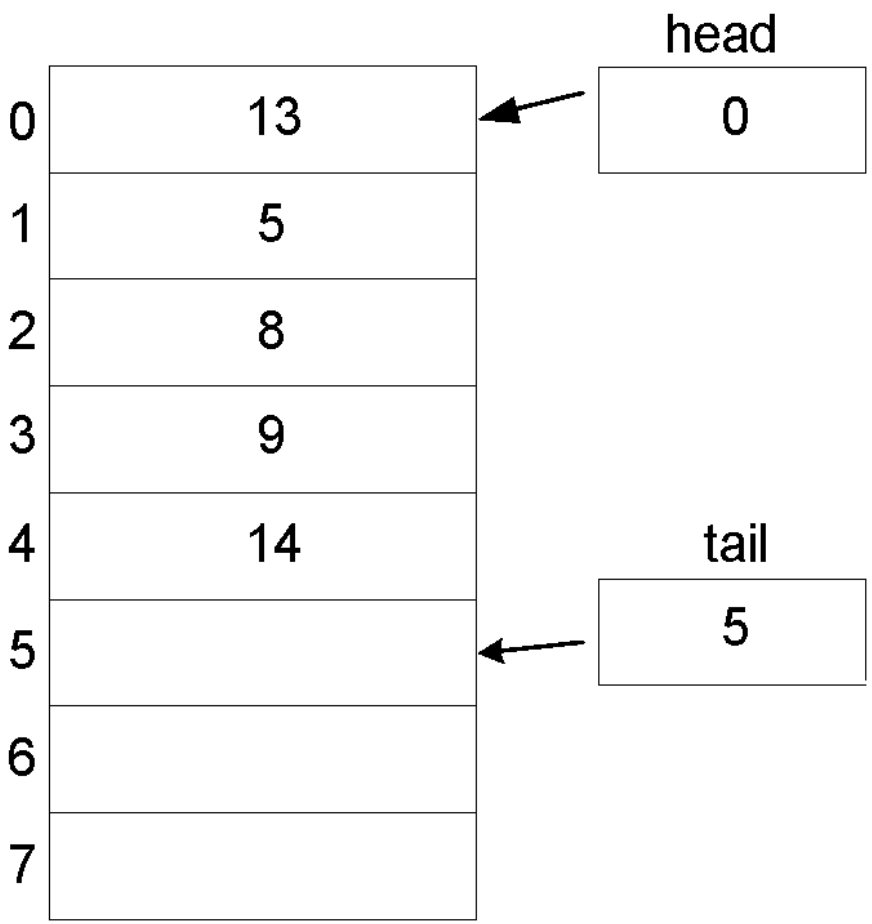
Процедура записи элемента не в пустую очередь должна содержать два параметра: первый определяет указатель на конец очереди, второй – записываемое значение.

```
Procedure Queue_Write(Var tail:pt;
x:Integer);
 Var t:pt;
 Begin
  New(t);
  t^.data:=x;
  t^.next:=nil;
  tail^.next:=t;
  tail:=t;
 End;
```

Извлечение элемента из непустой очереди сводится к чтению первого элемента списка.

```
Procedure Queue_Read(Var
head:pt; Var x:Integer);
 Var t:pt;
 Begin
  x:=head^.data;
  t:=head;
  head:=head^.next;
  Dispose(t);
 End;
```

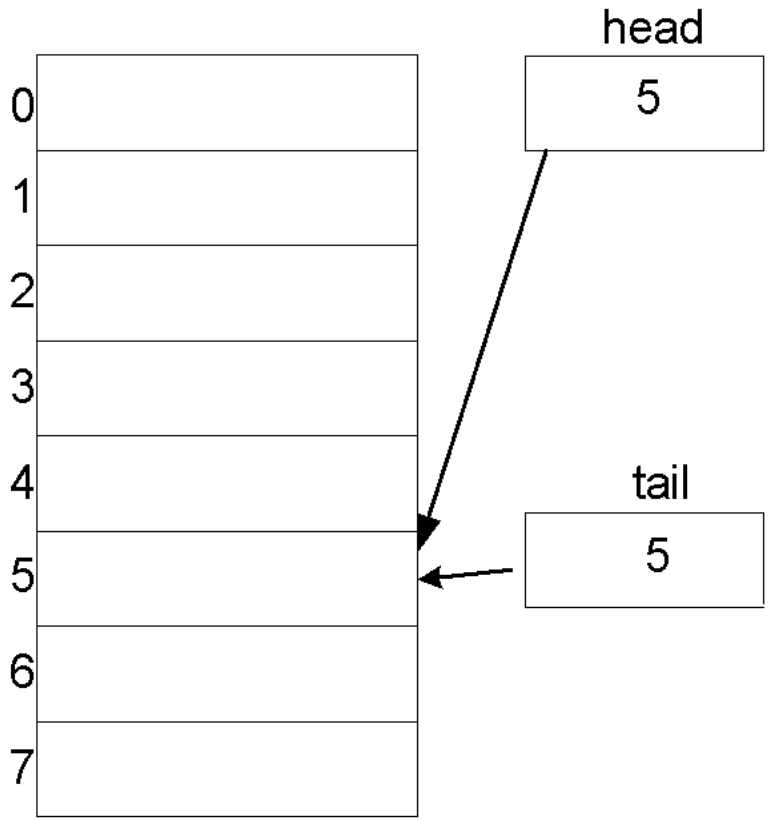
Реализация очереди через массив



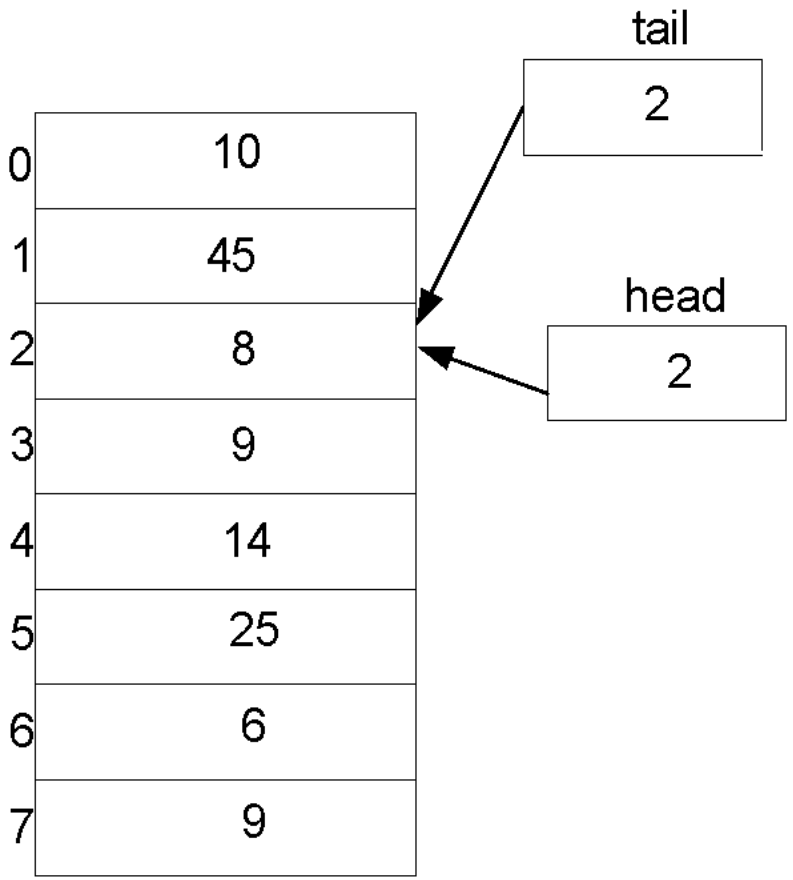
Начальные состояния::
head=0;
tail=0.

Запись:
A[tail]:=x;
tail:=(tail+1) Mod n.

Чтение:
x:=A[head];
head:=(head+1) Mod n.



а)



б)

Очередь пуста а) – *head=tail*; Очередь заполнена б) – *head=tail*

```
Function Queue_Full(head,tail:Integer;p:Boolean): Boolean;
 Begin
   If (head=tail) And Not p Then Queue_Full:=True
    Else Queue_Full:=False;
 End;
```

```
Function Queue_Empty(head,tail:Integer; p:Boolean):
Boolean;
 Begin
  If (head=tail) And p Then Queue_Empty:=True
    Else Queue_Empty:=False;
 End;
```

Как изменятся операции чтения и записи в очередь?

Деревья

Деревья — это математические абстракции, играющие фундаментальную роль при разработке и анализе алгоритмов.

Дерево (tree) — это непустая совокупность вершин и ребер, удовлетворяющих определенным требованиям. Вершина (vertex) — это простой объект (называемый также узлом (node)), который может иметь имя и содержать другую связанную с ним информацию; ребро (edge) — это связь между двумя вершинами. Путь (path) в дереве — это список отдельных вершин, в котором следующие друг за другом вершины соединяются ребрами дерева. Определяющее свойство дерева — **существование только одного пути, соединяющего любые два узла. Если между какой-либо парой узлов существует более одного пути или если между какой-либо парой узлов путь отсутствует, мы имеем граф, а не дерево.** Несвязанный набор деревьев называется бором.

Дерево с корнем (единственным, rooted) — это дерево, в котором один узел назначен корнем (root) дерева. Обычно деревья с корнем рисуются с корнем, расположенным в верхней части, и говорят, что узел `y` располагается под узлом `x` (а `x` располагается над `y`), если `x` находится на пути от `y` к корню (т.е., `y` находится под `x` и соединяется с `x` путем, который не проходит через корень). Каждый узел (за исключением корня) имеет только один узел над ним, который называется его родительским узлом (`parent`); узлы, расположенные непосредственно под данным узлом, называются его дочерними узлами (`children`). Иногда аналогия с генеалогическими деревьями расширяется еще больше, и тогда говорят об узлах-предках (`grand parent`) или родственных (`sibling`) узлах данного узла.

Бинарное дерево (`binary tree`) — это упорядоченное дерево, состоящее из узлов двух типов: внешних узлов без дочерних узлов и внутренних узлов, каждый из которых имеет ровно два дочерних узла. Поскольку два дочерних узла каждого внутреннего узла упорядочены, говорят о левом дочернем узле (`left child`) и правом дочернем узле (`right child`) внутренних узлов. Каждый внутренний узел должен иметь и левый, и правый дочерние узлы, хотя один из них или оба могут быть внешними узлами.

Математические свойства бинарных деревьев

Утверждение 1. Бинарное дерево с n внутренними узлами имеет n+1 внешних узлов.

Эта лемма доказывается методом индукции: бинарное дерево без внутренних узлов имеет один внешний узел, следовательно, лемма справедлива для n=0. Для n>0 любое бинарное дерево с n внутренними узлами имеет k внутренних узлов в левом поддереве и n–1–k внутренних узлов в правом поддереве для некоторого k в диапазоне между 0 и n–1, поскольку корень является внутренним узлом. В соответствии с индуктивным предположением левое поддерево имеет k+1 внешних узлов, а правое поддерево – n–k внешних узлов, что в сумме составляет n+1.

Утверждение 2. Бинарное дерево с n внутренними узлами имеет 2·n связей: n–1 связей с внутренними узлами и n+1 связей с внешними узлами.

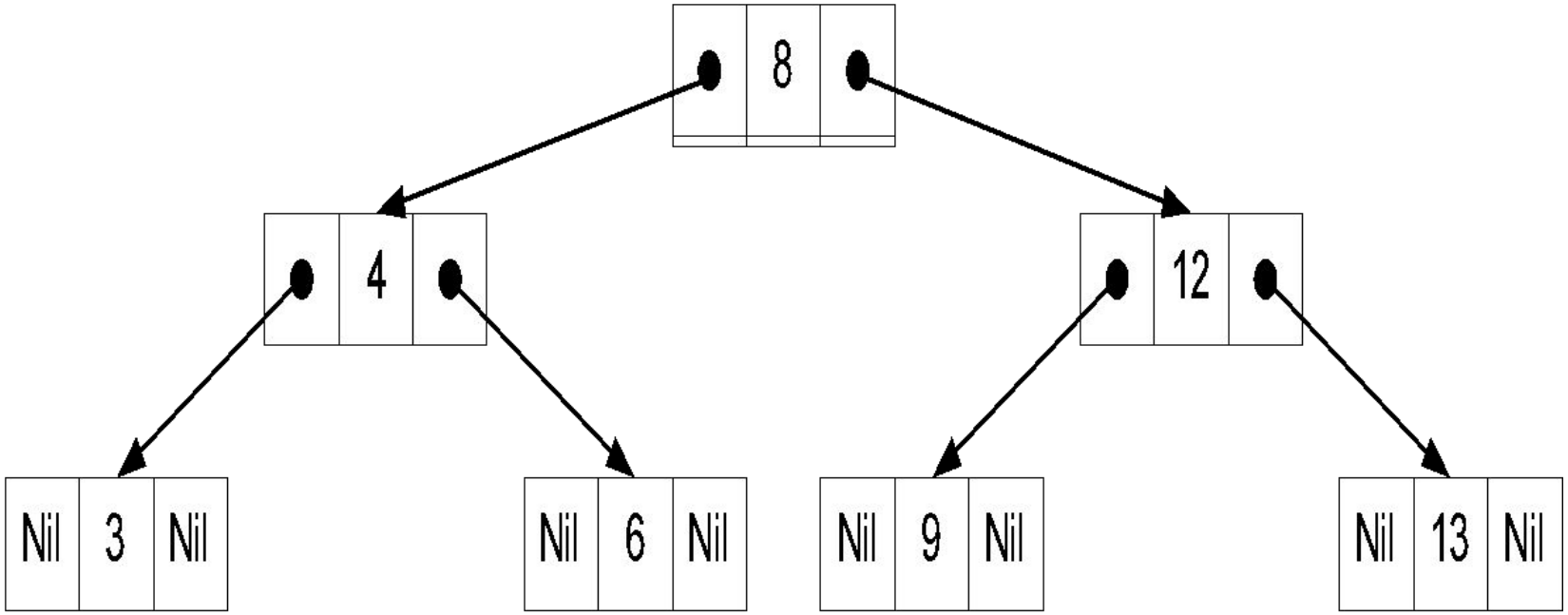
В дереве с корнем каждый узел, за исключением корня, имеет единственный родительский узел, и каждое ребро соединяет узел с его родительским узлом; следовательно, существует n–1 связей, соединяющих внутренние узлы. Аналогично, каждый из n+1 внешних узлов имеет одну связь со своим единственным родительским узлом.

Утверждение 3. Высота бинарного дерева с n внутренними узлами не меньше $\log_2 n$ и не больше n–1.

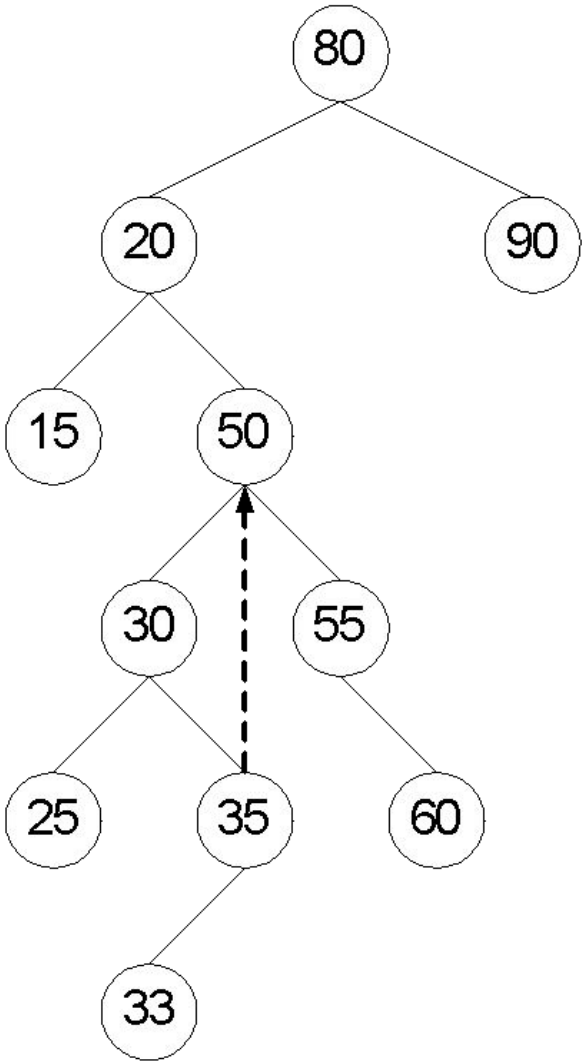
Худший случай — вырожденное дерево, имеющее только один лист и n– связь от корня до листа. В лучшем случае мы имеем уравновешенное дерево с 2^i внутренними узлами на каждом уровне i, за исключением самого нижнего. Если высота равна h, то должно быть справедливо соотношение $2^{h-1} < n+1 < 2^h$ поскольку существует n+1 внешних узлов. Из этого неравенства следует провозглашенное утверждение: в лучшем случае высота равна $\log_2 n$, округленному до ближайшего целого числа.

Стандартная реализация двоичного дерева поиска

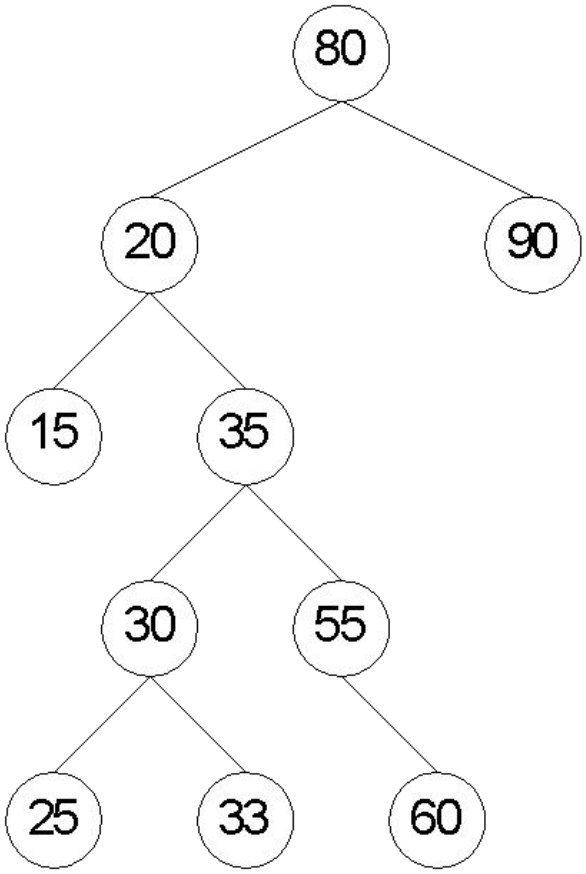
```
Type pt=^node;
     node=Record
     data:Integer;{Ключ.}
     left,right:pt;{Ссылки на левого и правого
потомков.}
    End;
Var root:pt;{Указатель на корень дерева.}
```



```
Procedure Ins_Tree(Var t:pt;x:Integer);{x - значение
вставляемого элемента.}
  Begin
   If t=nil Then Begin
     New(t);
     With t^ Do Begin
       left:=nil;
       right:=nil;
       data:=x;
      End;
     End
    Else If x<=t^.data Then Ins_Tree(t^.left,x)
          Else Ins_Tree(t^.right,x);
  End;
```



а)



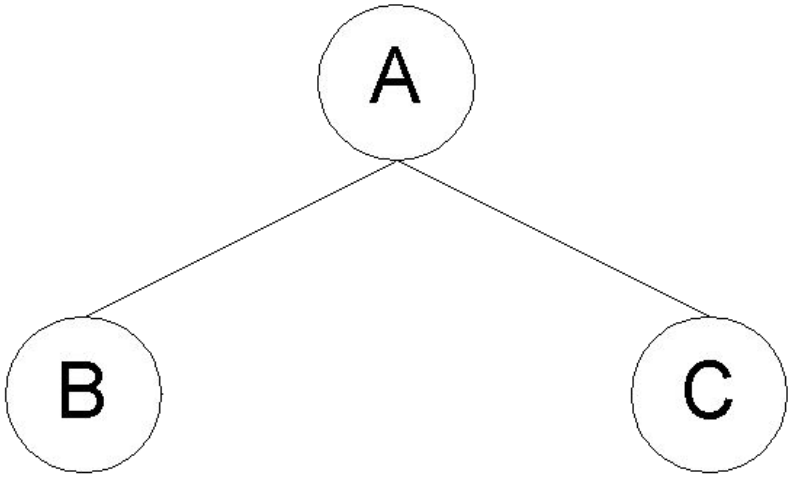
б)

Двоичное дерево поиска до удаления элемента 50 и после его удаления

```
Procedure deleteTree(var t:pt,x:Integer)
  Var q:pt;
 Procedure del(w:pt);
  Begin
   …
  End;
Begin
   If t<>nil Then
    If x<t^.data Then deleteTree(t^.left,x)
     Else If x>t^.data Then deleteTree(t^.right,x)
          Else Begin {Элемент найден. Приступаем
к его удалению.}
               q:=t;
               If t^.right=nil Then t:=t^.left
{Правого поддерева нет.}
                Else If t^.left=nil Then t:=t^.right
{Левого поддерева нет.}
                     Else del(t^.left);{Находим
самый правый элемент в левом поддереве.}
               Dispose(q);
              End;
  End;
```

```
  Procedure del(Var w:pt);{Поиск
самого правого элемента в дереве.}
   Begin
    If w^.right<>nil Then
del(w^.right)
     Else Begin
      q:=w;{Запоминаем адрес для
того, чтобы освободить место в
«куче».}
       t^.data:=w^.data;
       w:=w^.left;
      End;
   End;
```

```
Procedure Print_Tree(t:pt);
  Begin
   If t<>nil Then Begin
    Print_Tree(t^.left); {1}
    Write(t^.data:3); {2}
    Print_Tree(t^.right); {3}
   End;
  End;
```



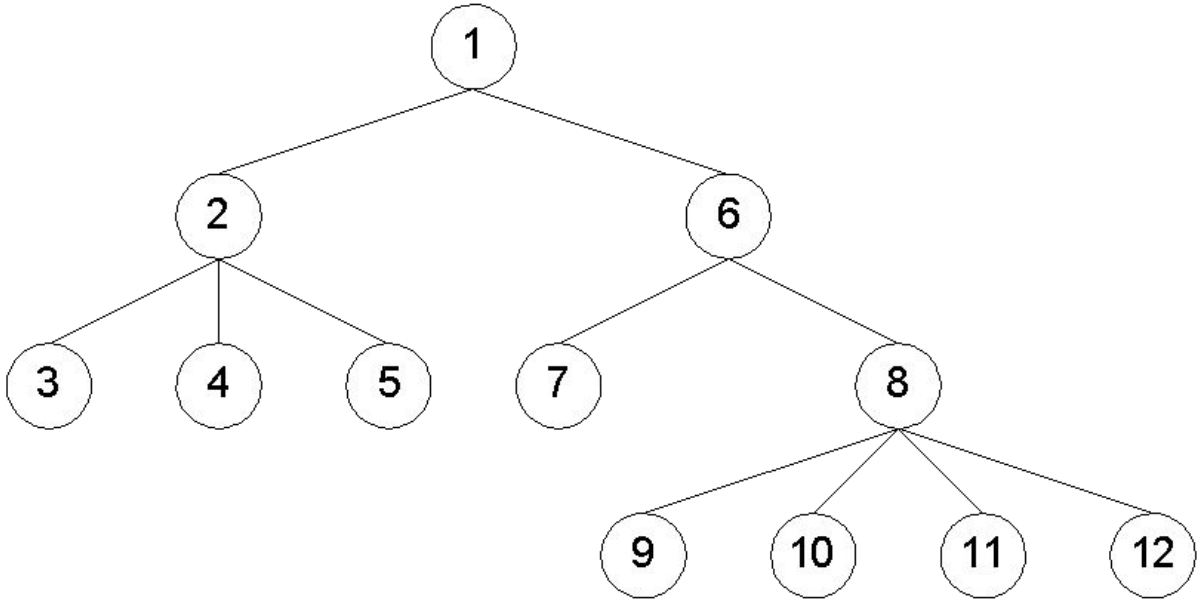
Способ	*Очередность вывода*
1, 2, 3	B A C
1, 3, 2	B C A
2, 1, 3	A B C
2, 3, 1	A C A
3, 1, 2	C B A

Способы описания деревьев

Значение переменной *root* равно 18, *headfree* – 12.

Номер элемента массива	*data*	*left*	*right*	*next*
1	0	0	0	3
2	90	0	0	0
3	0	0	0	6
4	15	0	0	0
5	50	19	7	0
6	0	0	0	9
7	55	0	10	0
8	25	0	0	0
9	0	0	0	17
10	60	0	0	0
11	33	0	0	0
12	0	0	0	13
13	0	0	0	1
14	0	0	0	15
15	0	0	0	
16	20	4	5	
17	0	0	0	14
18	80	16	2	
19	30	8	20	
20	35	11	0	0

Описание произвольного дерева в виде одного массива. Пусть у дерева T вершины имеют метки 1, 2, …, n. В этом случае возможно описание дерева с помощью одного одномерного массива (например, A) так, как это показано на рис. 4.9.



а) Дерево

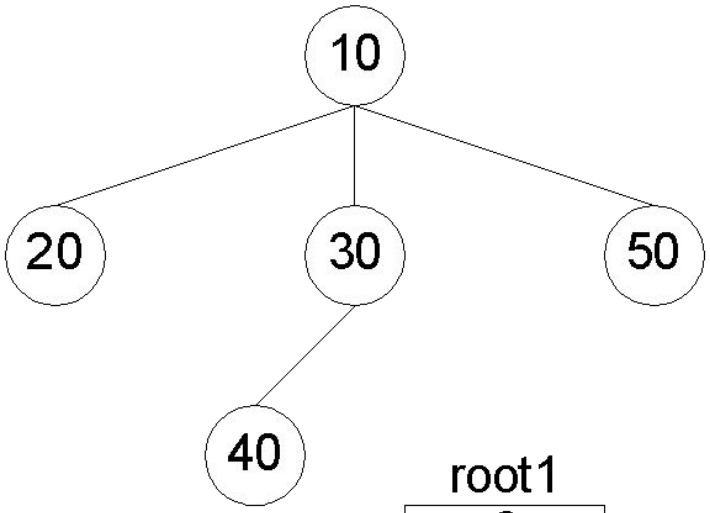
1	2	3	4	5	6	7	8	9	10	11	12
0	1	2	2	2	1	6	6	8	8	8	8

б) Массив указателей на вершины родителей

Дерево и массив указателей на вершины родителей

Описание дерева по принципу «левый сын и правый брат»

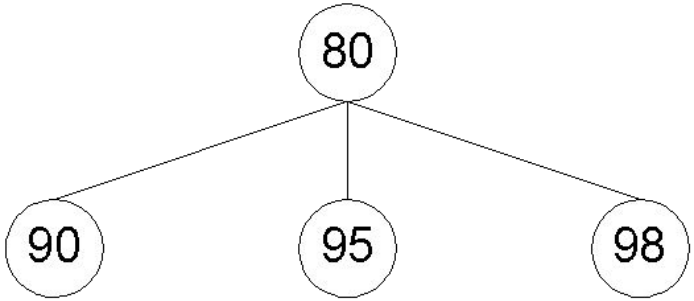
Пусть необходимо выполнить операцию Create(v,root1,root2). Объединяются два дерева: T_1 (указатель на корень root1) и T_2 (указатель на корень root2) – создается вершина с меткой v и двумя сыновьями, которыми являются корни деревьев T_1 и T_2. Рассмотрим операцию на конкретном примере.



root1
6

firstfree
2

	leftchild	data	rightsibling	free
1	/	40	/	
2				5
3	/	50	/	
4	/	20	8	
5				7
6	4	10	/	
7				9
8	1	30	3	
9				10
10				11
11				12
12				13
13				/

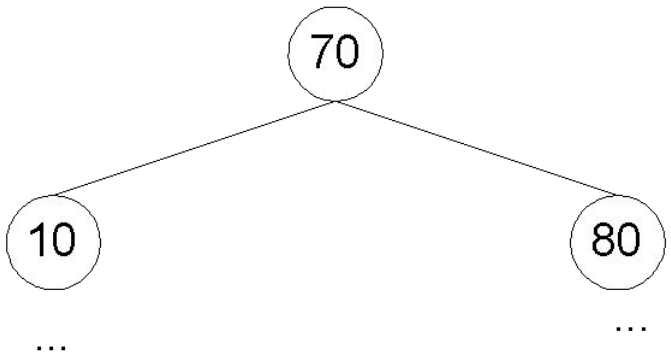


root1
6

firstfree
2

root2
10

	leftchild	data	rightsibling	free
1	/	40	/	
2				7
3	/	50	/	
4	/	20	8	
5	/	95	12	
6	4	10	/	
7				9
8	1	30	3	
9				11
10	13	80	/	
11				/
12	/	98	/	
13	/	90	5	



root1
2

firstfree
7

	leftchild	data	rightsibling	free
1	/	40	/	
2	6	70	/	
3	/	50	/	
4	/	20	8	
5	/	95	12	
6	4	10	10	
7				9
8	1	30	3	
9				11
10	13	80	/	
11				/
12	/	98	/	
13	/	90	5	

Результат выполнения операции Create(70,root1,root2)