# СТРУКТУРЫ ДАННЫХ

Термины *тип данных*, *абстрактный тип данных* и *структура данных* звучат очень похоже, но имеют различный смысл.

Напомним, что в языках программирования *тип данных* определяет внутреннее представление данных в памяти компьютера, объем оперативной памяти, выделяемой для размещения значения данного типа, множество значений, которые могут принимать величины этого типа а также операции и стандартные функции, которые можно применять к величинам этого типа.

Например, в C# целый тип int имеет знаковый формат представления данных в памяти ЭВМ, занимает в оперативной памяти 4 байта, множество его значений лежит в диапазоне от –2147483648 до 2147483647, а к величинам этого типа могут применяться: унарный минус (-), арифметические операции (/, *, +, –, %), операции отношения (<, <=, >, >=, =, !=), операции присваивания (=, +=, –=, *=, …) и т.д.

*Абстрактный тип данных (АТД)* – это некоторая математическая модель с совокупностью операций, определенных в рамках этой модели. Простым примером АТД может служить множество целых чисел с операциями пересечения, объединения и разности. Любой алгоритм разрабатывается в терминах АТД. Для представления АТД используются *структуры данных*, которые представляют собой набор переменных, возможно, различных типов данных, объединенных определенным образом.

В данном разделе мы рассмотрим такие АТД как списки, деревья и графы, а также их программную реализацию на языке C#.

## Списки

АДТ *список* – это последовательность элементов $a_1$, $a_2$,…,$a_n$ ($n \geq 0$) одного типа. Количество элементов *n* называется *длиной списка*. Если n = 0, то мы имеем *пустой список*. Элементы списка линейно упорядочены в соответствии с их позицией в списке. Так $a_1$ – первый элемент списка, $a_i$ *следует* за $a_{i-1}$ и *предшествует* $a_{i+1}$, $a_n$ – последний элемент списка.

АТД список может быть реализован с помощью массива, или ссылок. В данном пособии мы будем рассматривать реализацию списков с помощью ссылок, а при употреблении слова «список» иметь ввиду его программную реализацию в виде ссылок.

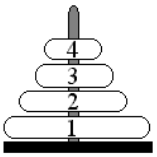
Отметим, что связать последовательность элементов определенного типа в список с помощью ссылок можно сделав так, чтобы каждый элемент содержал ссылку на следующий элемент последовательности. Организованный таким образом список называется *однонаправленным* (односвязным). Если каждый элемент списка содержит две ссылки (одну на следующий элемент в списке, а вторую на предыдущий), то такой список называется *двунаправленным* (двусвязным). Если же последний элемент связать ссылкой с первым, то получится кольцевой список.

Далее подробно рассмотрим такие АТД как стек, очередь и список общего вида, а также их программную реализацию с спомощью ссылок.

### Стек

*Стек* – это частный случай списка, добавление элементов в который и выборка элементов из которого выполняются с одного конца, называемого вершиной стека (головой, или head). При выборке элемент исключается из стека. Другие операции со стеком не определены. Говорят, что стек реализует принцип обслуживания LIFO (last in – fist out, последним пришел –

первым вышел). Стек проще всего представить в виде пирамиды, на которую надевают кольца.




Достать первое кольцо можно только после того, как будут сняты все верхние кольца.

Рассмотрим организацию стека с помощью однонаправленного списка, изображенного на Рис. 12.

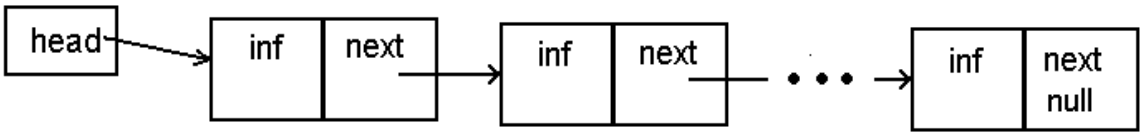



**Рис. 12. Стек.**

Каждый элемент предложенной структуры данных имеет:

- информационное поле inf, которое может быть любого типа и предназначено для хранения значений, например, чисел, строк, структур, или ссылок на внешние по отношению к стеку объекты;
- ссылочное поле next, которое будет использоваться для хранения ссылки на следующий элемент стека.

Предложенная структура базового элемента стека может быть реализована с помощью следующего класса:

```
private class Node //вложенный класс, реализующий элемент стека
{
   private object inf;
   private Node next;
   public Node(object nodeInfo)
   {
      inf = nodeInfo;
      next = null;
   }
   public Node Next
   {
      get { return next; }
      set { next = value; }
   }
   public object Inf
   {
      get { return inf; }
      set { inf = value; }
   }
}
```

Следует обратить внимание на то, что класс Node определяется рекусивно – поле next имеет тип Node. Таким образом мы организует связь между элементами списка.

Теперь создадим класс Stack, в котором будут определены:

1) вложенный класс для реализации базового элемента стека;
2) закрытое поле head, хранящее ссылку на вершину стека;
3) конструктор класса;
4) метод Push, позволяющий добавить элемент в вершину стека;
5) метод Pop, позволяющий извлечь элемент из вершины стека;
6) свойство Empty, позволяющее определить, является ли стек пустым, или нет.

Пример класса:

```
using System;
namespace Example
{
   public class Stack
   {
      //вложенный класс, реализующий элемент стека
      private class Node
      {
         private object inf;
         private Node next;
         public Node(object nodeInfo)
         {
            inf = nodeInfo;
            next = null;
         }
         public Node Next
         {
            get { return next; }
            set { next = value; }
         }
         public object Inf
         {
            get { return inf; }
            set { inf = value; }
         }
      }       //конец класса Node
      private Node head; //ссылка на вершину стека
      //конструктор класса, создает пустой стек
      public Stack()
      {
         head = null;
      }
      // добавляет элемент в вершину стека
      public void Push(object nodeInfo)
      {
         Node r = new Node(nodeInfo);
         r.Next = head;
         head = r;
      }
      //извлекает элемент из вершины стека, если он не пуст
      public object Pop()
      {
```

EPAM Systems

```
            if (head == null)
            {
               throw new Exception("Стек пуст");
            }
            else
            {
               Node r = head;
               head = r.Next;
               return r.Inf;
            }
         }
         //определяет пуст или нет стек
         public bool IsEmpty
         {
            get
            {
               if (head == null)
               {
                  return true;
               }
               else
               {
                  return false;
               }
            }
         }
      }
   }
```

***Задание***

*Объясните, почему для класса Stack был создан вложенный класс Node и почему класс Node объявлен с помощью спецификатора private.*

Более подробно рассмотрим методы Push и Pop.

***Метод Push*** реализует базовую операцию работы со стеком – добавление нового элемента в вершину стека.

```
public void Push(object nodeInfo)
{
   Node r = new Node(nodeInfo);  //1
   r.Next = head;                //2
   head = r;                     //3
}
```

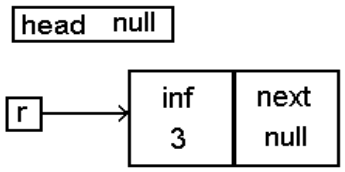
Проиллюстрируем работу метода Push на рисунках. Первоначально считаем, что описана и инициализирована переменная типа Stack:

```
Stack list = new Stack();
```

и мы добавляем в нее элемент со значением 3 следующей командой: list.Push(3).

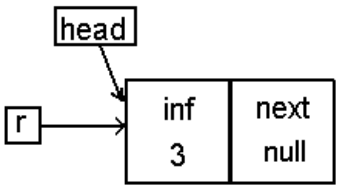
EPAM Systems

*Строка 1.* Создается базовый элемент стека (объект типа Node), значение информационного поля которого равно 3, а ссылочного null.




*Строка 2.* Ссылочное поле r.next переадресуется на верхний элемента стека. Так как стек пока пуст, то ссылочное поле r.next ссылается на значение null.

*Строка 3.* В head записывается адрес области памяти, связанный со ссылкой r. Теперь верхним элементом в стеке становится элемент, связанный со ссылкой r.




После выполнения команды list.Push(3) стек будет выглядеть следующим образом:

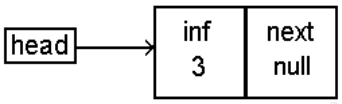
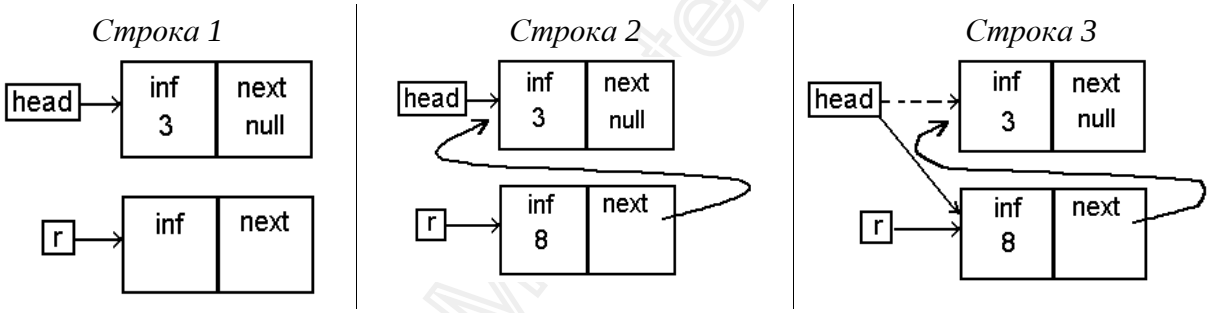



**Рис. 13. Стек с одним элементом.**

Командой list.Push(8) добавим в стек новый элемент со значением 8.




После выполнения команды list.Push(8) стек будет выглядеть следующим образом:

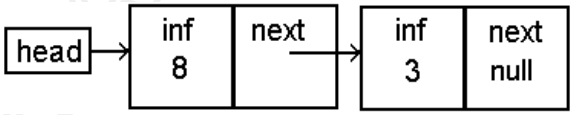



**Рис. 14. Стек с двумя элементами.**

***Замечание***

*Как видим, элемент со значением 8 является верхним элементом стека, хотя был добавлен в стек последним.*

***Задание***

*Добавьте в стек элемент со значением 10 и графически изобразите ход выполнения метода Push.*

***Метод Pop*** реализует другую базовую операцию – извлечение верхнего элемента из стека.

```
public object Pop()
{
   if (head == null)
   {
      throw new Exception("Стек пуст");
```

```
        }
        else
        {
           Node r = head;   //1
           head = r.Next;   //2
           return r.Inf;    //3
        }
     }
```

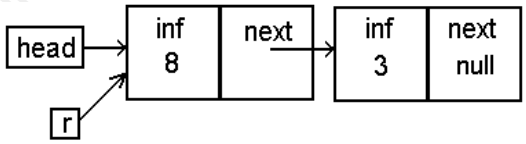
Очевидно, что извлечение элемента из стека возможно только тогда, когда стек не пуст. Поэтому, если мы обратимся к методу Pop для пустого стека, то будет сгенерировано исключение.

Следует обратить внимание на то, что класс Stack предназначен для хранения данных любого типа, т.к. поле inf описано типом object. Поэтому, при извлечении элемента из стека потребуется выполнять явное преобразование типа. В нашем случае в стеке хранились целые числа. Поэтому извлечь верхний элемент можно следующим образом:

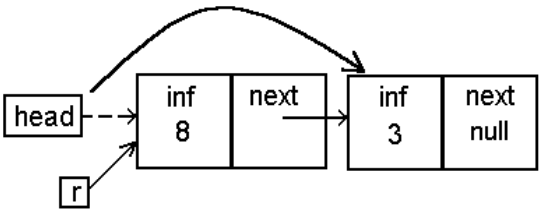
```
int item = (int)list.Pop();
```

Проиллюстрируем работу метода Pop на рисунках. Первоначально считаем, что структура стека соответствует Рис. 14, и мы выполняем команду list.Pop().

*Строка 1.* Устанавливаем ссылку r на верхний элемент стека.




*Строка 2.* Ссылку head переадресуем на элемент, следующий за верхним элементом стека. Таким образом, элемент со значением 8 исключается из стека, а верхним элементом стека становится элемент со значением 3.




*Строка 3.* Метод возвращает значение информационного поля элемента r, т.е., значение 8.

После выполнения команды list.Pop() стек будет соответствовать Рис. 13.

***Задание***

*Извлеките из стека еще один элемент и графически изобразите ход выполнения метода Pop.*

***Замечание***

*Операция проверки стека на пустоту, реализуемая свойством IsEmpty, не определена в рамках АДТ «стек», но будет нам очень полезна при решении практических задач. Заметим, что данная операция не противоречит принципам работы со стеком, т.к. элементы не извлекаются и не добавляются в стек.*

Теперь рассмотрим, как можно создать стек и вывести на экран его содержимое:

```
static void Main()
{
   Stack list = new Stack();      //инициализируем стек
```

```
        // добавляем в стек элементы из диапазона от 0 до 10
        for(int i = 0; i <= 10; i++)
        {
            list.Push(i);
        }
        // пока стек не пуст, извлекаем из него элементы и выводим
        // их на экран
        while (!list.IsEmpty)
        {
            Console.WriteLine(list.Pop());
        }
        Console.WriteLine();
        //пытаемся извлечь элемент из пустого стека
        try
        {
            Console.Write("{0} ",list.Pop());
        }
        catch (Exception error)
        {
            Console.WriteLine("Error: {0}!!!", error.Message);
        }
    }
```

Результат работы программы:

```
10 9 8 7 6 5 4 3 2 1 0
Error: Стек пуст!!!
```

Мы использовали стек для хранения целых чисел. В общем случае класс Stack позволяет организовывать хранение данных любого типа. Единственное, что не нужно забывать – это выполнять преобразование типа при извлечении данных их стека. Например:

```
    static void Main()
    {
        string line = "Hello!!!";
        Stack list = new Stack();
        foreach (char i in line)//посимвольно записываем строку в стек
        {
            list.Push(i);
        }
        //пока стек не пуст, извлекаем из него символы
        //и приписываем к строке
        while (!list.IsEmpty)
        {
            //при извлечении данных выполняем преобразование типов
            char i = (char)list.Pop();
            line += i;
        }
        Console.WriteLine(line);
    }
```

Результат работы программы:

```
Hello!!!!!!olleH
```

EDAM Systems

**Решение практических задач с использованием стеков**

***Пример 1***

Дан файл input.txt, компонентами которого являются символы. Переписать файл input.txt в файл output.txt в обратном порядке.

***Замечание***

*Свойство стека «первый зашел – последний вышел» позволит переписать файл input.txt в файл output.txt в обратном порядке. Здесь и далее подразумевается, что к проекту подключен класс Stack.*

```
using System;
using System.IO;

namespace Example
{
   public class Program
   {
      static void Main()
      {
         Stack list = new Stack();     //инициализируем стек
         //читаем данные из файла
         using (StreamReader fileIn =
                   new StreamReader("d:/Example/input.txt"))
         {
            string line = fileIn.ReadToEnd();
            for (int i = 0; i < line.Length; i++)
            {
               list.Push(line[i]);//каждый символ помещаем в стек
            }
         }
         //записываем данные в файл
         using (StreamWriter fileOut =
                   new StreamWriter("d:/Example/output.txt"))
         {
            char ch;
            while (!list.IsEmpty) //пока стек не пуст
            {
               //извлекаем верхний элемент стека и
               ch = (char)list.Pop();
               fileOut.Write(ch); //записываем его в файл
            }
         }
      }
   }
}
```

Результат работы программы:

| **input.txt** | **output.txt** |
|---|---|
| Hello!!! | !!!olleH |

***Задание***

*Измените программу так, чтобы в стек записывались только знаки пунктуации.*

***Пример 2***

Дана последовательность символов. Перед каждым символом x вставить символ y.

***Замечание***

*В файле input.txt в первой строке через пробел содержатся символы x и y, далее, с новой строки, перечисляется заданная последовательность символов. Выходная последовательность символов записывается в файл output.txt.*

```
using System;
using System.IO;

namespace Example
{
   public class Program
   {
      static void Main()
      {
         Stack listOne = new Stack();
         char x, y;
         //читаем данные из файла
         using (StreamReader fileIn =
                  new StreamReader("d:/Example/input.txt"))
         {
            //первые два символа записываем соответственно в x и y
            string line = fileIn.ReadLine();
            string[] mas = line.Split(' ');
            x = char.Parse(mas[0]);
            y = char.Parse(mas[1]);
            //заданную последовательность записываем в стек
            int ch;
            while ((ch = fileIn.Read()) != -1)
            {
               listOne.Push(Convert.ToChar(ch));
            }
         }
         Stack listTwo = new Stack();  //создаем второй стек
         //переписываем из первого стека во второй все символы
         while (!listOne.IsEmpty)
         {
            char ch = (char)listOne.Pop();
            listTwo.Push(ch);
            //помещаем во второй стек после символа x символ y
            if (ch == x)
            {
               listTwo.Push(y);
            }
         }
         //выводим из второго стека данные в файл
         using (StreamWriter fileOut =
                  new StreamWriter("d:/Example/output.txt"))
         {
            while (!listTwo.IsEmpty)
            {
```

```
                char ch = (char)listTwo.Pop();
                fileOut.Write(ch);
            }
        }
    }
}
```

Результат работы программы:

**input.txt**
```
! ?
Hello!!!
```

**output.txt**
```
Hello?!?!?!
```

***Задание 1***

*Объясните, почему:*

a) *для решения задачи использовалось два стека;*
b) *в отличие от предыдущего примера данные в файле output.txt записаны в прямом порядке.*

***Задание 2***

*Измените программу так, чтобы в заданной последовательности символов символ y вставлялся после каждого символа x.*

***Пример 3***

В файле находится текст, в котором имеются скобки (). Определить, соблюден ли баланс скобок в тексте.

```
using System;
using System.IO;

namespace Example
{
   public class Program
   {
      static void Main()
      {
         bool balanced = true;
         using (StreamReader fileIn =
                 new StreamReader("d:/Example/input.txt"))
         {
            Stack list = new Stack();
            string line = fileIn.ReadToEnd();
            Console.WriteLine(line);
            for (int i = 0; i < line.Length; i++)
            {
               // Если очередной символ «открывающаяся скобка», то
               // помещаем его в стек
               if(line[i] == '(')
               {
                  list.Push(line[i]);
               }
               else
               {
```

ERAM Systems

```
                        // Если очередной символ «закрывающаяся скобка», то
                        // в стеке должна находиться
                        // соответствующая ей «открывающаяся скобка». Если
                        // это так, то извлекаем из
                        // стека парную скобку; иначе полагаем, что баланс
                        // скобок нарушен и прерываем
                        //считывание данных из файла
                        if (line[i] == ')')
                        {
                           if (!list.IsEmpty)
                           {
                              list.Pop();
                           }
                           else
                           {
                              balanced = false;
                              break;
                           }
                        }
                     }
                     //выводим сообщение о проверке баланса скобок
                     if (!balanced)
                     {
                        Console.WriteLine("лишняя закрывающаяся скобка");
                     }
                     else
                     {
                        if (!list.IsEmpty)
                        {
                           Console.WriteLine("лишняя открывающаяся скобка");
                        }
                        else
                        {
                           Console.WriteLine("баланс скобок");
                        }
                     }
                  }
               }
            }
         }
```

| № | input.txt | Результат работы программы |
|---|---|---|
| 1 | (5-x*(4/x-(x+3)/(y-2)-y)-9 | лишняя открывающаяся скобка |
| 2 | (5-x)*(4/x-(x+3)/(y-2)-y)-9 | баланс скобок |
| 3 | (5-x)*(4/x-(x+3)/(y-2)-y))-9 | лишняя закрывающаяся скобка |
| 4 | 7+9-x/2+4-b*3 | баланс скобок |

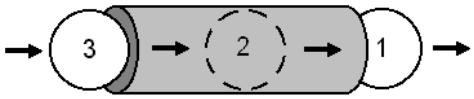
EPAM Systems

***Задание***

*Измените программу так, чтобы она позволяла проверять баланс разных типов скобок, например, (), {}, [].*

**Очередь**

*Очередь* – это частный случай списка, добавление элементов в который выполняется в один конец – хвост (tail), а выборка производится с другого конца – головы (head). Другие операции с очередью не определены. При выборке элемент исключается из очереди. Говорят, что очередь реализует принцип обслуживания FIFO (fist in – fist out, первым пришел – первым вышел). Очередь проще всего представить в виде узкой трубы, в один конец которой бросают мячи, с другого конца которой они вылетают. Понятно, что мяч, который был брошен в трубу первым, первым и вылетит.




Как в повседневной жизи, так в компьютерных задачах, нам часто приходиться ждать своей очереди перед получением доступа к чему-либо. В компьютере могут существовать очереди задач, ожидающих освобождения принтера, доступа к дисковой памяти или получения времени процессора. В рамках одной программы могут существовать множественные запросы, которые приходится сохранять в очереди, или одна задача может породить другие, выполняемые друг за другом и образующие, соответсвенно, очередь задач. Поэтому приложения очередей очень распространены, а сами очереди находят применение во многих прикладных задачах.

Рассмотрим организацию очереди с помощью структуры, изображенной на Рис. 15. Отличие очереди от стека в том, что у очереди две точки доступа: первая (head) ссылается на первый элемент очереди, вторая (tail) – на последний элемент.

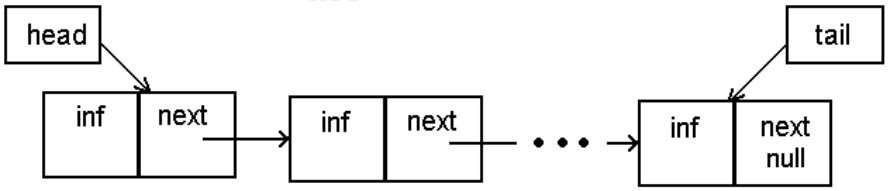



**Рис. 15. Очередь.**

Каждый элемент очереди имеет следующую структуру:

- информационное поле inf, которое может быть любого типа;
- ссылочное поле next, которое будет использоваться для хранения ссылки на следующий элемент очереди, т.е., для организации связи элементов.

Предложенная структура базового элемента очереди соответствует классу Node, рассмотренному в предыдущем разделе. А сам класс очередь (Queue) содержит следующие функциональные элементы:

1) вложенный класс Node для реализации базового элемента очереди;
2) закрытое поле head, хранящее ссылку на первый элемент очереди;
3) закрытое поле tail, хранящее ссылку на последний элемент очереди;
4) конструктор класса;

5) метод Add, позволяющий добавить элемент в конец очереди;
6) метод Take, позволяющий извлечь элемент из начала очереди;
7) свойство IsEmpty, позволяющее определить, является очередь пустой, или нет.

Пример класса:

```
using System;
namespace Example
{
   public class Queue
   {
      //вложенный класс, реализующий базовый элемент очереди
      private class Node
      {
         private object inf;
         private Node next;

         public Node(object nodeInfo)
         {
            inf = nodeInfo;
            next = null;
         }

         public Node Next
         {
            get { return next; }
            set { next = value; }
         }

         public object Inf
         {
            get { return inf; }
            set { inf = value; }
         }
      }       //конец класса Node

      private Node head;
      private Node tail;

      public Queue()
      {
         head = null;
         tail = null;
      }

      public void Add(object nodeInfo)
      {
         Node r = new Node(nodeInfo);
         if (head == null)
         {
            head = r;
            tail = r;
         }
         else
         {
            tail.Next = r;
            tail = r;
```

EPAM Systems

```
            }
         }
         public object Take()
         {
            if (head == null)
            {
               throw new Exception("Очередь пуста.");
            }
            else
            {
               Node r = head;
               head = head.Next;
               if (head == null)
               {
                  tail = null;
               }
               return r.Inf;
            }
         }
         public bool IsEmpty
         {
            get
            {
               if (head == null)
               {
                  return true;
               }
               else
               {
                  return false;
               }
            }
         }
      }
   }
```

Более подробно рассмотрим методы Add и Take.

***Метод Add*** реализует базовую операцию работы с очередью: добавление нового элемента в конец очереди.

```
   public void Add(object nodeInfo)
   {
      Node r = new Node(nodeInfo);  //1
      if (head == null)             //2
      {
         head = r;                  //3
         tail = r;                  //4
      }
      else
      {
         tail.Next = r;             //5
```

EPAM Systems

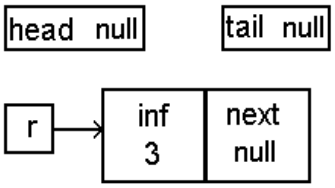
```
            tail = r;                    //6
        }
    }
```

Проиллюстрируем работу метода Add на рисунках. Первоначально считаем, что описана и инициализирована переменная типа Queue:

```
Queue list = new Queue();
```

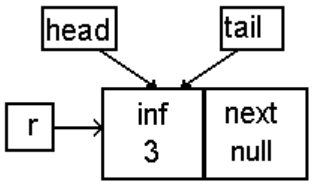
и мы добавляем в очередь элемент со значением 3 следующей командой: list.Add(3).

*Строка 1.* Создается базовый элемент очереди (объект типа Node), значение информационного поля которого равно 3, а ссылочного null.




*Строка 2.* Проверяем, пуста ли очередь? Так как очередь пуста, то выполняются строки 3-4, а строки 5-6 пропускаются.

*Строки 3-4.* Устанавливаем обе ссылки head и tail на базовый элемент, связанный со ссылкой r. Элемент r становится единственным элементом в очереди.




После выполнения команды list.Add(3) очередь будет выглядеть следующим образом:

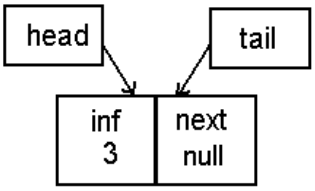
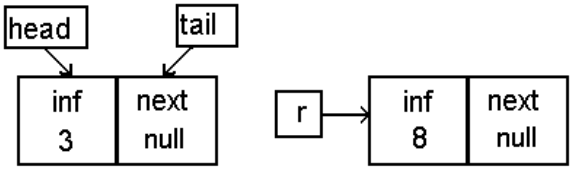



**Рис. 16. Очередь с одним элементом.**

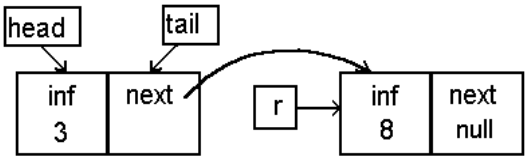
Командой list.Add(8) добавим еще один элемент в очередь.

*Строка 1.* Создается базовый элемент очереди (объект типа Node), значение информационного поля которого равно 8, а ссылочного null.

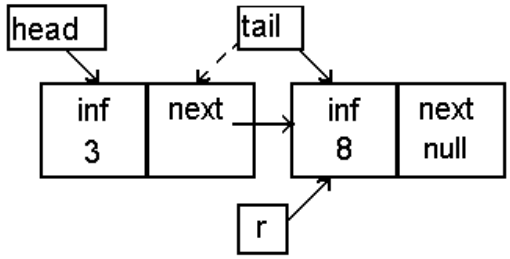



*Строка 2.* Проверяем, пуста ли очередь? Очередь не пуста, в ней есть один элемент, поэтому строки 3-4 пропускаются, выполняются строки 5-6.

*Строка 5.* Ссылка tail.next переадресуется на элемент, связанный с r, т.е., в конец очереди добавляется еще один элемент.



*Строка 6.* Ссылка tail переадресуется на элемент, связанный с r, т.е., на последний элемент очереди.




После выполнения команды list.Add(8) очередь будет выглядеть следующим образом:

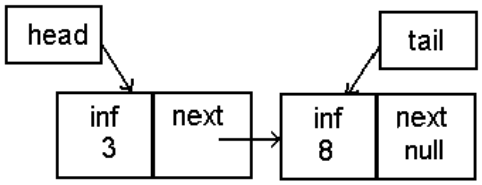



**Рис. 17. Очередь с двумя элементами.**

***Замечание***

*Как видим, элемент со значением 3 был добавлен в очередь первым и стоит первым, а элемент со значением 8 был добавлен последним и стоит в очереди последним.*

***Задание***

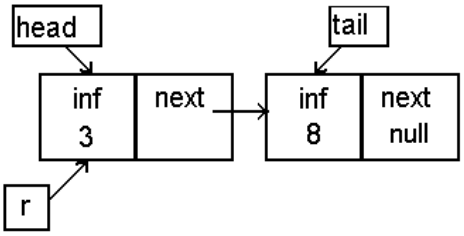
*Добавьте в очередь еще один элемент со значением 10 и графически изобразите ход выполнения метода Add.*

***Метод Take*** реализует другую базовую операцию – извлечение первого элемента из очереди.

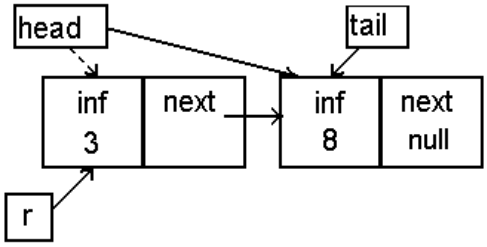
```
public object Take()
{
   if (head == null)
   {
      throw new Exception("Очередь пуста.");
   }
   else
   {
      Node r = head;              //1
      head = head.Next;           //2
      if (head == null)           //3
      {
         tail = null;             //4
      }
      return r.Inf;               //5
   }
}
```

Проиллюстрируем работу метода Take. Первоначально считаем, что структура очереди соответствует Рис. 17, и мы выполняем команду list.Take().

*Строка 1.* Устанавливаем ссылку r на первый элемент очереди.




*Строка 2.* Ссылка head переадресуется на элемент очереди, следующий за первым. Таким образом, элемент со значением 3 исключается из очереди и первым в ней становится элемент со значением 8.

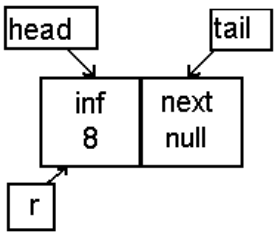



*Строка 3.* Проверяем, пуста ли очередь? Так как очередь не пуста, то строка 4 пропускается.

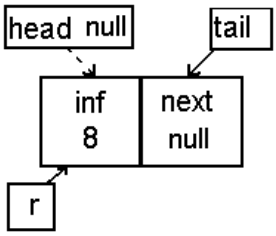
*Строка 5.* Метод возвращает значение информационного поля, связанного со ссылкой r, т.е., значение 3.

После выполнения команды list.Take() структура очереди будет соответствовать Рис. 16. Еще раз выполним команду list.Take().

*Строка 1.* Устанавливаем ссылку r на первый элемент очереди.

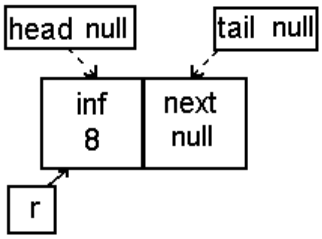



*Строка 3.* Ссылка head переадресуется на элемент очереди, следующий за первым. Таким образом, элемент со значением 8 исключается из очереди. Так как в очереди был только один элемент, то head принимает значение nil




*Строка 3.* Проверяем, пуста ли очередь? Так как очередь пуста, то выполняется строка 4.

*Строка 4.* Ссылке tail присваивается значение nil.




*Строка 5*: метод возвращает значение информационного поля, связанного со ссылкой r, т.е., значение 8.

***Задание.***

*Объясните, что произойдет если для пустой очереди выполнить метод Take().*

EPAM Systems

Теперь рассмотрим, как можно создать очередь и вывести на экран ее содержимое:

```
static void Main()
{
   Queue list = new Queue();     //инициализируем очередь
   //добавляем в очередь целые числа из диапазона от 0 до 10
   for(int i = 0; i <= 10; i++)
   {
      list.Add(i);
   }
   //пока очередь не пуста, извлекаем из нее очередной элемент
   //и выводим его на экран
   while (!list.IsEmpty)
   {
      Console.Write("{0} ",list.Take());
   }
   Console.WriteLine();
   //пытаемся извлечь данные из пустой очереди
   try
   {
      Console.WriteLine(list.Take());
   }
   catch (Exception error)
   {
      Console.WriteLine("Ошибка: {0}", error.Message);
   }
}
```

Результат работы программы:

```
0 1 2 3 4 5 6 7 8 9 10
Ошибка: Очередь пуста.
```

Так же как и стек, очередь может использоваться для хранения данных любого типа. И конечно, при извлечении данных из очереди нужно не забывать по преобразование типов.

***Замечание***

*В приведенном выше примере преобразование не проводилось, так как извлеченные данные сразу передавались в метод WriteLine, автоматически выполняющий приведение типов. В случае, когда речь идет о типах, для которых правил преобразования не существует, или для извлеченного объекта необходимо выполнить какие-либо действия, преобразование по-прежнему остается необходимым.*

```
using System;
namespace Example
{
   public class Program
   {
      public struct Point
      {
         public int x;
         public int y;
         public Point (int x, int y)
         {
```

EPAM Systems

```
            this.x = x;
            this.y = y;
         }
         public void Show ()
         {
            Console.WriteLine("({0}, {1})", x,y);
         }
      }
      static void Main()
      {
         //инициализируем очередь и добавляем в нее «точки»
         Queue list = new Queue();
         list.Add(new Point(0,0));
         list.Add(new Point(1, 2));
         list.Add(new Point(-4, 0));
         // пока очередь не пуста, извлекаем из нее очередной
         // элемент и выводим его на экран
         while (!list.IsEmpty)
         {
            Point item = (Point)list.Take(); //преобразование данных
            item.Show();
         }
      }
   }
}
```

Результата работы программы:

```
(0, 0)
(1, 2)
(-4,0)
```

**Решение практических задач с использованием очередей**

***Пример 1***

Дана последовательность символов. Каждый символ равный x, заменить символом y. Входная последовательность символов хранится в файле data.txt. Имя файла и значения x и y вводятся с клавиатуры. Выходная последовательность символов записывается в тот же файл.

```
using System;
using System.IO;

namespace Example
{
   public class Program
   {
      static void Main()
      {
         Queue list = new Queue();
         Console.Write("Введите полное имя файла: ");
         string fileName = Console.ReadLine();
         Console.Write("x: ");
         char x = char.Parse(Console.ReadLine());
         Console.Write("y: ");
```

ERAM Systems

```
            char y = char.Parse(Console.ReadLine());
            using (StreamReader fileIn = new StreamReader(fileName))
            {
               string line = fileIn.ReadToEnd();
               for (int i = 0; i < line.Length; i++)
               {
                  if (line[i] == x)
                  {
                     list.Add(y);
                  }
                  else
                  {
                     list.Add(line[i]);
                  }
               }
            }
            using (StreamWriter fileOut = new StreamWriter(fileName))
            {
               char ch;
               while (!list.IsEmpty)
               {
                  ch = (char)list.Take();
                  fileOut.Write(ch);
               }
            }
         }
      }
   }
```

Результат работы программы

```
Входные данные
Имя файла: d:/Example/data.txt
X: 7
Y: 0
Исходное содержимое файла
7 7 1 3 7 5 2 5 7 2 7 9 3 7 7
Итоговое содержимое файла
0 0 1 3 0 5 2 5 0 2 0 9 3 0 0
```

***Задание***

*Измените программу так, чтобы в заданном файле каждый символ x удваивался.*

***Пример 2***

Написать рекурсивный метод, сортирующий по возрастанию последовательность целых чисел. Последовательность целых чисел хранится в файле input.txt. Отсортированную последовательность записать в этот же файл.

```
using System;
using System.IO;
namespace Example
{
   public class Program
   {
```

ERAM Systems

```
// Рекурсивный метод, производящий сортировку
// очереди по возрастанию
static void Sort(Queue list)
{
   //инициализируем вспомогательные очереди
   Queue one = new Queue();
   Queue two = new Queue();
   if (!list.IsEmpty) // если исходная очередь не пуста
   {
      // извлекаем из нее первый элемент
      int a = (int)list.Take();
      // элементы исходной очереди
      // меньшие a записываем в первую
      // вспомогательную очередь, большие или равные
      // a - во вторую очередь
      while (!list.IsEmpty)
      {
         int b = (int)list.Take();
         if (a > b)
         {
            one.Add(b);
         }
         else
         {
            two.Add(b);
         }
      }
      //сортируем вспомогательные очереди
      Sort(one);
      Sort(two);
      //в исходную очередь записываем
      //вначале элементы 1-ой вспомогательной
      //очереди, затем элемент a, затем элементы
      //2-ой вспомогательной очереди}
      while (!one.IsEmpty)
      {
         list.Add(one.Take());
      }
      list.Add(a);
      while (!two.IsEmpty)
      {
         list.Add(two.Take());
      }
   }
}
static void Main()
{
   Queue list = new Queue();
   int ch;
   Console.Write("Введите полное имя файла: ");
   string fileName = Console.ReadLine();
   using (StreamReader fileIn = new StreamReader(fileName))
```

ЕРАМ Systems

```
            {
               string line = fileIn.ReadToEnd();
               string[] data = line.Split(' ');
               foreach (string item in data)
               {
                  list.Add(int.Parse(item));
               }
            }
            Sort(list);
            using (StreamWriter fileOut = new StreamWriter(fileName))
            {
               while (!list.IsEmpty)
               {
                  ch = (int)list.Take();
                  fileOut.Write("{0} ",ch);
               }
            }
         }
      }
   }
```

Результат работы программы

```
Исходные данные файла
1 6 7 56 4 12 0 7 0 8 7 0 -9 -4 8
Отсортированные данные
-9 -4 0 0 0 1 4 6 7 7 7 8 8 12 56
```

***Задание***

*Измените программу так, чтобы сортировка очереди выполнялась по убыванию значения элементов.*

**Однонаправленные списки общего вида**

Мы рассмотрели частные случаи линейного списка – стек и очередь, а также их реализацию с помощью ссылок. Заметим, что стек имеет одну «точку доступа», очередь – две «точки доступа».

Сам список можно представить в виде линейной связанной структуры с произвольным количеством «точек доступа» (см. Рис. 18 и Рис. 19), что позволяет определить следующие операции над списком: инициализация списка, добавление и удаление элемента из произвольной позиции, поиск элемента в списке по ключу, просмотр всех элементов без их извлечения списка и ряд других.

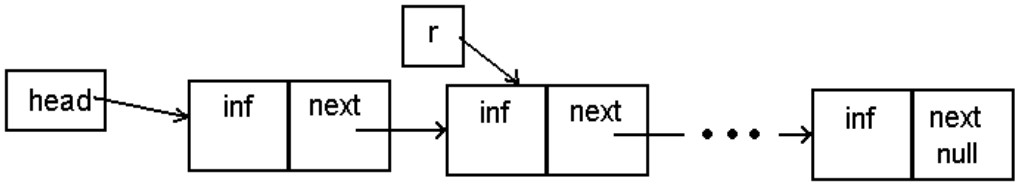



**Рис. 18. Структура однонаправленного списка со ссылками: head на начало списка, r – на произвольный элемент списка**

ERAM Systems

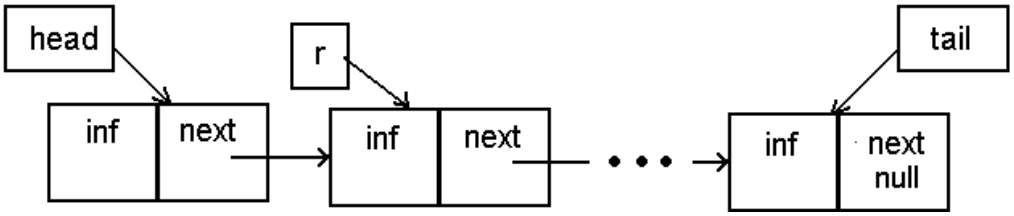



**Рис. 19. Структура однонаправленного списка со ссылками: head на начало списка, tail – на конец списка, r – на произвольный элемент списка**

Вид списка определяет набор допустимых операций с ним, поэтому выбор вида списка зависит от решаемой задачи и предпочтений программиста.

Рассмотрим некоторые из возможных операций для списка общего вида. Каждый элемент такого списка имеет структуру аналогичную структуре, описанной для стека и очереди, и может быть реализован классом Node. Однако однонаправленный список общего вида имеет произвольное количество точек доступа и поддерживает большее количество операций, в том числе, допускающих изменение содержимого списка без исключения из него элементов, или наоборот, включения в него новых элементов. Поэтому класс Node следует реализовать как самостоятельный класс, описанный спецификатором public.

***Замечание***

*Напомним, что ранее класс Node был внутренним классом и описывался спецификатором private.*

Класс «однонаправленный список» (List) будет содержать следующие функциональные элементы:

1) вложенный класс Node для реализации базового элемента списка;
2) закрытое поле head, хранящее ссылку на первый элемент списка;
3) закрытое поле tail, хранящее ссылку на последний элемент списка;
4) конструктор класса;
5) метод AddBegin, позволяющий добавить элемент в начало списка (модификация добавления элемента в стек);
6) метод AddEnd, позволяющий добавить элемент в конец списка (модификация добавления элемента в очередь);
7) метод TakeBegin, позволяющий извлечь элемент из начала списка (модификация извлечения элемента из очереди);
8) метод TakeEnd, позволяющий извлечь элемент с конца списка;
9) свойство Empty, позволяющее определить, является список пустым, или нет.
10) свойство Find, позволяющий осуществлять поиск элемента в списке по ключу;
11) метод Insert, позволяющий вставить элемент с указанным значением после ключевого элемента списка;
12) метод Delete, позволяющий удалить ключевой элемент из списка;
13) метод Show, позволяющий вывести содержимое списка на экран.

Рассмотрим реализацию данного класса:

```
using System;
namespace Example
{
   public class List
   {
      private Node head;
```

```
private Node tail;
public List()
{
   head = null;
   tail = null;
}
public void AddBegin(object nodeInfo)
{
   Node r = new Node(nodeInfo);
   if (head == null)
   {
      head = r;
      tail = r;
   }
   else
   {
      r.Next = head;
      head = r;
   }
}
public void AddEnd(object nodeInfo)
{
   Node r = new Node(nodeInfo);
   if (head == null)
   {
      head = r;
      tail = r;
   }
   else
   {
      tail.Next = r;
      tail = r;
   }
}
public object TakeBegin()
{
   if (head == null)
   {
      throw new Exception("Список пуст");
   }
   else
   {
      Node r = head;
      head = head.Next;
      if (head == null)
      {
         tail = null;
      }
      return r.Inf;
   }
}
```

EPAM Systems

```
public object TakeEnd()
{
   if (head == null)
   {
      throw new Exception("Список пуст");
   }
   else
   {
      Node r = head;
      // если элемент в списке единственный, то
      if (head.Next==null)
      {
         head = null;       // список «обнуляется»
         tail = null;
      }
      else
      {
         // в противном случае мы перемещаемся по ссылкам
         // до предпоследнего элемента в
         // списке и исключаем его из списка
         while (r.Next != tail)
         {
            r = r.Next;
         }
         Node temp = tail;
         tail = r;
         r = temp;
         tail.Next = null;
      }
      return r.Inf;
   }
}
public bool IsEmpty
{
   get
   {
      if (head == null)
      {
         return true;
      }
      else
      {
         return false;
      }
   }
}
public Node Find(object key)
{
   Node r = head;
   while (r != null)
   {
      if (((IComparable)(r.Inf)).CompareTo(key) == 0)
```

EPAM Systems

```
            {
               break;
            }
            else
            {
               r = r.Next;
            }
         }
         return r;
      }
      public void Insert (object key, object item)
      {
         Node r = Find(key);
         if (r != null)
         {
            Node p = new Node(item);
            p.Next = r.Next;
            r.Next = p;
         }
      }
      public void Delete (object key)
      {
         if (head == null)
         {
            throw new Exception("Список пуст");
         }
         else
         {
            if (((IComparable)(head.Inf)).CompareTo(key) == 0)
            {
               head = head.Next;
            }
            else
            {
               Node r = head;
               while (r.Next != null)
               {
                  if (((IComparable)(r.Next.Inf)).CompareTo(key)==0)
                  {
                     r.Next = r.Next.Next;
                     break;
                  }
                  else
                  {
                     r = r.Next;
                  }
               }
            }
         }
      }
```

EPAM Systems

```
        public void Show()
        {
           Node r = head;  //устанавливаем ссылку на начало списка
           while(r != null)//пока не достигли конца списка
           {
              //выводим на экран содержимое информационного поля
              Console.Write("{0} ", r.Inf);
              //перемещаем ссылку на следующий элемент списка
              r = r.Next;
           }
           Console.WriteLine();
        }
     }
  }
```

Более подробно рассмотрим работу таких методов как Find, Insert и Delete.

***Метод Find*** реализует классический алгоритм линейного поиска.

```
public Node Find(object key)
{
   Node r = head;                                          //1
   while (r != null)                                       //2
   {
      if (((IComparable)(r.Inf)).CompareTo(key) == 0)      //3
      {
         break;                                            //4
      }
      else
      {
         r = r.Next;                                       //5
      }
   }
   return r;                                               //6
}
```

Проиллюстрируем работу данного метода на рисунках. Первоначально считаем, что структура списка соответствует Рис. 20 и мы выполняем команду list.Find(3).

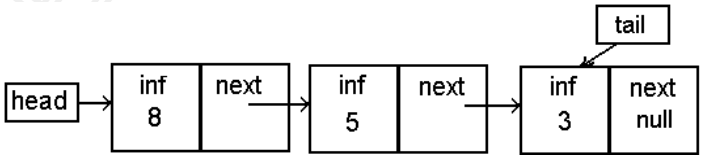
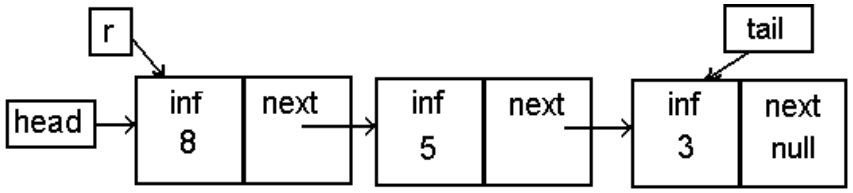



**Рис. 20. Список из трех элементов.**

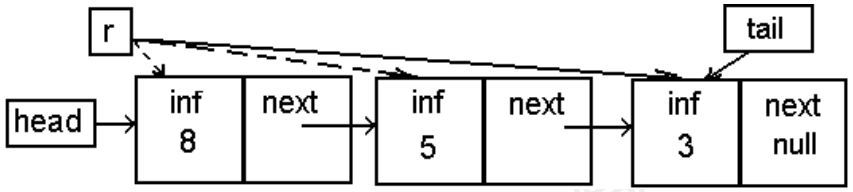
*Строка 1.* Устанавливаем ссылку r на начало списка.



*Строка 2.* Организуем цикл, в котором просматриваем все элементы списка. Если информационное поле некоторого элемента списка будет равно ключу (*строка 3*), то поиск завершится *(строка 4*); иначе осуществляется переход по ссылке к следующему элементу списка *(строка 5).*

Обратите внимание на то, как выполняется сравнение значений в строке 3. Значения key и r.Inf объявлены типом object, т.е., ссылочным типом. Поэтому при сравнении этих величин будут сравниваться ссылки, а ссылки равны только тогда, когда они ссылаются на один и тот же объект. Поэтому мы предполагаем, что объект списка реализует стандартный интерфейс IComparable.

В результате циклического выполнения строк 2-5 ссылка r будет установлена на элемент со значением 3.




*Строка 6.* Метод возвращает ссылку r на искомый элемент.

Таким образом, после выполнения команды list.Find(3), указатель item будет ссылаться на элемент списка, в информационное поле которого записано значение 3. Значение этого информационного поля можно изменить с помощью команды присваивания, например:

```
item.Inf = 10
```

***Задание***

*Объясните, какое значение примет ссылка item, если выполнить команду*

```
Node item = list.Find(30)
```

***Метод Insert*** позволяет вставить новый элемент после ключевого. Реализация этого метода основывается на использовании метода Find.

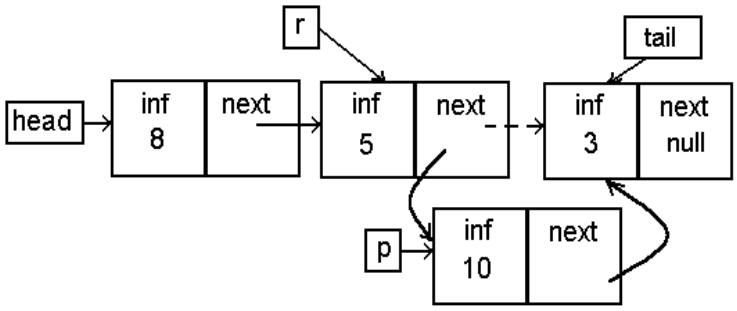
```
public void Insert (object key, object item)
{
   Node r = Find(key);           //1
   if (r != null)                //2
   {
      Node p = new Node(item);   //3
      p.Next = r.Next;           //4
      r.Next = p;                //5
   }
}
```

Пусть структура списка соответствует Рис. 20 и мы выполняем команду list.Insert(5, 10).

*Строка 1.* Ссылка r устанавливается на искомый элемент списка. Если элемент с заданным значением не найден, ссылка r получает значение null.

*Строка 2.* Если ссылка r не ссылается на null, то мы нашли ключевой элемент и можем выполнять вставку (строки 3-5).

Схематично выполнение строк 7-9 можно изобразить следующим образом:




Таким образом, после выполнения команды list.Insert(5, 10) список будет выглядеть следующим образом:

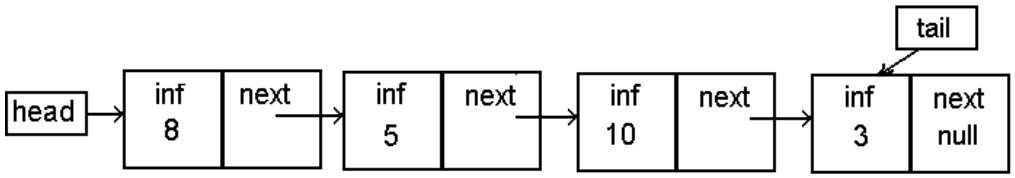



**Рис. 21. Список из четырех элементов.**

***Задание***

*Объясните, как изменится структура списка, изображенного на Рис. 21, если:*

a) *выполнить команду list.Insert(20, 10);*
b) *последовательно выполнить две команды list.AddBegin(3); list.Insert(3, 0).*

***Метод Delete*** позволяет удалить ключевой элемент списка. Реализация этого метода основывается на модификации алгоритма линейного поиска: если искомый элемент не первый, то нас интересует ссылка не на сам элемент, а на элемент, *предшествующий* тому, который мы хотим удалить.

```
public void Delete (object key)
{
   if (head == null)
   {
      throw new Exception("Список пуст");
   }
   else
   {
      //если первый элемент ключевой, то
      if (((IComparable)(head.Inf)).CompareTo(key) == 0)
      {
         head = head.Next;  //исключаем его из списка
      }
      else
      {
         //иначе, устанавливаем ссылку на первый элемент
         //и осуществляем поиск элемента,
         //предшествующего ключевому
```

```
            Node r = head;
            while (r.Next != null)
            {
               if (((IComparable)(r.Next.Inf)).CompareTo(key) == 0)
               {
                  //найденный элемент исключаем из списка см. Рис. 22
                  r.Next = r.Next.Next;
                  break;
               }
               else
               {
                  r = r.Next;
               }
            }
         }
      }
   }
```

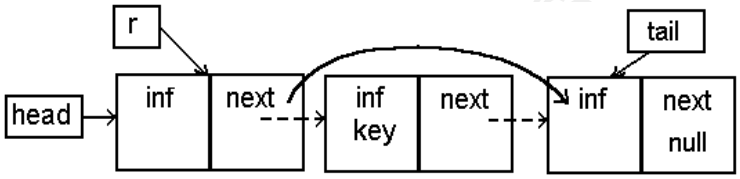



**Рис. 22. Удаление ключевого элемента из списка.**

***Задания***

1. *При удалении ключевого элемента из списка, указатель r устанавливался не на сам ключевой элемент, а на элемент, предыдущий по отношению к ключевому. Объясните почему.*
2. *Метод Delete находит первое вхождение элемента с ключевым значением в списке и удаляет его. Измените метод так, чтобы из списка удалялись все элементы с ключевым значением.*

Рассмотрим использование созданного класса на примере.

```
using System;
using System.IO;
namespace Example
{
   public class Program
   {
      static void Main()
      {
         List list = new List();
         Console.WriteLine("Создали список: ");
         for (int i = 0; i < 10; i++)
         {
            if (i % 2 == 0)
            {
               list.AddBegin(i);
            }
```

```
            else
            {
               list.AddEnd(i);
            }
         }
         list.Show();
         Console.WriteLine("Произвели замену в списке:");
         Node item = list.Find(0);
         item.Inf = 100;
         list.Show();
         Console.WriteLine("Произвели вставку элемента в списке:");
         list.Insert(100,-1);
         list.Show();
         Console.WriteLine("Произвели удаление элемента в списке:");
         list.Delete(100);
         list.Show();
         Console.WriteLine("Извлекли значение {0} и {1} из списка:",
                           list.TakeBegin(), list.TakeEnd());
         list.Show();
      }
   }
}
```

Результат работы программы:

```
Создали список:
8 6 4 2 0 1 3 5 7 9
Произвели замену в списке:
8 6 4 2 100 1 3 5 7 9
Произвели вставку элемента в списке:
8 6 4 2 100 -1 1 3 5 7 9
Произвели удаление элемента в списке:
8 6 4 2 -1 1 3 5 7 9
Извлекли значение 8 и 9 из списка:
6 4 2 -1 1 3 5 7
```

***Замечание***

*Мы рассмотрели лишь несколько функциональных членов класса List. В общем случае их может быть гораздо больше.*

**Решение практических задач с использованием однонаправленных списков общего вида**

***Пример 1***

Дан входной файл, компонентами которого являются целые числа. На основе данных файла создать однонаправленный список. Заменить в списке все элементы со значением X на элементы со значением Y. Вывести на экран исходный и измененный список.

```
using System;
using System.IO;
namespace Example
{
```

EPAM Systems

```
public class Program
{
   static void Main()
   {
      List list = new List();  //инициализируем список
      //считываем данные из файла в список
      using (StreamReader fileIn =
              new StreamReader("d:/Example/input.txt"))
      {
         string line = fileIn.ReadToEnd();
         string[] data = line.Split(' ');
         foreach (string item in data)
         {
            list.AddEnd(int.Parse(item));
         }
      }
      Console.Write("x=");
      int x = int.Parse(Console.ReadLine());
      Console.Write("y=");
      int y = int.Parse(Console.ReadLine());
      list.Show();  //выводим данные из списка на экран
      Node r;
      //пока есть ключевой элемент в списке
      while ((r = list.Find(x)) != null)
      {
         r.Inf = y; //заменяем его
      }
      //выводим измененные  данные из списка на экран
      list.Show();
   }
}
```

Результат работы программы:

```
input.txt
1 3 2 4 5 3 1 2 1
Вывод на экран
1 2 1 3 5 4 2 3 1
0 2 0 3 5 4 2 3 0
```

***Задание***

*Измените программу так, чтобы она подсчитывала, сколько раз элемент со значением X встречается в списке.*

***Пример 2***

Дан входной файл, компонентами которого являются целые числа. На основе данных файла создать однонаправленный список. В списке удвоить все элементы со значением X. Вывести на экран данные исходного и измененного списка.

EPAM Systems

Для решения данной задачи в класс List добавим два метода:

1) метод *public Node Find(Node begin, object key)*, позволяющий производить поиск ключевого элемента не с начала списка, а начиная с элемента, на который установлена ссылка begin.
2) Метод *public void Insert( Node begin, object item)*, позволяющий производить вставку нового элемента не после ключевого значения, а после элемента, на который установлена ссылка begin.

```
public class List
{
   …
   public Node Find(Node begin, object key)
   {
      Node r = begin;
      while (r != null)
      {
         if (((IComparable)(r.Inf)).CompareTo(key) == 0)
         {
            break;
         }
         else
         {
            r = r.Next;
         }
      }
      return r;
   }
   public void Insert(Node begin, object item)
   {
      Node r = begin;
      if (r != null)
      {
         Node p = new Node(item);
         p.Next = r.Next;
         r.Next = p;
      }
   }
}
```

Следует заметить, что метод Find(object key), рассмотренный ранее, является частным случаем метода Find(Node begin, object key). Поэтому мы рекомендуем метод Find(object key) изменить следующим образом:

```
public Node Find(object key)
{
   return Find(head, key);
}
```

Это позволит избежать дублирования кода и даст возможность вносить изменения в поиск только в одной обобщенной версии метода.

А теперь рассмотрим решение задачи.

EPAM Systems

```
using System;
using System.IO;
namespace Example
{
   public class Program
   {
      static void Main()
      {
         List list = new List();
         using (StreamReader fileIn =
                  new StreamReader("d:/Example/input.txt"))
         {
            string line = fileIn.ReadToEnd();
            string[] data = line.Split(' ');
            foreach (string item in data)
            {
               list.AddEnd(int.Parse(item));
            }
         }
         Console.Write("x=");
         int x = int.Parse(Console.ReadLine());
         list.Show();  //выводим на экран исходный список
         //устанавливаем ссылку на первый элемент, равный x
         Node r = list.Find(x);
         while (r != null)       //не просмотрен весь список
         {
            //вставляем после ссылки r элемент со значением x
            list.Insert(r,x);
            //перемещаем ссылку r  на два элемента (текущий
            // и вставленный)
            r = r.Next.Next;
            //осуществляем поиск элемента со значением x относительно
            //ссылки r
            r = list.Find(r,x);
         }
         list.Show();  //выводим на экран измененный список
      }
   }
}
```

Результат работы программы:

```
input.txt
1 1 3 2 4 5 3 1 1 2 1
Вывод на экран
1 1 3 2 4 5 3 1 1 2 1
1 1 1 1 3 2 4 5 3 1 1 1 1 2 1 1
```

***Задание***

*Измените программу так, чтобы каждый элемент со значением X утраивался в списке.*

EFAM Systems

***Пример 3***

Дан текстовый файл. На основе данных файла создать однонаправленный список, включив в него каждое слово файла только один раз. Полученные данные вывести на экран.

```
using System;
using System.IO;

namespace Example
{
   public class Program
   {
      static void Main()
      {
         List list = new List();
         using (StreamReader fileIn =
                     new StreamReader("d:/Example/input.txt"))
         {
            string line = fileIn.ReadToEnd();
            string[] data = line.Split(' ');
            foreach (string item in data)
            {
               //осуществляем поиск значения текущего слова в списке
               Node r = list.Find(item);
               //если поиск показал, что текущего слова в списке нет,
               if (r == null)
               {
                  // то включаем его в список
                  list.AddEnd(item);
               }
            }
         }
         list.Show();
      }
   }
}
```

Результат работы программы:

```
input.txt
to be or not to be
Вывод на экран:
to be or not
```

## Практикум №14

***Задание 1***

***Замечание***

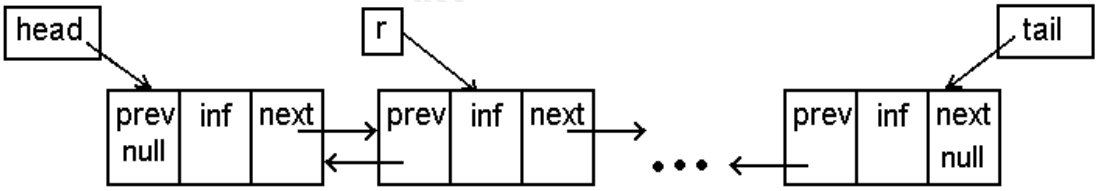
*Каждую задачу данного раздела решить, реализовав список в виде линейно связанной структуры*[*]*: стека, очереди и списка общего вида. Исходный и измененный список вывести на экран.*

[*] По усмотрению преподавателя

1. На основе файла создать список. Удвоить в нем вхождение каждого четного элемента.
2. На основе файла создать список, включив в него только повторяющиеся числа.
3. На основе файла создать список, включив в него только уникальные числа.
4. На основе файла создать список. Перед каждым элементом равным x вставить элемент равный y.
5. На основе файла создать список. Перед каждым элементом равным x удалить один элемент.
6. На основе файла создать список. Заменить каждую последовательность повторяющихся элементов на один элемент.
7. На основе файла создать список. Найти минимальный элемент, и удалить из списка все элементы равные минимальному.
8. На основе файла создать список. Поменять в списке местами первый максимальный и последний минимальный элементы.
9. На основе файла создать список, вычислив среднее арифметическое значение всех элементов. Удалить из списка элементы, значение которых меньше среднего арифметического всех элементов исходного списка.
10. На основе файла создать список. Найти максимальный элемент, и после каждого максимального элемента в списке вставить элемент, значение которого равно среднему арифметическому значению предыдущих элементов.

***Задание 2***

Мы рассмотрели организацию однонаправленных списков. Однако во многих приложениях возникает необходимость организовывать эффективное перемещение по списку как в прямом, так и в обратном направлении. В этой ситуации можно прибегнуть к организации двунаправленного списка, структура которого изображена ниже:




Предложенная структура содержит ссылки на первый, последний и произвольный элементы двунаправленного списка. Разработайте класс, реализующий данную модель организации двунаправленного списка, самостоятельно определив необходимые функциональные члены класса.

С помощью разработанного класса решите следующую задачу: дан файл, компонентами которого являются целые числа. На основе файла создайте двунаправленный список так, чтобы элементы заносились в него в порядке возрастания значений. Выведите на экран содержимое списка в прямом и обратном порядке.

***Задание 3***

Пусть дано математическое выражение, в котором используются лексемы (синтаксически неделимые единицы):

1) целые и действительные числа;
2) математические операции: +, –, *, /;
3) круглые скобки;

4) однобуквенные переменные.

Для программного подсчета значения математического выражения необходимо:

1) разбить данное математическое выражение на лексемы;
2) проверить корректность математической записи;
3) записать выражение в виде обратной польской нотации;
4) по обратной польской нотации подсчитать значение выражения (если в выражении встречаются переменная, то ее значение должно запрашиваться с клавиатуры только один раз).

***Замечание***

*Существуют три способа записи арифметических выражений: инфиксная (привычная для нас – знак математической операции помещается между операндами), префиксная (знак математической операции помещается перед операндами) и постфиксная (знак математической операции помещается после операндов). Постфиксную запись арифметического выражения называют обратной польской записью, или нотацией).*

Рассмотрим алгоритм формирования обратной польской нотации математического выражения. Для его реализации нам потребуется два списка: очередь (основной список) и стек (вспомогательный список). Напомним, что математические операции умножения и деления имеют высший приоритет по отношению к сложению и вычитанию. При формировании обратной польской нотации будем использовать приведенные ниже правила.

1. Если текущая лексема является числом, или переменной, то она помещается в очередь.
2. Если текущая лексема является открывающейся скобкой, то она помещается в стек.
3. Если текущая лексема является математической операцией и стек пуст, или вершиной стека является открывающаяся скобка, то лексема помещается в стек.
4. Если текущая лексема является математической операцией и стек не пуст, причем вершиной стека не является открывающаяся скобка, то:
   a) если вершиной стека является математическая операция одного приоритета с текущей лексемой, то эта операция извлекается из стека и помещается в очередь, а текущая лексема записывается в стек;
   b) если вершиной стека является математическая операция с приоритетом выше текущей лексемы, то все операции до открывающейся скобки извлекаются из стека и записываются в очередь, а текущая операция помещается в стек;
   c) если вершиной стека является математическая операция с приоритетом ниже текущей лексемы, то текущая лексема помещается в стек.
5. Если текущая лексема является закрывающей скобкой, то из стека извлекаются все операции до открывающейся скобки и помещаются в очередь; открывающаяся скобка также извлекается из стека;
6. Если лексемы закончились и стек оказался не пуст, то все операции извлекаются из стека и помещаются в очередь.

Проиллюстрируем правила формирования обратной польской нотации на примере математического выражения: $3+(4*a/7-3.5/(2.1*4))*10-a$.

| № | Текущая лексема | Стек | Очередь |
|---|---|---|---|
| | 3 | | 3 |

| № | Текущая лексема | Стек | Очередь |
|---|---|---|---|
| 1 | + | + | 3 |
| 2 | ( | ( + | 3 |
| 3 | 4 | ( + | 3 4 |
| 4 | * | * ( + | 3 4 |
| 5 | a | * ( + | 3 4 a |
| 6 | / | / ( + | 3 4 a * |
| 7 | 7 | / ( + | 3 4 a * 7 |
| 8 | - | - ( + | 3 4 a * 7 / |
| 9 | 3.5 | - ( + | 3 4 a * 7 / 3.5 |
| 10 | / | / - ( + | 3 4 a * 7 / 3.5 |
| 11 | ( | ( / - ( + | 3 4 a * 7 / 3.5 |
| 12 | 2.1 | ( / - ( + | 3 4 a * 7 / 3.5 2.1 |
| 13 | * | * ( / - ( + | 3 4 a * 7 / 3.5 2.1 |
| 14 | 4 | * ( / - ( + | 3 4 a * 7 / 3.5 2.1 4 |
| 15 | ) | / - ( + | 3 4 a * 7 / 3.5 2.1 4 * |
| 16 | ) | + | 3 4 a * 7 / 3.5 2.1 4 * / - |
| 17 | * | * + | 3 4 a * 7 / 3.5 2.1 4 * / - |
| 18 | 10 | * + | 3 4 a * 7 / 3.5 2.1 4 * / - 10 |
| 19 | - | - | 3 4 a * 7 / 3.5 2.1 4 * / - 10 * + |
| 20 | a | - | 3 4 a * 7 / 3.5 2.1 4 * / - 10 * + a |

Больше лексем нет, поэтому итоговая очередь, содержащая польскую запись математического выражения, будет выглядеть следующим образом:

```
3 4 a * 7 / 3.5 2.1 4 * / - 10 * + a –
```

Чтобы подсчитать значение выражения по обратной польской нотации необходимо последовательно применять операцию к двум аргументам, стоящим слева от операции. Для рассматриваемого выражения порядок выполнения операций будет иметь следующий вид:

```
                1    2              3  4  5     6  7    8
((3 ((((4 a *) 7 /) (3.5 (2.1 4 *) /) -) 10 *) +) a –)
```

Реализуйте рассмотренный алгоритм самостоятельно.

***Замечание 1***

*Разбиение исходного выражения на лексемы можно проводить с помощью стандартных методов для работы со строками, а проверку корректности записи математического выражения осуществлять на этапе формирования польской записи выражения.*

ЕВАМ Systems

***Замечание 2***

*Если выражение начинается с унарного минуса, например, «–7+4/2», или «-a*b», то рекомендуется свести унарный минус к бинарному минусу добавлением незначащего нуля следующим образом: «0-7+4/2» или «0-a*b».*

## Деревья

### Основные понятия

*Деревья* представляют собой иерархическую структуру некой совокупности элементов. Примером деревьев могут служить генеалогические и организационные диаграммы, а также структура каталогов носителя информации. Деревья активно используются для организации информации в системах управления базами данных и для представления синтаксических структур в компиляторах программ.

В качестве примера давайте рассмотрим хранение в памяти компьютера коллекции дисков, структура которой показана на Рис. 23.

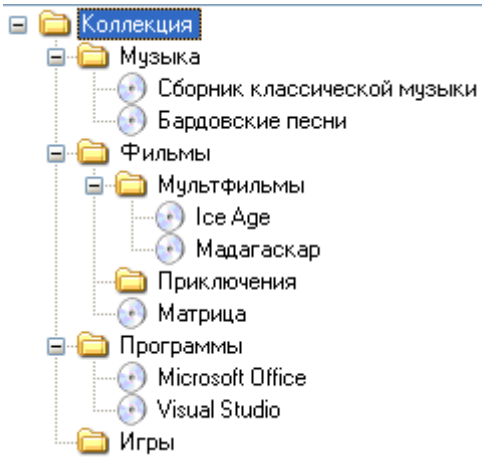



**Рис. 23. Коллекция дисков**

Данная коллекция содержит в себе разнотипные элементы: диски и папки для их хранения, причем любая папка может содержать в себе как диски, так и вложенные папки с дисками, или папками более глубокого уровня вложенности.

Структуру такой коллекции дисков можно представить с помощью АТД *дерево* следующим образом:

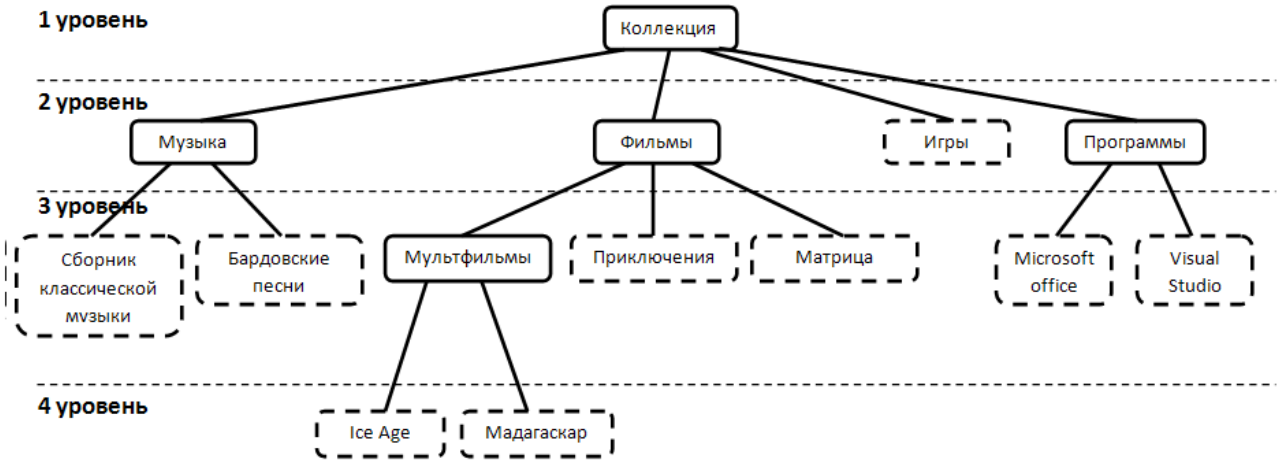



**Рис. 24. Дерево коллекции дисков**

Теперь рассмотрим основные понятия, связанные с АТД дерево.

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

1) имеется один специально обозначенный узел, называемый *корнем* данного дерева;
2) остальные узлы (исключая корень) содержатся в $n \geq 0$ попарно не пересекающихся множествах $T_1, T_2, \ldots, T_n$, каждое из которых в свою очередь является деревом. Деревья $T_1, T_2, \ldots, T_n$ называются *поддеревьями* данного корня.

Это определение является рекурсивным, т.е., мы определили дерево в терминах самих же деревьев. Из данного определения следует, что каждый узел дерева является корнем некоторого поддерева.

Число поддеревьев узла называется его *степенью*. Узел с нулевой степенью называется *листом*.

Каждый корень является *отцом (родителем)* корней своих поддеревьев; последние являются *сыновьями (детьми)* своего отца. Каждый узел (кроме корня) имеет одного родителя и произвольное число сыновей. Корень не имеет родителя, листы не имеют сыновей.

*Уровень* узла по отношению к дереву определяется следующим образом: говорят, что корень имеет уровень 1, а другие узлы имеют на единицу выше их уровня относительно содержащего их поддерева $T_j$ этого корня.

Рассмотрим дерево, представленное на Рис. 25. Оно имеет корень A и листья: H, J, D, G, F. Степени вершин этого дерева следующие: A и B имеют степень 2, C – 3, E – 1. Корень A располагается на 1 уровне, узлы B, C – на втором, H, J, D, E, F – на третьем, G – на четвертом.

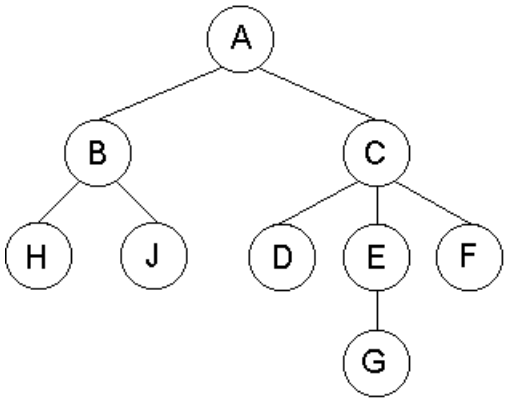



**Рис. 25. Пример дерева**

***Задание***

*Определите корень, листья и узлы со степенью 3 для дерева, изображенного на Рис. 24.*

*Путем* из узла $n_1$ в узел $n_k$ называется последовательность узлов $n_1, n_2, \ldots, n_k$, где $\forall i, 1 \le i < k$, узел $n_i$ является родителем узла $n_{i+1}$. Если существует путь из узла $n_1$ в узел $n_k$, то $n_1$ называется предком $n_k$, а $n_k$ – потомком $n_1$. *Длиной* пути называется число на 1 меньше числа узлов, составляющих этот путь. Например, для дерева изображенного на Рис. 26. путем из вершины A в вершину G является последовательность вершин A, C, E, G. Длина этого пути равна 3.

***Задание***

*Определите путь от вершины «Фильмы» до вершины «Мадагаскар» и его длину для дерева, изображенного на Рис. 25.*

*Высотой узла* дерева называется длина самого длинного пути из этого узла до какого-либо листа. *Высота дерева* совпадает с высотой корня. Например, для дерева, изображенного на Рис. 25, высота узла H равна 0, высота узла B – 1, узла C – 2, высота дерева – 3.

*Глубина узла* определяется как длина пути от корня до этого узла. Например, для дерева, изображенного на Рис. 25, глубина узла H равна 2, глубина узла B – 1, а узла G – 3.

***Задание***

*Определите высоту и глубину узла «Фильмы» для дерева, изображенного на Рис. 24.*

*Бинарное дерево* – это дерево, в котором каждый узел имеет не более двух поддеревьев. В этом случае будем различать левое и правое поддерево. Например, дерево, изображенное на Рис. 25. не является бинарным, т.к. узел C содержит три поддерева. Бинарное дерево изображено на Рис. 26.

*Дерево двоичного поиска* – это бинарное дерево, узлы которого помечены элементами множества. Определяющее свойство дерева двоичного поиска заключается в том, что все элементы, хранящиеся в узлах левого поддерева любого узла x, меньше элемента, содержащегося в узле x, а все элементы, хранящиеся в узлах правого поддерева узла x, больше элемента, содержащегося в узле x. Это свойство называется *характеристическим свойством дерева двоичного поиска* и выполняется для любого узла дерева двоичного поиска, включая его корень.

EPAM Systems

*Идеально сбалансированное дерево* – это дерево минимальной высоты из элементов некоторого множества, для каждого узла которого будет выполняться условие: модуль разности количеств узлов в любых двух его поддеревьях не превышает единицы.

Например, бинарное дерево, изображенное на Рис. 26, является деревом бинарного поиска. Однако условие идеального сбалансирования для этого дерева не выполняется: для узла со значением 25 количество узлов в левом поддереве равно 3, а в правом поддереве – 1. Дерево, изображенное на Рис. 27, является деревом бинарного поиска и идеально сбалансированным деревом одновременно.

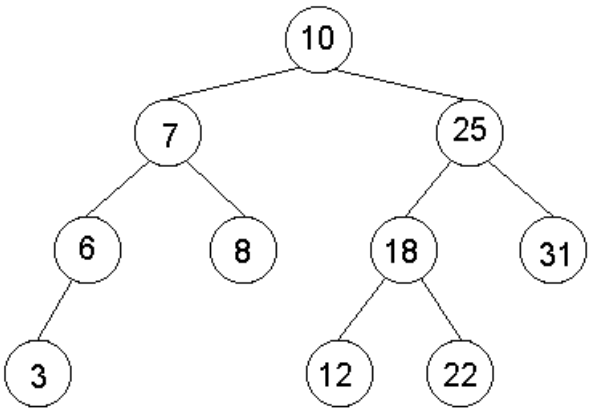



**Рис. 26. Дерево бинарного поиска**

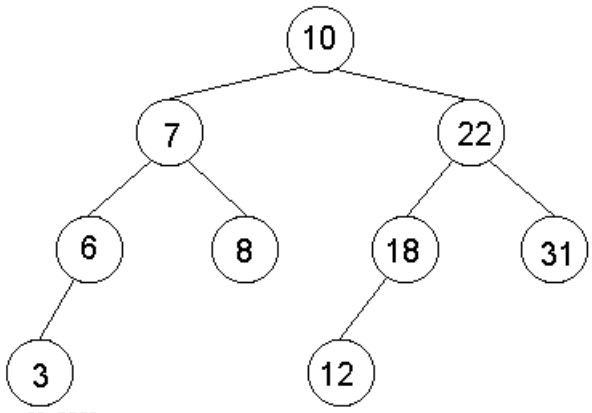



**Рис. 27. Идеально сбалансированное дерево, а также дерево бинарного поиска**

***Задание***

*Определите, является ли дерево, изображенное на Рис. 25, деревом бинарного поиска, или идеальносбалансированным деревом.*

Далее мы будем рассматривать только деревья бинарного поиска.

Программным путем реализовать АТД дерево можно с помощью одномерного массива, массива структур, или с помощью ссылок. Мы будем использовать ссылки.

**Деревья бинарного поиска**

Структуру бинарного дерева, в том числе, и дерева бинарного поиска можно представить следующим образом:

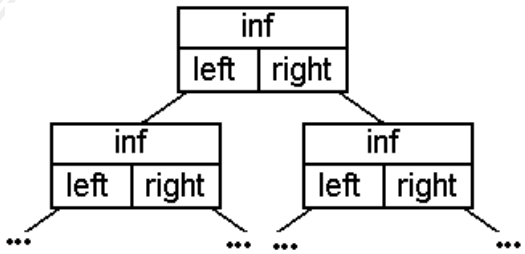



**Рис. 28. Структура бинарного дерева**

Каждый элемент предложенной структуры имеет:

- информационное поле (узел) inf, которое может быть любого типа;
- ссылочные поля left и right, которые будут использоваться для организации связи узлов дерева.

Предложенная структура базового элемента дерева может быть реализована с помощью вложенного класса Node. Дополнительно класс Node будет содержать функциональные члены, позволяющие не только создать узел дерева, но и, в силу рекурсивной природы АТД, сформировать все дерево (класс Tree).

Рассмотрим программную реализацию АТД дерево:

```
using System;
namespace Example
{
   //класс, реализующий АТД «дерево бинарного поиска»
   public class BinaryTree
   {
      //вложенный класс, отвечающий за узлы и операции допустимы
      //для дерева бинарного поиска
      private class Node
      {
         public object inf; //информационное поле
         public Node left;  //ссылка на левое поддерево
         public Node right; //ссылка на правое поддерево
         //конструктор вложенного класса, создает узел дерева
         public Node(object nodeInf)
         {
            inf = nodeInf;
            left = null;
            right = null;
         }
         // добавляет узел в дерево так, чтобы дерево
         // оставалось деревом бинарного поиска
         public static void Add(ref Node r, object nodeInf)
         {
            if (r == null)
            {
               r = new Node (nodeInf);
            }
            else
            {
               if (((IComparable)(r.inf)).CompareTo(nodeInf) > 0)
               {
                  Add(ref r.left, nodeInf);
               }
               else
               {
                  Add(ref r.rigth, nodeInf);
               }
            }
         }
         //прямой обход дерева
         public static void Preorder (Node r)
         {
            if (r != null)
            {
```

EPAMSystems

```
      Console.Write("{0} ", r.inf);
      Preorder(r.left);
      Preorder(r.right);
   }
}
//симметричный обход дерева
public static void Inorder (Node r)
{
   if (r != null)
   {
      Inorder(r.left);
      Console.Write("{0} ",r.inf);
      Inorder(r.right);
   }
}
//обратный обход дерева
public static void Postorder (Node r)
{
   if (r != null)
   {
      Postorder(r.left);
      Postorder(r.right);
      Console.Write("{0} ", r.inf);
   }
}
//поиск ключевого узла в дереве
public static void Search(Node r, object key,
                          out Node item)
{
   if (r == null)
   {
      item = null;
   }
   else
   {
      if (((IComparable)(r.inf)).CompareTo(key) == 0)
      {
         item = r;
      }
      else
      {
         if (((IComparable)(r.inf)).CompareTo(key) > 0)
         {
            Search(r.left, key, out item);
         }
         else
         {
            Search (r.right, key, out item);
         }
      }
   }
}
```

EPAM Systems

```
//методы Del и Delete позволяют удалить узел в дереве
//так, чтобы дерево при этом
//оставалось деревом бинарного поиска
private static void Del (Node t, ref Node tr)
{
   if (tr.right != null)
   {
      Del(t, ref tr.right);
   }
   else
   {
      t.inf = tr.inf;
      tr = tr.left;
   }
}
public static void Delete (ref Node t, object key)
{
   if (t == null)
   {
      throw new Exception("Данное значение в
                           дереве отсутствует");
   }
   else
   {
      if (((IComparable)(t.inf)).CompareTo(key) > 0)
      {
         Delete(ref t.left, key);
      }
      else
      {
         if (((IComparable)(t.inf)).CompareTo(key) <0)
         {
            Delete(ref t.right, key);
         }
         else
         {
            if (t.left == null)
            {
               t = t.right;
            }
            else
            {
               if(t.right == null)
               {
                  t = t.left;
               }
               else
               {
                  Node tr = t.left;
                  Del(t, ref tr);
               }
            }
```

ЕРАМ Systems

```
            }
         }
      }
   }
}     //конец вложенного класса
Node tree;    //ссылка на корень дерева
//свойство позволяет получить доступ к
//значению информационного поля корня дерева
public object Inf
{
   set { tree.inf = value; }
   get { return tree.inf; }
}
public BinaryTree()//открытый конструктор
{
   tree = null;
}
private BinaryTree(Node r) //закрытый конструктор
{
   tree = r;
}
public void Add(object nodeInf)  //добавление узла в дерево
{
   Node.Add(ref tree, nodeInf);
}
//организация различных способов обхода дерева
public void Preorder ()
{
   Node.Preorder(tree);
}
public void Inorder ()
{
   Node.Inorder (tree);
}
public void Postorder ()
{
   Node.Postorder(tree);
}

//поиск ключевого узла в дереве
public BinaryTree Search(object key)
{
   Node r;
   Node.Search(tree, key, out r);
   BinaryTree t = new BinaryTree(r);
   return t;
}
```

EPAM Systems

```
        //удаление ключевого узла в дереве
        public void Delete(object key)
        {
           Node.Delete(ref tree, key);
        }
     }
  }
```

Как вы уже, наверное, обратили внимание, в классе Node и в классе BinaryTree содержатся одноименные методы. При этом основная задача методов класса BinaryTree вызвать соответствующие рекурсивные статические методы класса Node. С одной стороны, это сделано для того, чтобы реализовать корректную работу рекурсивных методов с объектами, с другой стороны, чтобы скрыть реализацию АТД дерева от пользователя и избежать возможных ошибок в организации связей между узлами дерева. Рассмотрим указанные тандемы более подробно.

***Методы Add и Node.Add*** позволяют добавить новый узел в дерево так, чтобы формировалось дерево бинарного поиска. Новый узел должен быть сформирован либо как корень дерева (если дерево было до этого пустое), либо в виде левого или правого сына сформированного раньше узла дерева, у которого этот сын отсутствует. Определение места для вставки нового узла производится на основе анализа значения ссылки t:

1) если дерево пусто, или найдено место для нового узла, то инициализируется новый узел;
2) если дерево не пустое, то определятся положение нового узла в этом дереве:
   a) если добавляемое значение меньше значения данного узла, то поиск места новой записи продолжается по левому поддереву данного узла; для этого производится рекурсивный вызов метода Add для левого поддерева;
   b) если добавляемое значение больше значения данного узла, то поиск места новой записи продолжается по правому поддереву данного узла; для этого производится рекурсивный вызов метода Add для правого поддерева.

В соответствии с предложенным алгоритмом метод Node.Add выглядит следующим образом:

```
//добавляет узел в дерево так, чтобы дерево оставалось
//деревом бинарного поиска
public static void Add(ref Node r, object nodeInf)
{
   if (r == null)  //если корень или найдено местоположение узла
   {
      r = new Node (nodeInf);    //то инициализируем узел
   }
   else
   {
      //если добавляемое значение меньше информационного
      //поля текущего узла
      if (((IComparable)(r.inf)).CompareTo(nodeInf) > 0)
      {
         //то спускаемся по левому поддереву
         Add(ref r.left, nodeInf);
      }
      else
      {
```

```
            //иначе по правому поддереву
            Add(ref r.right, nodeInf);
         }
      }
   }
```

Для того чтобы вызвать метод Node.Add добавляем в класс BinaryTree одноименный метод:

```
public void Add(object nodeInf)  //добавление узла в дерево
{
      //при этом местоположение нового узла определяется
      //относительно корня дерева
      Node.Add(ref tree, nodeInf);
}
```

Рассмотрим следующий фрагмент программы:

```
static void Main()
{
   int[] mas = {10, 7, 25, 31, 18, 6, 3, 12, 22, 8};
   BinaryTree tree = new BinaryTree();
   foreach (int item in mas)
   {
      tree.Add(item);
   }
   …
}
```

В результате выполнения данного фрагмента программы будет сформировано дерево, изображенное на Рис. 29.

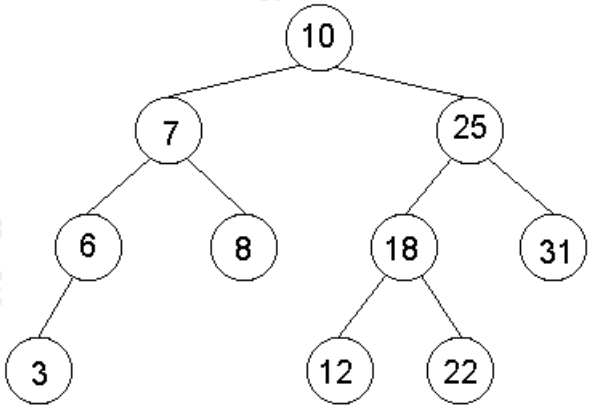



**Рис. 29. Дерево бинарного поиска**

***Задания***

*Пусть дерево бинарного поиска строится на основе следующей последовательности целых чисел {45, 17, 23, 56, 16, 6, 19, 80, 49, 101, 14, 0, 95}. Изобразите графически полученное дерево.*

*Объясните, почему в методе Node.Add формальный параметр r, являясь ссылкой, передается со спецификатором ref.*

***Методы Preorder (Node.Preorder), Inorder (Node.Inorder), Postorder(Node. Postorder)*** предназначены для выполнения обходов бинарных деревьев. Обойти дерево – это побывать в каждом из его узлов точно по одному разу. Рассмотрим три наиболее часто используемых

способов обхода бинарных деревьев – это обход в прямом, симметричном и обратном порядке. Все три обхода будем определять рекурсивно.

1. Прямой обход:
   1) попасть в корень;
   2) пройти левое поддерево данного корня;
   3) пройти правое поддерево данного корня.

Метод, реализующий алгоритм прямого обхода в классе Node, можно записать следующим образом:

```
public static void Preorder (Node r)  //прямой обход дерева
{
   if (r != null)
   {
      Console.Write("{0} ", r.inf);
      Preorder(r.left);
      Preorder(r.right);
   }
}
```

Чтобы запустить прямой обход из класса BinaryTree, используется одноименный метод:

```
public void Preorder ()  //прямой обход дерева
{
   Node.Preorder(tree);  //начиная с корня дерева
}
```

2. Симметричный обход:
   1) пройти левое поддерево данного корня;
   2) попасть в корень;
   3) пройти правое поддерево данного корня.

Метод, реализующий алгоритм симметричного обхода в классе Node, можно записать следующим образом:

```
public static void Inorder (Node r)   //симметричный обход дерева
{
   if (r != null)
   {
      Inorder(r.left);
      Console.Write("{0} ", r.inf);
      Inorder(r.right);
   }
}
```

Чтобы запустить симметричный обход из класса BinaryTree используется одноименный метод:

```
public void Inorder ()     //симметричный обход дерева
{
   Node. Inorder (tree);   //начиная с корня дерева
}
```

3. в) Обратный обход:
   1) пройти левое поддерево данного корня;

EPAM Systems

2) пройти правое поддерево данного корня;
3) попасть в корень.

Метод, реализующий алгоритм обратного обхода в классе Node, можно записать следующим образом:

```
public static void Postorder (Node r) //обратный обход дерева
{
   if (r != null)
   {
      Postorder(r.left);
      Postorder(r.right);
      Console.Write("{0} ", r.inf);
   }
}
```

Чтобы запустить обратный обход из класса BinaryTree используется одноименный метод:

```
public void Postorder ()   //обратный обход дерева
{
   Node.Postorder (tree);  //начиная с корня дерева
}
```

Рассмотрим выполнение обходов на примере дерева, изображенного на Рис. 29.

1. При прямом обходе последовательность узлов выглядит следующим образом:
   10 7 6 3 8 25 18 12 22 31
2. При симметричном обходе последовательность узлов выглядит следующим образом:
   3 6 7 8 10 12 18 22 25 31
3. При обратном обходе последовательность узлов выглядит следующим образом:
   3 6 8 7 12 22 18 31 25 10

Таким образом, при симметричном обходе дерева бинарного поиска на экран выводится упорядоченная по возрастанию последовательность данных. Этот свойство дерева бинарного поиска можно использовать для сортировки данных.

***Замечание***

*Изложенные виды обходов применимы ко всем типам бинарных деревьев.*

***Задание***

*Для дерева, построенного при выполнении предыдущего задания, запишите последовательность узлов при прямом, обратном и симметричном обходе.*

***Методы Search и Node.Search*** предназначены для организации поиска ссылки на узел в дереве по заданному ключу.

Метод Node.Search предназначен для поиска узла по ключу key в дереве с корнем r. В качестве результата метод возвращает ссылку item на найденный узел, или null, если ключевое поле в дереве не содержится. При этом поиск происходит рекурсивно. Если t=null, то либо дерево пустое, либо просмотрены все узлы дерева и искомый узел не найден. В этом случае функция возвращает значение null. В противном случае, производится анализ значения текущего узла путем сравнения его с искомым значением key:

EPAM Systems

1) если key = r.inf, то искомый узел найден и возвращается ссылка на него;
2) если key < r.inf, то поиск продолжается по левому поддереву данного узла, иначе по правому поддереву данного узла;

Метод, реализующий алгоритм поиска в классе Node, можно записать следующим образом:

```
public static void Search(Node r, object key, out Node item)
{
   if (r == null)
   {
      item = null;
   }
   else
   {
      if (((IComparable)(r.inf)).CompareTo(key) == 0)
      {
         item = r;
      }
      else
      {
         if (((IComparable)(r.inf)).CompareTo(key) > 0)
         {
            Search(r.left, key, out item);
         }
         else
         {
            Search (r.right, key, out item);
         }
      }
   }
}
```

Чтобы запустить поиск из класса BinaryTree используется одноименный метод:

```
public BinaryTree Search(int key)
{
   Node r;
   //запускаем поиск ключевого элемента относительно корня дерева
   Node.Search(tree, key, out r);
   //преобразуем полученную ссылку на искомый элемент к
   //типу BinaryTree за счет вызова скрытого конструктора
   BinaryTree t = new BinaryTree(r);
   return t;
}
```

Рассмотрим следующий фрагмент программы, полагая, что у нас сформировано дерево соответствующее Рис. 29:

```
tree.Preorder(); //выполнили прямой обход дерева
//получили ссылку на узел дерева со значением 25
BinaryTree ob = tree.Search(31);
ob.Inf = 100; //изменили значение данного узла
tree.Preorder(); //повторно выполнили прямой обход дерева
```

EPAM Systems

Результат выполнения фрагмента программы:

```
10 7 6 3 8 25 18 12 22 31
10 7 6 3 8 25 18 12 22 100
```

***Задание***

*Рассмотренный фрагмент программы изменил значение только одного узла дерева. Объясните, останется ли при этом дерево деревом бинарного поиска.*

***Методы Delete и Node.Delete*** предназначены для удаления ключевого узла из дерева. Удаление производится таким образом, что дерево остается деревом бинарного поиска.

Метод Node.Delete имеет два формальных параметра: key –значение удаляемого узла и t – ссылка на анализируемый узел (в начале – на корень дерева). В процессе поиска удаляемого узла дерева могут быть 3 случая:

1) удаляемого узла в дереве нет;
2) удаляемый узел имеет не более одного сына;
3) удаляемый узел имеет двух сыновей.

Поиск по дереву производится путем обхода дерева. Если в результате этого поиска обнаружено, что ссылка t равна null, то удаляемого узла в дереве нет и выдается сообщение об ошибке. В противном случае:

1) если key<t.inf, то поиск продолжается по левому поддереву данного узла;
2) если key>t.inf, то поиск продолжается по правому поддереву данного узла;
3) если не выполняется ни условие п.1, ни условие п.2, то искомый узел найден и его следует удалить так, чтобы дерево осталось деревом бинарного поиска. Для этого проводится анализ количества потомков у узла:
   a. если у узла нет левого поддерева, то производится исключение этого узла из дерева;
   b. если у узла есть только левое поддерево, то производится анализ наличия правого поддерева:
      i. если нет правого поддерева, то производится исключение этого узла из дерева;
      ii. если у узла есть и левое, и правое поддерево, вызывается вспомогательный метод Del, который на место удаляемого узла помещает самый правый узел в левом поддереве.

Метод, реализующий алгоритм удаления в классе Node можно записать следующим образом:

```
private static void Del (Node t, ref Node tr)
{
   if (tr.rigth!=null)
   {
      Del(t, ref tr.right);
   }
   else
   {
      t.inf = tr.inf;
      tr = tr.left;
   }
}
```

```
public static void Delete (ref Node t, object key)
{
   if (t == null)  //если узел не найден, то генерируем исключение
   {
      throw new Exception("Данное значение в дереве отсутствует");
   }
   else
   {
      //если информационное поле больше ключа, то
      if (((IComparable)(t.inf)).CompareTo(key) > 0)
      {
         //продолжаем поиск в левом поддереве
         Delete(ref t.left, key);
      }
      else
      {
         //если информационное поле больше ключа, то
         if (((IComparable)(t.inf)).CompareTo(key) < 0)
         {
            //продолжаем поиск в правом поддереве
            Delete(ref t.right, key);
         }
         else //в противном случае узел найден, и мы его удаляем
         { // для этого
            if (t.left == null)//если у узла нет левого потомка,
            {
               t = t.right;     //то заменяем его правым потомком
            }
            else
            {
               if(t.right == null)//если у узла нет правого потомка,
               {
                  t = t.left;     //то заменяем его левым потомком
               }
               else //в противном случае у узла есть оба потомка,
               {
                  // поэтому вызываем вспомогательный метод Del,
                  // который заменяет удаляемый узел на самый
                  // правый узел в левом поддереве
                  Node tr = t.left;
                  Del(t, ref tr);
               }
            }
         }
      }
   }
}
```

Чтобы запустить метод удаления из класса BinaryTree используется одноименный метод:

```
public void Delete(int key)
{
   Node.Delete(ref tree, key);
}
```

EPAM Systems

Рассмотрим следующий фрагмент программы, полагая, что у нас сформировано дерево соответствующее Рис. 29:

```
tree.Preorder(); //выполняем прямой обход дерева
tree.Delete(25); //удаляем узел со значением 25
Console.WriteLine();
tree.Preorder(); //повторно выполняем прямой обход дерева
```

Результат работы фрагмента:

```
10 7 6 3 8 25 18 12 22 31
10 7 6 3 8 22 18 12 31
```

Таким образом, после удаления узла со значением 25 из дерева, представленного на Рис. 29, получим дерево, соответствующее Рис. 30. А после удаления из этого же дерева узла со значением 6 получим дерево, представленное на Рис. 31.

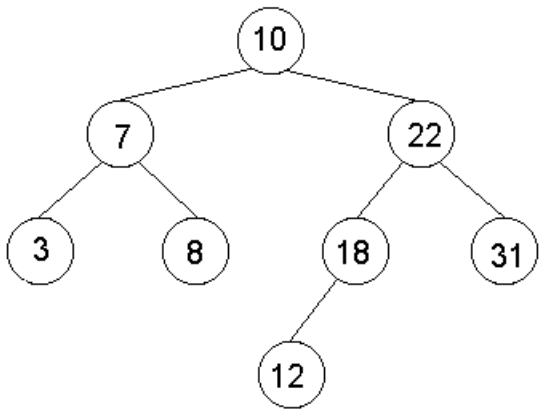



**Рис. 30. Удален узел со значением 6**

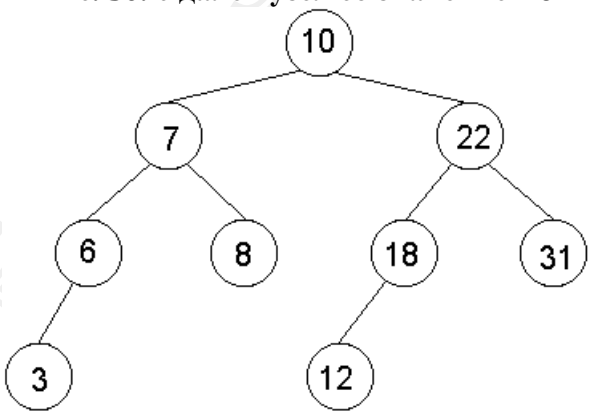



**Рис. 31. Удален узел со значением 25**

***Задание***

*В дереве, изображенном на Рис. 31, удалите узел со значение 10. Графически изобразите полученное дерево.*

**Решение практических задач**

***Пример 1***

В файле input.txt хранится последовательность целых чисел. Построить по этой последовательности дерево двоичного поиска и вывести листья данного дерева.

Для решения данной задачи в класс Node добавим статический метод Sheet:

```
public static void Sheet(Node t)
{
   if (t != null) //если узел не пустой
   {
      //но не имеет потомков, то он лист
      if (t.left == null && t.right == null)
      {
         //выводим на экран его значение
         Console.Write("{0} ", t.inf);
      }
      else               //в противном случае
      {
         Sheet(t.left);     //ищем листья в левом поддереве
         Sheet(t.right);    //и в правом поддереве
      }
   }
}
```

Соответственно в класс BinaryTree добавим метод Sheet:

```
public void Sheet()
{
   Console.WriteLine();
   Node. Sheet (tree);
   Console.WriteLine();
}
```

Теперь рассмотрим основную программу:

```
using System;
using System.IO;

namespace Example
{
   class Program
   {
      static void Main()
      {
         BinaryTree tree = new BinaryTree(); //инициализируем дерево
         //на основе данных файла создаем дерево
         using (StreamReader fileIn =
                   new StreamReader("d:/Example/input.txt"))
         {
            string line = fileIn.ReadToEnd();
            string[] data = line.Split(' ');
            foreach (string item in data)
            {
               tree.Add(int.Parse(item));
            }
         }
         //используя прямой обход выводим на экран узлы дерева
         tree.Preorder();
         tree. Sheet();  //выводим на экран листья дерева
      }
```

EPAM Systems

```
    }
}
```

Результат работы программы

```
input.txt
10 7 25 31 18 6 3 12 22 8
Вывод на экран:
10 7 6 3 8 25 18 12 22 31
3 8 12 22 31
```

***Задание***

*Измените программу так, чтобы на экран выводилось количество листьев данного дерева.*

***Пример 2***

В файле input.txt хранится последовательность целых чисел. Построить по этой последовательности дерево двоичного поиска и вывести экран высоту этого дерева.

Для решения данной задачи в класс Node добавим статический метод

```
HeightTree (Node t, ref int count, ref int height)
```

в котором параметр t отвечает за ссылку на корень дерева, параметр count за длину пути от корня до текущего узла, параметр height – за высоту узла.

```
public static void HeightTree(Node t, ref int count,
                              ref int height)
{
   if (t != null)  // если текущий узел не пустой
   {
      // и длина пути от корня до текущего узла больше
      // высоты дерева, то
      if (count > height)
      {
         // полагаем в качестве высоты дерева длину пути до
         // текущего узла
         height = count;
      }
      // в любом случае увеличиваем  длину пути от корня до
      // текущего узла
      count++;
      // обходим левое поддерево
      HeightTree(t.left, ref count, ref height);
      // обходим правое поддерево
      HeightTree(t.right, ref count, ref height);
      // после чего уменьшаем длину пути от корня до текущего узла
      count--;
   }
}
```

В класс BinaryTree добавим одноименный метод:

```
public int HeightTree()
{
   int count = 0;
   int height = 0;
```

```
      //начинаем подсчет высоты с корня дерева
      Node.HeightTree(tree, ref count, ref height);
      return height;
   }
```

Теперь рассмотрим саму программу:

```
using System;
using System.IO;
namespace Example
{
   class Program
   {
      static void Main()
      {
         BinaryTree tree = new BinaryTree();
         using (StreamReader fileIn =
                  new StreamReader("d:/Example/text.txt"))
         {
            string line = fileIn.ReadToEnd();
            string[] data = line.Split(' ');
            foreach (string item in data)
            {
               tree.Add(int.Parse(item));
            }
         }
         //выводим на экран высоту дерева
         Console.WriteLine(tree.HeightTree());
      }
   }
}
```

Результат работы программы

```
input.txt
10 7 25 31 18 6 3 12 22 8
Вывод на экран:
3
```

***Задание***

*Измените программу так, чтобы для каждого узла дерева выводилась его высота.*

***Пример 3***

В файле input.txt хранится последовательность целых чисел. Построить по этой последовательности дерево двоичного поиска и вывести на экран путь из вершины *a* в вершину *b* этого дерева (если путь не существует, то вывести сообщение об этом).

Для решения данной задачи нам потребуется подключить к проекту класс Queue, а такжедобавить в класс Node добавить следующий метод:

```
public static void Way(Node t, int b,ref Queue q)
{
   if (t != null)  //если узел не пустой
   {
      q.Add(t.inf); //помещаем значение узла в очередь
      //если найден искомый узел
```

```
            if (((IComparable)(t.inf)).CompareTo(b) == 0)
            {
               return;    //то завершаем работу метода
            }
            else          //в противном случае выполняем обход поддеревьев
            {
               if (((IComparable)(t.inf)).CompareTo(b) > 0)
               {
                  Way(t.left, b, ref q);
               }
               else
               {
                  Way(t.right, b, ref q);
               }
            }
         }
         else //если ссылка на узел равна null, то узлы a и b лежали
              //в разных поддереьях
         {    //относительно корня и пути между ними нет,
              //поэтому освобождаем очередь
            while (!q.IsEmpty)
            {
               q.Take();
            }
         }
      }
```

В класс BinaryTree добавим одноименный метод:

```
      public void Way(int a, int b)
      {
         Node itemA;
         //проверяем, содержится ли узел a в дереве
         Node.Search(tree,a, out itemA);
         Node itemB;
         //проверяем, содержится ли узел b в дереве
         Node.Search(tree,b, out itemB);
         if (itemA == null)
         {
            Console.WriteLine("узла {0} нет в дереве", a);
         }
         else
         {
            if (itemB == null)
            {
               Console.WriteLine("узла {0} нет в дереве", b);
            }
            else //если узлы a и b содержатся в дереве
            {
               Queue q = new Queue(); //то инициализируем очередь
               //запускаем метод поиска пути от ссылки itemA до узла b
               Node.Way(itemA, b, ref q);
```

EPAM Systems

```
            //если очередь не пуста, то выводим путь на экран
            if (!q.Empty)
            {
               while (!q.Empty)
               {
                  Console.Write("{0} ", q.Take());
               }
               Console.WriteLine();
            }
            else //иначе выдаем соответствующее сообщение
            {
               Console.WriteLine("пути между узлами {0} и {1} не
                                  существует", a, b);
            }
         }
      }
   }
```

Теперь рассмотрим саму программу:

```
using System;
using System.IO;

namespace Example
{
   class Program
   {
      static void Main()
      {
         BinaryTree tree = new BinaryTree();
         using (StreamReader fileIn =
                     new StreamReader("d:/Example/input.txt"))
         {
            string line = fileIn.ReadToEnd();
            string[] data = line.Split(' ');
            foreach (string item in data)
            {
               tree.Add(int.Parse(item));
            }
         }
         Console.Write("a= ");
         int a = int.Parse(Console.ReadLine());
         Console.Write("b= ");
         int b = int.Parse(Console.ReadLine());
         tree.Way(a, b);
      }
   }
}
```

EPAM Systems

Результат работы программы:

```
input.txt
10  7  25  31  18  6  3  12  22  8
a   b   сообщение на экране
10  22  10  25  18  22
3   31  путь между узлами 3 и 31 не существует
7   3   7 6 3
```

***Задание***

*Измените программу так, чтобы на экран выводилась длина искомого пути от узла a до узла b.*

***Пример 4***

Дан текстовый файл. На основе текстового файла создать «словарь» и вывести на экран упорядоченные по алфавиту пары ключ/значение, где ключ – это слово, содержащееся в «словаре», а значение – это количество вхождений ключа в исходный файл.

Для решения данной задачи будем использовать дерево бинарного поиска, т.к. оно позволяет эффективно организовать хранение и поиск данных. Для этого создадим новый класс DictionaryTree:

```
using System;
namespace Example
{
   public class DictionaryTree
   {
      private class Node
      {
         //ассоциированная пара ключ/значение
         public object inf;
         public uint count;

         public Node left;  //ссылка на левое поддерево
         public Node right; //ссылка на правое поддерево

         //конструктор вложенного класса, создает узел дерева
         public Node(object nodeInf)
         {
            count=1;
            inf = nodeInf;
            left = null;
            right = null;
         }

         //добавляет узел в дерево так, чтобы дерево
         //оставалось деревом бинарного поиска
         //и каждое слов вносилось в него только один раз
         public static void Add(ref Node r, object nodeInf)
         {
            if (r == null)
            {
               r = new Node (nodeInf);
            }
            else
            {
```

EPAM Systems

```
                    if (((IComparable)(r.inf)).CompareTo(nodeInf) > 0)
                    {
                       Add(ref r.left, nodeInf);
                    }
                    else
                    {
                       if (((IComparable)(r.inf)).CompareTo(nodeInf) < 0)
                       {
                          Add(ref r.right, nodeInf);
                       }
                       else
                       {
                          r.count++;
                       }
                    }
                 }
              }
              public static void Inorder (Node r)
              {
                 if (r != null)
                 {
                    Inorder(r.left);
                    Console.WriteLine("{0} {1}", r.inf, r.count);
                    Inorder(r.right);
                 }
              }
           } //конец вложенного класса
           Node tree;      //ссылка на корень дерева
           public DictionaryTree()    //открытый конструктор
           {
              tree = null;
           }
           public void Add(object nodeInf)  //добавление узла в дерево
           {
              Node.Add(ref tree, nodeInf);
           }
           //симметричный обход дерева позволит выводить
           //словарь в алфавитном порядке
           public void Inorder ()
           {
              Node.Inorder (tree);
           }
        }
     }
```

Теперь рассмотрим саму программу:

```
using System;
using System.IO;
namespace Example
{
```

```
class Program
{
   static void Main()
   {
      DictionaryTree tree = new DictionaryTree();
      using (StreamReader fileIn =
                  new StreamReader("d:/Example/input.txt"))
      {
         string line = fileIn.ReadToEnd();
         string[] data = line.Split(' ');
         foreach (string item in data)
         {
            tree.Add(item);
         }
      }
      tree.Inorder();
   }
}
```

*Результат работы программы:*

```
input.txt
to be or not to be
Вывод на экран
be 2
not 1
or 1
to 2
```

***Задание***

*Добавьте в класс DistionaryTree:*

1) *возможность определять, содержится ли ключевое слово в словаре и сколько раз;*
2) *возможность эффективного поиска слов начинающихся на заданную букву.*

## Практикум №15

***Задание 1***

В файле input.txt хранится последовательность целых чисел. По входной последовательности построить дерево бинарного поиска и найти для него:

1) сумму нечетных значений узлов дерева;
2) количество четных значений узлов дерева;
3) среднее арифметическое положительных значений узлов дерева;
4) наибольшее из значений листьев;
5) сумму значений листьев;
6) количество узлов, имеющих только одного левого потомка;
7) сумму значений узлов, имеющих только одного правого потомка;
8) количество узлов, имеющих двух потомков;
9) среднее арифметическое значение узлов, имеющих два потомка;
10) количество узлов, значение которых больше среднего арифметического.

EPAM Systems

***Задание 2***

В файле input.txt хранится последовательность целых чисел. По входной последовательности построить дерево бинарного поиска и:

1) распечатать узлы k-го уровня дерева;
2) найти количество узлов на k-том уровне дерева;
3) распечатать дерево по уровням;
4) вычислить глубину заданного узла;
5) для каждого узла дерева вычислить его глубину;
6) заменить отрицательные значения узлов дерева на противоположные;
7) проверить, является ли дерево идеально сбалансированным;
8) проверить, можно ли удалить какой-то один узел так, чтобы дерево стало идеально сбалансирванным;
9) проверить, останется ли дерево деревом бинарного поиска, если из него удалить все нечетные узлы;
10) проверить, является ли первое дерево поддеревом второго, и найти для него наименьшее из значений листьев;
11) найти для него количество узлов, имеющих только одного правого потомка;
12) найти для него сумму значений узлов, имеющих только одного потомка;
13) определить, количество узлов дерева со значением, отличным от *k*;
14) проверить, является ли данное дерево деревом бинарного поиска;
15) поменять в нем местами узлы, хранящие минимальное и максимальное значение;
16) найти глубину заданного узла.

***Задание 3***

Мы рассмотрели реализацию АТД «дерево бинарного поиска», которое по способу построения может оказаться «перегруженным» на левое или правое поддерево. В некоторых задачах возникает необходимость равномерно разместить элементы некоторой последовательности по уровням дерева. В этой ситуации лучше использовать идеально сбалансированные деревья.

Разработайте класс, реализующий АТД «идеально сбалансированное дерево», самостоятельно определив в нем функциональные члены класса.

С помощью разработанного класса решите следующую задачу: дан файл, компонентами которого являются целые числа. На основе файла создайте идеально сбалансированное дерево. Выведите его на экран по уровням. Удалите в нем наибольший и наименьший элементы так, чтобы дерево осталось идеально сбалансированным. Затем еще раз выведите дерево по уровням.

***Задание 4***

Код Хаффмана.

Приведем пример применения бинарных деревьев в качестве структур данных. Для этого рассмотрим задачу конструирования кода Хаффмана. Предположим, что мы имеем сообщение, состоящее из последовательности символов. В каждом сообщении символы независимы и появляются с известной вероятностью, не зависящей от позиции символа в сообщении. Например, мы имеем сообщение из шести символов *a, b, c, +, *, /* с вероятностью появления символов в тексте 0.32, 0.1, 0.11, 0.22, 0.13 и 0.12 соответственно.

Мы хотим закодировать каждый символ последовательностью из нулей и единиц так, чтобы код любого символа являлся префиксом кода сообщения, состоящего из последующих

символов. Это префиксное свойство позволяет декодировать строку из нулей и единиц последовательным удалением префиксов (кодов символов) из этой строки. В таблице показаны две возможные кодировки наших символов.

| символ | вероятность | код 1 | код 2 |
|---|---|---|---|
| a | 0.32 | 110 | 11 |
| b | 0.1 | 000 | 000 |
| c | 0.11 | 001 | 001 |
| + | 0.22 | 010 | 01 |
| * | 0.13 | 101 | 101 |
| / | 0.12 | 100 | 100 |

**Таблица 3**

Первый код обладает префиксным свойством, поскольку любая последовательность из трех битов будет префиксом для другой последовательности из трех битов. Алгоритм декодирования очень прост: надо брать по три бита и преобразовывать каждую группу битов в соответствующие символы. Так, например, последовательности 110010000100001010110 (110 010 000 100 001 010 110 – после разбиения на группы битов) будет соответствовать исходное сообщение a+b/c+a.

Второй код также обладает префиксным свойством, однако процесс декодирования чуть сложнее: последовательность нельзя заранее разбить на группы из трех битов, так как символы могут кодироваться двумя или тремя битами. Для примера рассмотрим последовательность 11010001000010111 (11 01 000 100 001 01 11 – после разбиения на группы битов), которой будет соответствовать то же исходное сообщение a+b/c+a.

Преимуществом второго кода является то, что сообщение, закодированное с его помощью, будет иметь минимизированную среднюю длину кода. Так *код 1* имеет среднюю длину кода 3, а *код 2* – 2.7, благодаря чему кодовая последовательность исходного сообщения при кодировании *кодом 2* уменьшилась на четыре бита.

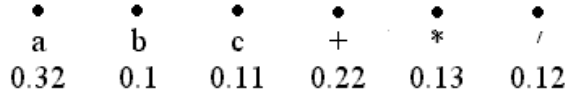
Таким образом, задача конструирования кода Хаффмана заключается в следующем: имея множество символов и значения вероятностей их появления в сообщениях, построить код с префиксным свойством, чтобы средняя длина кода (в вероятностном смысле) последовательности символов была минимальна.

Для построения кода мы будем использовать *лес* – совокупность деревьев, чьи листья будут помечены символами, для которых разрабатывается кодировка, а корни помечены суммой вероятностей этих символов. Мы будем называть эти суммарные вероятности весом дерева. В начале работы каждому символу соответствует дерево, состоящее из одного узла. В конце мы получим одно дерево, все листья которого помечены кодируемыми символами. В этом дереве путь от корня к какому-либо листу представляет код для символа-метки этого листа, составленный по схеме, согласно которой левый сын узла соответствует метке 0, правый сын узла – метке 1.

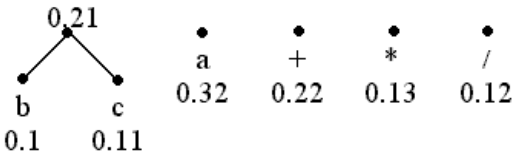
Важным этапом в формировании кода является выбор из леса двух деревьев с наименьшими весами. Эти два дерева комбинируются в одно с весом, равным сумме весов составляющих его деревьев, причем в качестве левого поддерева выбирается дерево с наименьшим весом.

Рассмотрим конструирования кода Хаффмана по шагам.

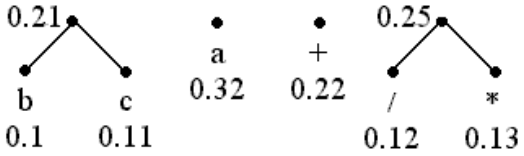
1) Исходная ситуация



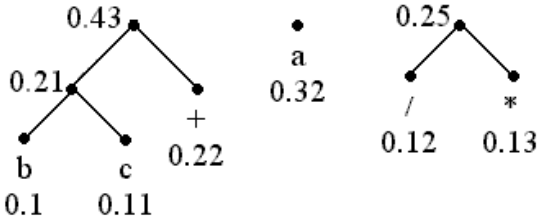

2) Слияние поддеревьев с весами 0.1 и 0.11



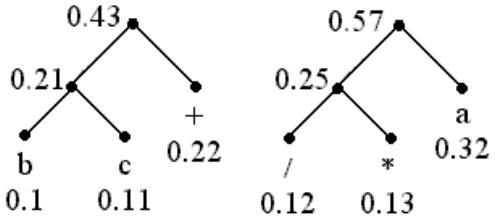

3) Слияние поддеревьев с весами 0.12 и 0.13



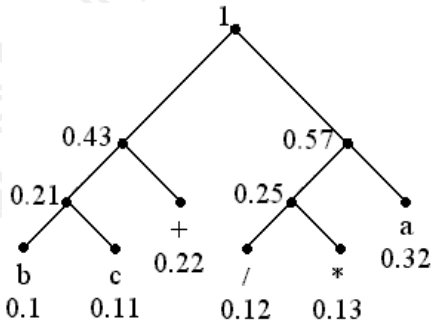

4) Слияние поддеревьев с весами 0.21 и 0.22




5) Слияние поддеревьев с весами 0.25 и 0.32




6) Слияние поддеревьев с весами 0.43 и 0.57




Итоговое бинарное дерево, представляющее код Хаффмана с префиксным свойством изображено на Рис. 32.

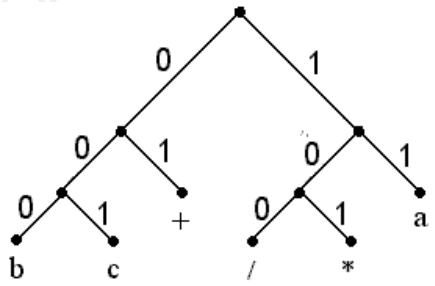



**Рис. 32. Законченное дерево кода Хаффмана**

Чтобы по дереву получить код символа, необходимо спускаться от корня к соответствующему листу, последовательно записывая метки. Коды построенного дерева совпадают с *кодом 2* см. Таблица 3.

Реализуйте алгоритм построения кода Хаффмана самостоятельно и с его помощью проведите кодирование и декодирование произвольных сообщений.

EPAM Systems

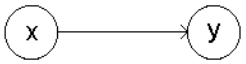
# Графы

Во многих задачах, встречающихся в компьютерных науках, математике, технических дисциплинах, социологии, географии, химии, и ряде других дисциплин часто возникает необходимость наглядно представить отношения между какими-либо объектами. Графы представляют собой естественную модель для таких отношений. В этом разделе мы рассмотрим ориентированные и неориентированные графы, а также структуры данных, которые применяются для представления графов в ЭВМ и некоторые основные алгоритмы обработки данных, представленных в виде графов.

### Основные понятия

*Граф* G состоит из множества вершин V и множества ребер (дуг) E, соединяющих некоторые вершины графа.

*Ориентированным* называется граф, в котором каждая дуга представляет собой упорядоченную пару вершин вида (x, y), где x называется *началом*, а y – *концом* дуги. Дугу (x, y) часто изображают следующим образом:



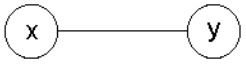

Говорят также, что дуга (x, y) ведет от вершины x к вершине y, а вершина y является *смежной* с вершиной x.

Далее ориентированный граф будем называть *орграфом*. На Рис. 33 показан орграф, в котором вершины – это документы, а дуги – это ссылки одних документов на другие. На Рис. 34 показан орграф, в котором вершинами являются люди, а дуги показывают на проявление симпатии между ними.


**Рис. 33. Связи документов**


**Рис. 34. Проявление симпатий между людьми**

*Неориентированным* называется граф, в котором каждое ребро представляет собой неупорядоченную пару вершин, т.е., ребро (x, y) = (y, x), и изображается в виде:




Вершины, соединенные ребром, называются *смежными*. Ребра, имеющие общую вершину, также называются *смежными*. Ребро и любая из двух его вершин называются *инцидентными*.

Далее неориентированные графы будем называть просто *графами*. На практике графы используется для задания симметричных отношений для объектов. На Рис. 35 приведен пример графа, в котором вершины – это населенные пункты, а ребра – это дороги между ними. На Рис. 36 приведен граф, в котором вершины – это атомы, а ребра – химические связи между неми.

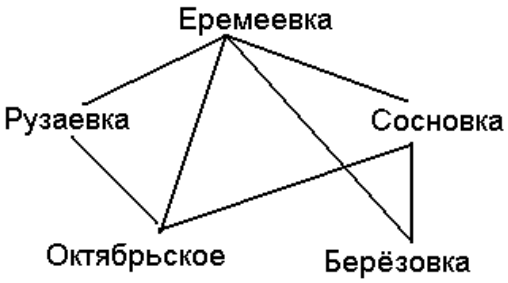



**Рис. 35. Схема дорог**

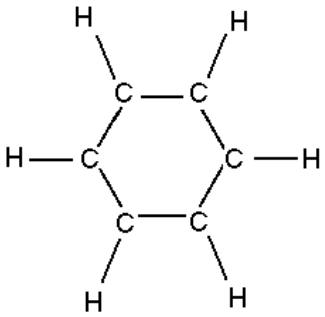


**Рис. 36. Молекула бензола**

Теперь рассмотрим основные понятия, связанные с графами.

Пусть дан орграф, состоящий из n вершин. Вершины пронумерованы от 0 до n-1. Естественно было бы пронумеровать вершины графа с единицы, но т.к. для реализации графа будет использоваться двумерный массив, то мы начали нумеровать вершин с нуля. Если над дугами орграфа указать веса, например, расстояние, или стоимость проезда от одного города до другого, то мы получим взвешенный орграф.

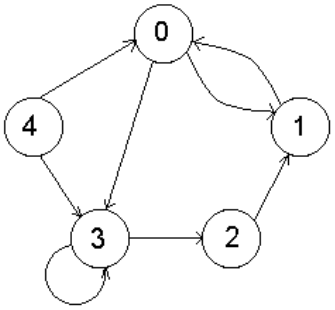


**Рис. 37. Орграф**

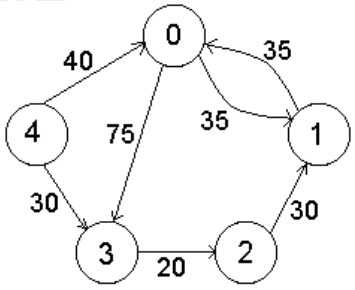


**Рис. 38. Взвешенный орграф**

Аналогичным образом из неориентированного графа можно получить взвешенный неориентированный граф.

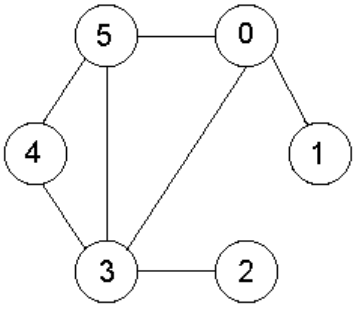


**Рис. 39. Неориентированный граф**

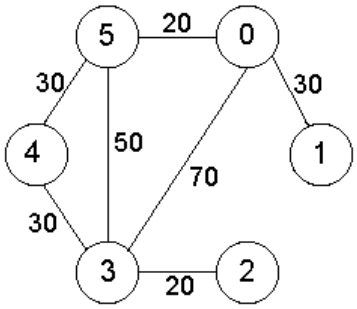


**Рис. 40. Взвешенный неориентированный граф**

*Путем*, соединяющим вершины u и v, назовем такую последовательность вершин $v_0, v_1, \ldots, v_n$ $(n \geq 0)$, где $v_0 = u, v_n = v$, что для всех i, $1 \leq i < n$, существуют ребра $(v_i, v_{i+1})$.

Путь, в котором каждая вершина используется не более одного раза, называется *простым путем*.

Рассмотрим орграф, представленный на Рис. 37. Одним из существующих путей, соединяющих вершины 0 и 2, является последовательность вершин 0, 1, 0, 3, 2.

Единственным простым путем для той же пары вершин является последовательность 0, 3, 2. Пути из вершины 0 в вершину 4 для того же графа не существует.

Путь называют *замкнутым*, если начальная и конечная вершины совпадают. Замкнутый путь называется *циклом*, если все его вершины (кроме начальной и конечной) различны.

Рассмотрим граф, изображенный на Рис. 39. Для него путь 1, 0, 5, 4, 3, 0, 1 является замкнутым, путь 1, 0, 5, 4, 3, 0, 2 – не замкнутым; а путь 0, 5, 4, 3, 0 является циклом.

*Длиной пути в графе* полагают количество ребер (дуг), составляющих этот путь. *Длиной пути во взвешенном графе* полагают сумму весов всех дуг (ребер), входящих в этот путь.

Рассмотрим орграф, изображенный на Рис. 37. Путем из вершины 1 в вершину 2 будет являться последовательность 1, 0, 3, 2; длина этого пути равна 3. Для аналогичного взвешенного орграфа (см. Рис. 38) длина того же пути равна 130.

Если существует путь из вершины u в вершину v, то говорят, что *v достижима из u*. *Источником* в графе называется вершина, из которой достижимы все остальные вершины. *Стоком* называется вершина, достижимая из всех остальных вершин графа.

Например, для графа, изображенного на Рис. 37 источником будет являться вершина 4, а стоками – все остальные вершины.

**Способы представления графов**

Для представления графов можно использовать различные структуры данных. Рассмотрим некоторые из них.

***1. Матрица смежности***

Пусть граф G содержит n вершин. Тогда матрица смежности имеет размерность n×n и элементы ее определяются следующим образом:

$$a_{i,j} = \begin{cases} 1, \textit{ если вершины } i, j \textit{ смежны;} \\ 0, \textit{ если вершины } i, j \textit{ не смежны.} \end{cases}$$

Для взвешенных графов элементы матрицы смежности определяются следующим образом:

$$a_{i,j} = \begin{cases} k_{i,j}, \textit{ если вершины } i, j \textit{ смежны, где } k_{i,j} - \textit{вес ребра (дуги), соединяющего вершины } i, j; \\ 0, \textit{ если вершины } i, j \textit{ не смежны.} \end{cases}$$

Рассмотрим примеры матриц смежности:

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 0 | 0 |
| 3 | 0 | 0 | 1 | 1 | 0 |
| 4 | 1 | 0 | 0 | 1 | 0 |

для Рис. 37

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 35 | 0 | 75 | 0 |
| 1 | 35 | 0 | 0 | 0 | 0 |
| 2 | 0 | 30 | 0 | 0 | 0 |
| 3 | 0 | 0 | 20 | 0 | 0 |
| 4 | 40 | 0 | 0 | 30 | 0 |

для Рис. 38

ERAM Systems

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 1 | 0 | 0 |
| 3 | 1 | 0 | 1 | 0 | 1 | 1 |
| 4 | 0 | 0 | 0 | 1 | 0 | 1 |
| 5 | 1 | 0 | 0 | 1 | 1 | 0 |

для Рис. 39

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 30 | 0 | 70 | 0 | 20 |
| 1 | 30 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 0 | 0 | 20 | 0 | 0 |
| 3 | 70 | 0 | 20 | 0 | 30 | 50 |
| 4 | 0 | 0 | 0 | 30 | 0 | 30 |
| 5 | 20 | 0 | 0 | 50 | 30 | 0 |

для Рис. 40

***Замечание***

*Видно, что матрица смежности для неориентированного графа является симметричной.*

Недостатком этого представления является то, что матрица требует $n^2$ ячеек памяти и может быть сильно разрежена в случае, когда число ребер (дуг) много меньше, чем $n^2$. Достоинством является то, что матрица позволяет быстро определить, соединены ли две данные вершины ребром (дугой).

С этой матрицей достаточно просто работать, поэтому ее имеет смысл использовать для плотных графов (когда количество дуг сравнимо с $n^2$), или когда граф небольшой и места в памяти достаточно.

Программным путем матрицу смежности можно реализовать в виде обычного двумерного массива.

***Задание***

*Постройте матрицу смежности для графа, изображенного на Рис. 35.*

***2. Списки смежности***

Для каждой вершины графа имеется список вершин, таких, что существует ребро/дуга (v,u). Рассмотрим примеры списков смежности:

| | | |
|---|---|---|
| 0 | 2 | 4 |
| 1 | 1 | |
| 2 | 2 | |
| 3 | 3 | 4 |
| 4 | 1 | 5 |

для Рис. 37

| | | | | |
|---|---|---|---|---|
| 0 | 2 | 4 | 6 | |
| 1 | 1 | | | |
| 2 | 4 | | | |
| 3 | 1 | 3 | 5 | 6 |
| 4 | 4 | 6 | | |
| 5 | 1 | 4 | 6 | |

для Рис. 38

Число ячеек памяти в этом случае всего n+m, где n – количество вершин, а m –количество ребер/дуг в графе. Неудобство заключается в том, что когда необходимо определить, есть ли ребро из v в u, приходится просматривать весь список, соответствующий вершине v.

Программным путем список смежности можно реализовать в виде одномерного массива списков, или списка списков.

***Задание***

*Постройте список смежности для графа, изображенного на Рис. 35.*

Существуют и другие способы представления графов. Далее, при рассмотрении алгоритмов обработки данных, представленных в виде графа, будет использоваться представление графа в виде матрицы смежности.

**Алгоритмы обхода графа**

При решении многих задач нам требуется исследовать все вершины графа в том или ином систематическом порядке, как было и в случае с двоичными деревьями. При просмотре дерева у нас есть корневой узел, от которого мы и начинали просмотр. В графе часто нет вершины, выделенной среди всех остальных, поэтому просмотр может начаться с любой произвольной вершины.

Рассмотрим алгоритмы обхода графа в глубину и в ширину.

***Обход графа в глубину***

Алгоритм обхода графа в глубину является обобщением обхода дерева в прямом порядке.

Предположим, что по ходу просмотра только что была просмотрена вершина v, и пусть $w_1$, $w_2$, …, $w_k$ есть вершины, смежные с v и еще не просмотренные. Тогда на следующем шаге мы применяем алгоритм обхода для вершины $w_1$, а остальные вершины $w_2$, …, $w_k$ будут ждать. После посещения вершины $w_1$ мы просматриваем все вершины, смежные с $w_1$, и только после этого возвращаемся к просмотру $w_2$, …, $w_k$.

Таким образом, когда все вершины, достижимые из v, будут просмотрены, обход закончится. Если при этом некоторые вершины графа останутся не просмотренными, это будет означать, что они не достижимы из вершины v.

Если нужно просмотреть оставшиеся вершины, то выбираем одну из не просмотренных вершин и снова запускаем алгоритм обхода. Этот процесс продолжается до тех пор, пока обходом не будут просмотрены все вершины данного графа.

***Обход графа в ширину***

Обход графа в ширину является обобщением поуровневого просмотра дерева.

Если по ходу просмотра только что была просмотрена вершина v, тогда следующим шагом будет посещение всех вершин $w_1$, $w_2$, …, $w_k$, смежных с v и еще не просмотренных. Вершины, смежные с $w_1$, $w_2$, …, $w_k$, буду помещены в список ожидающих, их просмотр будет выполнен после того, как будут просмотрены вершины $w_1$, $w_2$, …, $w_k$. Обход завершится тогда, когда список ожидающих просмотра вершин будет пуст.

Если после завершения обхода некоторые вершины графа останутся не просмотренными, это будет означать, что они не достижимы из вершины v.

На Рис. 41 показан порядок просмотра вершин графа обоими обходами при условии, что начальной была выбрана вершина с номером 4.

ERAM Systems

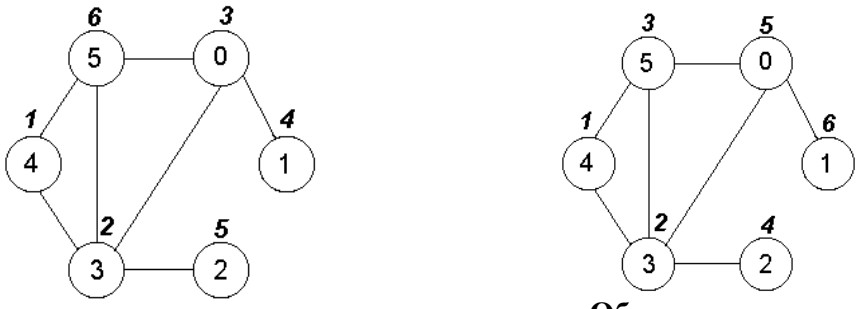



**Рис. 41. Порядок просмотра вершин графа.**

***Задания***

*Выполните обходы графа с Рис. 37, полагая в качестве начальной:*

1) *вершину с номером 1;*
2) *вершину с номером 4.*

Следует отметить, что модификации алгоритмов обхода легли в основу многих других эффективных алгоритмов на графах. Например, алгоритм обхода в ширину позволяет находить кратчайший путь между вершинами a и b, а также циклы в неориентированном графе.

***Замечание***

*Программная реализация рассмотренных алгоритмов будет приведена позже.*

**Алгоритмы нахождения кратчайших путей**

Пусть дан взвешенный орграф G, у которого все веса неотрицательные. Нашей задачей является нахождение такого пути от одной вершины к другой, для которого сумма весов по маршруту принимает минимальное значение. Мы назовем такой путь кратчайшим.

Следует отметить, что вес может представлять собой комплексную характеристику. Например, для графа моделирующего схему дорог, весом ребра/дуги может являться только расстояние от одной вершины до другой, а может и целый комплекс величин, таких как стоимость, время и расстояние.

Как мы говорили ранее, кратчайший путь между вершинами a и b в обычном графе можно найти с помощью модификации алгоритма обхода в ширину. Для взвешенного графа обычно применяют алгоритм Дейкстры, который позволяет решить более общую задачу – нахождение кратчайшего пути от произвольной вершины, называемой источником, до всех остальных вершин. Если необходимо найти кратчайшие пути между всеми парами вершин во взвешенном графе, то используют алгоритм Флойда.

Рассмотрим алгоритм Дейкстры и Флойда более подробно.

***Алгоритм Дейкстры***

Для алгоритма Дейкстры предполагается, что все вершины графа G поименованы целыми числами, т.е., определено множество вершин V={0, 1, 2, …, n-1}, причем вершина $v_0$ считается источником. Граф G определяется массивом $C_{n\times n}$, где $c_{i,j}$ равно стоимости ребра/дуги (i, j), если ребро/дуга (i, j) существует; в противном случае $c_{i,j} = \infty$.

Алгоритм строит множество S вершин, для которых кратчайшие пути от источника уже известны. На каждом шаге к множеству S добавляется та из оставшихся вершин, расстояние

до которой от источника меньше, чем до других оставшихся вершин. Если стоимости ребер/дуг неотрицательны, то можно быть уверенным, что кратчайший путь проходит только через множество S. Назовем такой путь особым. Дополнительно используются:

1) массив D размерности n, в который записываются длины кратчайших особых путей от источника к каждой вершине;
2) массив P размерности n, в котором в $p_i$ записывается вершина, непосредственно предшествующая вершине i в кратчайшем пути.

Рассмотрим сам алгоритм.

1. Полагаем $S=\{v_0\}$.
2. Инициализируем массив D: для u от 1 до n (за исключением $u=v_0$) $D[u]=C[v_0, u]$.
3. Инициализируем массив P: для u от 1 до n (за исключением $u=v_0$) $P[u]=v_0$.
4. Для i от 1 до n-1:
   a. выбираем из множества V\S такую вершину w, что D[w] минимально;
   b. добавляем w к множеству S;
   c. для каждой вершины u из множества V\S: если $D[u] > D[w]+C[w, u]$, то $D[u]=D[w]+C[w, u]$ и $P[u]=w$;

После завершения работы алгоритма для каждого $u \neq v_0$ D[u] хранит кратчайший путь от источника $v_0$ до вершины u; массив P позволит восстановить путь от источника до вершины u.

В качестве примера рассмотрим граф, представленный на Рис. 42.

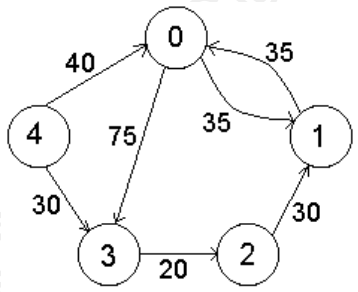



**Рис. 42. Взвешенный орграф**

Для этого графа $C = \begin{bmatrix} \infty & 35 & \infty & 75 & \infty \\ 35 & \infty & \infty & \infty & \infty \\ \infty & 30 & \infty & \infty & \infty \\ \infty & \infty & 20 & \infty & \infty \\ 40 & \infty & \infty & 30 & \infty \end{bmatrix}$.

В качестве $v_0$ берем вершину с номером 4.

Тогда:

```
S={4}
D[0] = 40, D[1] = ∞, D[2] = ∞, D[3] = 30.
P[0] = P[1] = P[2] = P[3] = 4.
Для i = 1:
   w = 3;
   S = {4, 3};
   D[0] = 40 < D[3]+C[3, 0] = 30+∞;
   D[1] =  ∞ < D[3]+C[3, 1] = 30+∞;
```

```
   D[2] = ∞ > D[3]+C[3, 2] = 30+20 ⇒ D[2] = 50, P[2] = 3.
Для i = 2:
   w = 0;
   S = {4, 3, 0};
   D[1] = ∞ > D[0]+C[0, 1] = 40+35 ⇒ D[1] = 75, P[1] = 0;
   D[2] = 50 < D[0]+C[0,2] = 40+∞.
Для i = 3:
   w = 3;
   S = {5, 4, 1, 3};
   D[2] = 75 < D[3]+C[3][2] = 50+30.
Для i = 4:
   w = 1;
   S = {5, 4, 1, 3, 2}.
В итоге:
   D[0] = 40, D[1] = 75, D[2] = 50, D[3] = 30;
   P[0] = 4, P[1] = 0, P[2] = 3, P[3] = 4.
```

Если теперь мы хотим определить стоимость кратчайшего пути от источника $v_0$=4 до вершины 2, то нужно обратиться к D[2]. В данном случае кратчайший путь равен 50. Сам путь соответствует последовательности вершин 4, 3, 2, которую можно восстановить по массиву P (т.к. P[2]=3, то вершине 2 в кратчайшем пути предшествует вершина 3; в свою очередь P[3]=4, следовательно вершине 3 предшествует вершина 4).

***Задание***

*Выполните поиск кратчайшего пути (и его длины) для графа, представленного на Рис. 40 с источником $v_0$=3.*

***Алгоритм Флойда***

Для алгоритма Флойда предполагается, что все вершины графа G поименованы целыми неотрицательными числами, т.е., задано множество вершин V={0, 1, 2, …, n-1}. Граф G определяется массивом $A_{n\times n}$, в котором вычисляются длины кратчайших путей. Вначале $a_{i,j}$ равно стоимости ребра/дуги (i, j) для всех i≠j. Если ребро/дуга (i, j) не существует, то $a_{i,j}=\infty$. Каждый диагональный элемент равен нулю. Для восстановления кратчайшего пути используется матрица $P_{n\times n}$. Вначале $p_{i,j}$= -1 для всех i≠j, т.е., кратчайший путь из вершины i в вершину j состоит из одного ребра/дуги (i, j).

Над матрицей A выполняется n итераций. После итерации с номером k $a_{i,j}$ содержит значение наименьшей длины путей из вершины i в вершину j, которые не проходят через вершины с номером, большим k. Другими словами, между концевыми вершинами пути i и j могут находиться только вершины, номера которых меньше или равны k.

На k-той итерации для вычислении матрицы A применяется следующая формула:

$$A_k[i, j]=\min(A_{k-1}[i, j], A_{k-1}[i, k]+A_{k-1}[k, j])$$

Другими словами, для вычисления $A_k[i, j]$ проводится сравнение величины $A_{k-1}[i, j]$ (т.е., стоимости пути от вершины i к вершине j без участия вершины k, или другой вершины с более высоким номером) с величиной $A_{k-1}[i, k]+A_{k-1}[k, j]$ (т.е., стоимости пути от вершины i до вершины k плюс стоимость пути от вершины k до вершины j). Если путь через вершину k «дешевле», чем $A_{k-1}[i, j]$, то величина $A_k[i, j]$ изменяется, и запоминается текущая вершина, т.е., $P_k[i, j]=k$.

В качестве примера рассмотрим граф, приведенный на Рис. 42. Для этого графа матрицы A и P последовательно примут следующие значения:

Начальное состояние:

A

| | | | | |
|---|---|---|---|---|
| 0 | 35 | ∞ | 75 | ∞ |
| 35 | 0 | ∞ | ∞ | ∞ |
| ∞ | 30 | 0 | ∞ | ∞ |
| ∞ | ∞ | 20 | 0 | ∞ |
| 40 | ∞ | ∞ | 30 | 0 |

P

| | | | | |
|---|---|---|---|---|
| -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 |

Итерации:

$A_0$

| | | | | |
|---|---|---|---|---|
| 0 | 35 | ∞ | 75 | ∞ |
| 35 | 0 | ∞ | 110 | ∞ |
| ∞ | 30 | 0 | ∞ | ∞ |
| ∞ | ∞ | 20 | 0 | ∞ |
| 40 | 75 | ∞ | 30 | 0 |

$P_0$

| | | | | |
|---|---|---|---|---|
| -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | 0 | -1 |
| -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 |
| -1 | 1 | -1 | -1 | -1 |

$A_1$

| | | | | |
|---|---|---|---|---|
| 0 | 35 | ∞ | 75 | ∞ |
| 35 | 0 | ∞ | 110 | ∞ |
| 65 | 30 | 0 | 140 | ∞ |
| ∞ | ∞ | 20 | 0 | ∞ |
| 40 | 75 | ∞ | 30 | 0 |

$P_1$

| | | | | |
|---|---|---|---|---|
| -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | 0 | -1 |
| 1 | -1 | -1 | 1 | -1 |
| -1 | -1 | -1 | -1 | -1 |
| -1 | 1 | -1 | -1 | -1 |

$A_2$

| | | | | |
|---|---|---|---|---|
| 0 | 35 | ∞ | 75 | ∞ |
| 35 | 0 | ∞ | 110 | ∞ |
| 65 | 30 | 0 | 140 | ∞ |
| 85 | 50 | 20 | 0 | ∞ |
| 40 | 75 | ∞ | 30 | 0 |

$P_2$

| | | | | |
|---|---|---|---|---|
| -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | 0 | -1 |
| 1 | -1 | -1 | 1 | -1 |
| 2 | 2 | -1 | -1 | -1 |
| -1 | 1 | -1 | -1 | -1 |

$A_3$

| | | | | |
|---|---|---|---|---|
| 0 | 35 | 95 | 75 | ∞ |
| 35 | 0 | 130 | 110 | ∞ |
| 65 | 30 | 0 | 140 | ∞ |
| 85 | 50 | 20 | 0 | ∞ |
| 40 | 75 | 50 | 30 | 0 |

$P_3$

| | | | | |
|---|---|---|---|---|
| -1 | -1 | 3 | -1 | -1 |
| -1 | -1 | 3 | 0 | -1 |
| 1 | -1 | -1 | 1 | -1 |
| 2 | 2 | -1 | -1 | -1 |
| -1 | 0 | 3 | -1 | -1 |

ERM Systems

$A_4$

| | | | | |
|---|---|---|---|---|
| 0 | 35 | 95 | 75 | ∞ |
| 35 | 0 | 130 | 110 | ∞ |
| 65 | 30 | 0 | 140 | ∞ |
| 85 | 50 | 20 | 0 | ∞ |
| 40 | 75 | 50 | 30 | 0 |

$P_4$

| | | | | |
|---|---|---|---|---|
| -1 | -1 | 3 | -1 | -1 |
| -1 | -1 | 3 | 0 | -1 |
| 1 | -1 | -1 | 1 | -1 |
| 2 | 2 | -1 | -1 | -1 |
| -1 | 0 | 3 | -1 | -1 |

Если теперь мы хотим определить стоимость кратчайшего пути от вершины 4 до вершины 2, то нужно обратиться к A[4, 2]. В данном случае кратчайший путь равен 50. Сам путь соответствует последовательности вершин 4, 3, 2, который можно восстановить по массиву P.

***Задание***

*Примените алгоритм Флойда к графу, представленному на Рис. 40.*

**Программная реализация АТД «граф»**

Реализацию АТД «граф» мы будем осуществлять с помощью следующего класса Graph:

```
using System;
using System.IO;
namespace Example
{
   public class Graph
   {
      //вложенный класс для скрытия данных и алгоритмов
      private class Node
      {
         private int[,] array;   //матрица смежности
         //индексатор для обращения к матрице смежности
         public int this [int i, int j]
         {
            get
            {
               return array[i,j];
            }
            set
            {
               array[i,j] = value;
            }
         }
         //свойство для получения числа строк/столбцов
         //матрицы смежности
         public int Size
         {
            get
            {
               return array.GetLength(0);
            }
         }
         //вспомогательный массив: если i-ый элемент массива равен
         //true, то i-ая вершина еще не просмотрена; если i-ый
```

```
//элемент равен false, то i-ая вершина просмотрена
private bool[] nov;

//метод помечает все вершины графа как непросмотреные
public void NovSet()
{
   for (int i=0; i<Size; i++ )
   {
      nov[i] = true;
   }
}
//конструктор вложенного класса, инициализирует матрицу
// смежности и вспомогательный массив
public Node(int[,] a)
{
   array = a;
   nov = new bool[a.GetLength(0)];
}
//реализация алгоритма обхода графа в глубину
public void Dfs(int v)
{
   //просматриваем текущую вершину
   Console.Write("{0} ", v);
   nov[v] = false;      //помечаем ее как просмотренную
   // в матрице смежности просматриваем строку с номером v
   for (int u = 0; u < Size; u++)
   {
      //если вершины v и u смежные, к тому же вершина u
      //не просмотрена,
      if (array[v,u] != 0 && nov[u])
      {
         Dfs(u); // то рекурсивно просматриваем вершину
      }
   }
}
//реализация алгоритма обхода графа в ширину
public void Bfs(int v)
{
   Queue q = new Queue(); // инициализируем очередь
   q.Add(v);       //помещаем вершину v в очередь
   nov[v] = false; // помечаем вершину v как просмотренную
   while (!q.IsEmpty)    // пока очередь не пуста
   {
      v = q.Take(); //извлекаем вершину из очереди
      Console.Write("{0} ", v);     //просматриваем ее
      //находим все вершины
      for (int u = 0; u < Size; u++)
      {
         // смежные с данной и еще не просмотренные
         if (array[v,u] != 0 && nov[u])
         {
            //помещаем их в очередь
```

EPAM Systems

```
                q.Add(u);
                //и помечаем как просмотренные
                nov[u] = false;
             }
          }
       }
    }
    //реализация алгоритма Дейкстры
    public long[] Dijkstr(int v, out int []p)
    {
       nov[v] = false; // помечаем вершину v как просмотренную
       //создаем матрицу c
       int[,] c = new int [Size,Size];
       for (int i = 0; i < Size; i++)
       {
          for (int u = 0; u < Size; u++)
          {
             if (array[i,u] == 0 || i == u)
             {
                c[i,u] = int.MaxValue;
             }
             else
             {
                c[i,u] = array[i,u];
             }
          }
       }
       //создаем матрицы d и p
       long[]d = new long [Size];
       p = new int [Size];
       for (int u = 0; u < Size; u++)
       {
          if (u != v)
          {
             d[u] = c[v,u];
             p[u] = v;
          }
       }
       for (int i = 0; i < Size-1; i++) // на каждом шаге цикла
       {
          // выбираем из множества V\S такую вершину w,
          // что D[w] минимально
          long min = int.MaxValue;
          int w = 0;
          for (int u = 0; u < Size; u++)
          {
             if (nov[u] && min > d[u])
             {
                min = d[u];
                w = u;
             }
          }
```

EDAN Systems

```
            nov[w] = false; //помещаем w в множество S
            //для каждой вершины из множества V\S определяем
            //кратчайший путь от источника до этой вершины
            for (int u = 0; u < Size; u++)
            {
               long distance = d[w] + c[w,u];
               if (nov[u] && d[u] > distance)
               {
                  d[u] = distance;
                  p[u] = w;
               }
            }
         }
         //в качестве результата возвращаем массив кратчайших
         //путей для заданного  источника
         return d;
      }
      //восстановление пути от вершины a до вершины b для
      //алгоритма Дейкстры
      public void WayDijkstr(int a, int b, int[] p,
                             ref Stack items)
      {
         items.Push(b);  //помещаем вершину b в стек
         if (a == p[b])  //если предыдущей для вершины b является
                         //вершина a, то
         {
            items.Push(a);     //помещаем a в стек и завершаем
                               //восстановление пути
         }
         else    //иначе метод рекурсивно вызывает сам себя для
                 //поиска пути от вершины a до вершины,
                 //предшествующей вершине b
         {
            WayDijkstr(a, p[b], p, ref items);
         }
      }
      //реализация алгоритма Флойда
      public long[,] Floyd(out int [,]p)
      {
         int i,j,k;
         //создаем массивы p и a
         long[,] a = new long[Size, Size];
         p = new int[Size, Size];
         for (i = 0; i < Size; i++)
         {
            for (j = 0; j < Size; j++)
            {
               if (i == j)
               {
                  a[i,j] = 0;
               }
```

ЕРАМ Systems

```
                else
                {
                   if (array[i,j] == 0)
                   {
                      a[i,j] = int.MaxValue;
                   }
                   else
                   {
                      a[i,j] = array[i,j];
                   }
                }
                p[i,j] = -1;
             }
          }
          //осуществляем поиск кратчайших путей
          for(k = 0; k < Size; k++)
          {
             for (i = 0; i < Size; i++)
             {
                for (j = 0; j < Size; j++)
                {
                   long distance = a[i, k] + a[k,j];
                   if (a[i,j] > distance)
                   {
                      a[i,j] = distance;
                      p[i,j] = k;
                   }
                }
             }
          }
          return a; //в качестве результата возвращаем массив
                    //кратчайших путей между всеми парами вершин
       }
       //восстановление пути от вершины a до вершины в
       //для алгоритма Флойда
       public void WayFloyd(int a, int b, int[,] p,
                            ref Queue items)
       {
          int k = p[a,b];
          //если k != -1, то путь состоит более чем из двух вершин
          //a и b, и проходит через вершину k, поэтому
          if (k != -1)
          {
             //рекурсивно восстанавливаем путь между вершинами
             //a и k
             WayFloyd(a, k, p, ref items);
             items.Add(k); //помещаем вершину к в очередь
             //рекурсивно восстанавливаем путь между вершинами
             //k и b
             WayFloyd(k,b,p,ref items);
          }
       }
    } //конец вложенного класса
```

ЕДИНАЯ СИСТЕМА

```
private Node graph;//закрытое поле, реализующее АТД «граф»
public Graph(string name)  //конструктор внешнего класса
{
   using (StreamReader file = new StreamReader(name))
   {
      int n = int.Parse(file.ReadLine());
      int[,] a = new int[n, n];
      for (int i = 0; i < n; i++)
      {
         string line = file.ReadLine();
         string[] mas = line.Split(' ');
         for (int j = 0; j < n; j++)
         {
            a[i, j] = int.Parse(mas[j]);
         }
      }
      graph = new Node(a);
   }
}
//метод выводит матрицу смежности на консольное окно
public void Show ()
{
   for (int i = 0; i < graph.Size; i++)
   {
      for (int j = 0; j < graph.Size; j++)
      {
         Console.Write("{0,4}", graph[i,j]);
      }
      Console.WriteLine();
   }
}
public void Dfs(int v)
{
   //помечаем все вершины графа как непросмотренные
   graph.NovSet();
   graph.Dfs(v); //запускаем алгоритм обхода графа в глубину
   Console.WriteLine();
}
public void Bfs(int v)
{
   //помечаем все вершины графа как непросмотренные
   graph.NovSet();
   graph.Bfs(v); //запускаем алгоритм обхода графа в ширину
   Console.WriteLine();
}
public void Dijkstr(int v)
{
   //помечаем все вершины графа как непросмотренные
   graph.NovSet();
   int[] p;
```

ERAM Systems

```
            //запускаем алгоритм Дейкстры
            long[] d = graph.Dijkstr(v,out p);
            //анализируем полученные данные и выводим их на экран
            Console.WriteLine("Длина кратчайшие пути от вершины {0}
                               до вершины", v);
            for (int i = 0; i < graph.Size; i++)
            {
               if (i != v)
               {
                  Console.Write("{0} равна {1}, ", i, d[i]);
                  Console.Write("путь ");
                  if (d[i] != int.MaxValue)
                  {
                     Stack items = new Stack();
                     graph.WayDijkstr(v, i, p, ref items);
                     while (!items. IsEmpty)
                     {
                        Console.Write("{0} ", items.Pop());
                     }
                  }
               }
               Console.WriteLine();
            }
         }
         public void Floyd()
         {
            int[,] p;
            //запускаем алгоритм Флойда
            long[,] a = graph.Floyd(out p);
            int i, j;
            //анализируем полученные данные и выводим их на экран
            for (i = 0; i < graph.Size; i++)
            {
               for (j = 0; j < graph.Size; j++)
               {
                  if (i != j)
                  {
                     if (a[i,j] == int.MaxValue)
                     {
                        Console.WriteLine("Пути из вершины {0} в вершину
                                           {1} не существует", i, j);
                     }
                     else
                     {
                        Console.Write("Кратчайший путь  от вершины {0}
                                       до вершины {1}      равен {2}, ",
                                       i, j, a[i,j]);
                        Console.Write(" путь ");
                        Queue items = new Queue();
                        items.Add(i);
                        graph.WayFloyd(i, j, p, ref items);
                        items.Add(j);
```

ERA Systems

```
                        while (!items. IsEmpty)
                        {
                           Console.Write("{0} ", items.Take());
                        }
                        Console.WriteLine();
                     }
                  }
               }
            }
         }
      }
```

Как видим, предложенная реализация АТД «граф» инициализирует объект класса Graph на основе данных из файла. При этом в файле в первой строке нужно указать размерность графа, а далее привести матрицу смежности. Например, для взвешенного орграфа с Рис. 43 файл input.txt выглядит следующим образом:

```
5
0  35 0  75 0
35 0  0  0  0
0  30 0  0  0
0  0  20 0  0
40 0  0  30 0
```

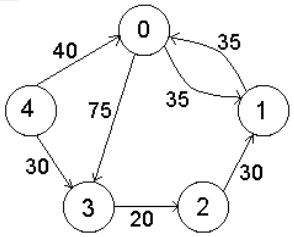



**Рис. 43. Взвешенный орграф.**

В общем случае в файле может быть записана матрица смежности как для орграфа, так и для неориентированного графа, при этом граф может быть и невзвешенным.

***Замечание***

*Данная реализация использует такие структуры данных как стек и очередь, поэтому соответствующие классы должны быть доступны для вашего проекта.*

Рассмотрим пример использования класса Graph:

```
using System;
namespace Example
{
   class Program
   {
      static void Main()
      {
         Graph g = new Graph("d:/Example/text.txt");
         Console.WriteLine("Graph:");
         g.Show();
         Console.WriteLine();
         Console.WriteLine("DFS:");
         g.Dfs(4);
         Console.WriteLine();
         Console.WriteLine("BFS:");
         g.Bfs(4);
         Console.WriteLine();
```

```
            Console.WriteLine("Dijkstr:");
            g.Dijkstr(4);
            Console.WriteLine();
            Console.WriteLine("Floyd:");
            g.Floyd();
        }
    }
}
```

Результат работы программы:

```
Graph:
0   35  0   75  0
35  0   0   0   0
0   30  0   0   0
0   0   20  0   0
40  0   0   30  0

DFS:
4 0 1 3 2

BFS:
4 0 3 1 2

Dijkstr:
Длина кратчайшего пути от вершины 4 до вершины
0 равна 40, путь 4 0
1 равна 75, путь 4 0 1
2 равна 50, путь 4 3 2
3 равна 30, путь 4 3

Floyd:
Кратчайший путь от вершины 0 в 1 равен 35, путь 0 1
Кратчайший путь от вершины 0 в 2 равен 95, путь 0 3 2
Кратчайший путь от вершины 0 в 3 равен 75, путь 0 3
Путь из вершины 0 в вершину 4 не существует
Кратчайший путь от вершины 1 в 0 равен 35, путь 1 0
Кратчайший путь от вершины 1 в 2 равен 130, путь 1 0 3 2
Кратчайший путь от вершины 1 в 3 равен 110, путь 1 0 3
Путь из вершины 1 в вершину 4 не существует
Кратчайший путь от вершины 2 в 0 равен 65, путь 2 1 0
Кратчайший путь от вершины 2 в 1 равен 30, путь 2 1
Кратчайший путь от вершины 2 в 3 равен 140, путь 2 1 0 3
Путь из вершины 2 в вершину 4 не существует
Кратчайший путь от вершины 3 в 0 равен 85, путь 3 2 1 0
Кратчайший путь от вершины 3 в 1 равен 50, путь 3 2 1
Кратчайший путь от вершины 3 в 2 равен 20, путь 3 2
Путь из вершины 3 в вершину 4 не существует
Кратчайший путь от вершины 4 в 0 равен 40, путь 4 0
Кратчайший путь от вершины 4 в 1 равен 75, путь 4 0 1
Кратчайший путь от вершины 4 в 2 равен 50, путь 4 3 2
Кратчайший путь от вершины 4 в 3 равен 30, путь 4 3
```

EPAM Systems

***Задание***

*Внесите в файл input.txt данные, соответствующие Рис. 33. Запустите программу и объясните полученные данные, в частности, что будет выведено в качестве длины пути.*

**Решение практических задач с использованием графов**

***Пример 1***

Во входном файле input.txt указаны количество вершин графа и его матрица смежности. Вывести на экран вершины, соседние с заданной вершиной.

Добавим в класс Graph метод Neighbouring, который для заданной вершины v выводит на экран соседние вершины:

```
public void Neighbouring(int v)
{
   Console.Write("Вершины соседние с {0} вершиной: ", v);
   //просматриваем строку с номером v в матрице смежности
   for (int i = 0; i < graph.Size; i++)
   {
      //если на пересечении строки v и столбца i стоит не ноль, то
      //вершина i является соседней для вершины v
      if (graph[v,i] != 0)
      {
         Console.Write("{0} ", i);
      }
   }
   Console.WriteLine();
}
```

Теперь рассмотрим основную программу:

```
using System;
namespace Example
{
   class Program
   {
      static void Main()
      {
         Graph g = new Graph("d:/Example/input.txt");
         Console.WriteLine("Graph:");
         g.Show();
         int v = 3;
         g.Neighbouring(v);
      }
   }
}
```

Результат работы программы:

```
Graph:
0 1 0 1 0 1
1 0 0 0 0 0
0 0 0 1 0 0
1 0 1 0 1 1
0 0 0 1 0 1
```

EPAM Systems

```
1 0 0 1 1 0
Вершины соседние с 3 вершиной: 0 2 4 5
```

***Замечание***

*Данные, приведенные в файле* input.txt, *соответствуют Рис. 39.*

***Задание***

*Измените метод Neighbouring так, чтобы в качестве результата он возвращал список вершин, соседних с заданной вершиной. Продемонстрируйте на примере использование данного метода.*

***Пример 2***

Во входном файле input.txt указаны количество вершин графа и его матрица смежности. Вывести на экран вершины, достижимые из заданной вершины.

Для решения данной задачи можно воспользоваться любым алгоритмом обхода графа – Bfs, или Dfs. Вершины, которые окажутся достижимыми из данной, будут выведены на экран. Однако в данном случае вершины будут выведены в порядке их обхода, а не по возрастанию их номеров. Кроме того, вершина, с которой начался обход, также будет выводиться на экран. Чтобы решить эту проблему, в класс Node добавим метод Reach, который выполняет обход графа в глубину для вершины v, но не выводит просмотренные вершины на экран:

```
public void Reach (int v)
{
   nov[v] = false;
   for (int u = 0;u < size; u++)
   {
      if (array[v,u] != 0 && nov[u])
      {
         Reach(u);
      }
   }
}
```

В класс Graph добавим метод Reachable, который на основе вспомогательного массива nov выводит на экран достижимые вершины:

```
public void Reachable (int v)
{
   graph.NovSet();
   Console.Write("Вершины достижимые из {0} вершины: ", v);
   graph.Reach(v);
   for (int i = 0; i < graph.Size; i++)
   {
      //если вершина была просмотрена, то она достижима
      if (!graph.NovGet(i) && i != v)
      {
         Console.Write("{0} ", i);
      }
   }
   Console.WriteLine();
}
```

Теперь рассмотрим саму программу:

```
using System;
namespace Example
{
   class Program
   {
      static void Main()
      {
         Graph g = new Graph("d:/Example/input.txt");
         Console.WriteLine("Graph:");
         g.Show();
         int v = 3;
         g.Reachable(v);
      }
   }
}
```

Результат работы программы:

```
Graph:
0 1 0 1 0
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
1 0 0 1 0
Вершины достижимые из 3 вершины: 0 1 2
```

***Замечание***

*Данные, приведенные в файле* input.txt, *соответствуют Рис. 37.*

***Задание***

*Измените метод Reachable так, чтобы в качестве результата он возвращал список вершин, достижимых из заданной вершины. Продемонстрируйте на примере использование данного метода.*

***Пример 3***

В файле input.txt указаны количество вершин в графе и матрица смежности взвешенного графа. Вывести на экран номер наиболее удаленной достижимой вершины (возможно, она не единственная).

В класс Graph добавляем метод InMaxDistance, который позволяет определить вершины, наиболее удаленные от заданной. Причем, если таких вершин будет несколько, то метод выведет все вершины.

```
public void InMaxDistance(int v)
{
   graph.NovSet(); //помечаем все вершины графа как непросмотренные
   int[] p;
   long[] d = graph.Dijkstr(v,out p); //запускаем алгоритм Дейкстры
   long max = 0;
   //вычисляем наибольшее расстояние для достижимых вершин
   for (int i = 0; i < p.Length; i++)
   {
      if (d[i] > max && d[i] != int.MaxValue)
      {
```

```
            Max = d[i];
         }
      }
      //выводим на экран все вершины, находящиеся от заданной на
      //наибольшем расстоянии
      if (max == 0)
      {
         Console.WriteLine("Из заданной вершины другие вершины
                           недостижимы");
      }
      else
      {
         Console.Write("На наибольшем удалении от вершины {0}
                        находятся вершины: ", v);
         for (int i = 0; i < d.Length; i++)
         {
            if (d[i] == max)
            {
               Console.Write("{0} ", i);
            }
         }
      }
   }
```

Теперь рассмотрим саму программу:

```
using System;
namespace Example
{
   class Program
   {
      static void Main()
      {
         Graph g = new Graph("d:/Example/input.txt");
         Console.WriteLine("Graph:");
         g.Show();
         int v = 1;
         g.InMaxDistance(v);
      }
   }
}
```

Результат работы программы для v=1:

```
Graph:
0   35  0   75  0
35  0   0   0    0
0   30  0   0    0
0   0   20  0    0
40  0   0   30   0
На наибольшем удалении от вершины 1 находятся вершины: 2
```

***Замечание***

*Данные, приведенные в файле* input.txt, *соответствуют Рис. 38.*

***Задание***

*Разработайте метод InMinDistance, который в качестве результата возвращет список вершин, наиболее близких из достижимых. Продемонстрируйте на примере использование данного метода.*

***Пример 4***

В файле input.txt указаны количество вершин в графе и матрица смежности взвешенного графа. Вывести на экран центральную вершину графа.

***Замечание***

*Центральной вершиной графа называется вершина с минимальным эксцентриситетом, т.е. такая вершина, для которой максимальное расстояние до других вершин минимально. Для определения центральной вершины необходимо:*

1) *с помощью алгоритма Флойда определить все кратчайшие пути в графе;*
2) *для каждой строки матрицы A определить максимальное из кратчайших путей; это значение равно эксцентриситету вершины i;*
3) *среди найденных эксцентриситетов вершин найти эксцентриситет с наименьшим значением; соответствующая ему вершина будет являться центром графа.*

Для решения данной задачи в класс Graph добавим метод CentralApex, вычисляющий центральную вершину графа:

```
public int CentralApex()
{
   int[,] p;
   long[,] a = graph.Floyd(out p);     //вызываем алгоритм Флойда
   long min = int.MaxValue;
   int imin = -1;
   for (int i = 0; i < graph.Size; i++)
   {
      int imax = i;
      int jmax = 0;
      // для каждой строки матрицы определяем эксцентриситет
      for (int j = 0; j < graph.Size; j++)
      {
         if (a[i,j] > a[imax, jmax])
         {
            imax = i;
            jmax = j;
         }
      }
      //среди найденных эксцентриситетов определяем наименьший
      if (a[imax, jmax] < min)
      {
         min = a[imax, jmax];
         imin = imax;
      }
   }
   return imin; //возвращаем номер вершины с наименьшим
                //эксцентриситетом
}
```

ERAM Systems

Теперь рассмотрим саму программу:

```
using System;
namespace Example
{
   class Program
   {
      static void Main()
      {
         Graph g = new Graph("d:/Example/input.txt");
         Console.WriteLine("Graph:");
         g.Show();
         Console.WriteLine("Центральная вершина графа {0}",
                           g.CentralApex());
      }
   }
}
```

Результат работы программы:

```
Graph:
0   30  0   70  0   20
30  0   0   0   0   0
0   0   0   20  0   0
70  0   20  0   30  50
0   0   0   30  0   30
20  0   0   50  30  0
Центральная вершина графа 5
```

***Замечание***

*Данные, приведенные в файле* input.txt, *соответствуют Рис. 40.*

***Задание***

*Граф может содержать несколько центральных вершин. В связи с этим, измените метод CentralApex так, чтобы в качестве результата он возвращал список центральных вершин графа. Продемонстрируйте на примере использование данного метода.*

EPAM Systems

## Практикум №16

***Задание 1***

Во входном файле указывается количество вершин графа/орграфа и матрица смежности.

Для заданного графа:

1) подсчитать количество вершин, смежных с данной;
2) вывести на экран все вершины, не смежные с данной;
3) удалить из графа ребро, соединяющее вершины a и b;
4) добавить в граф ребро, соединяющее вершину a и b;
5) добавить новую вершину;
6) исключить данную вершину;
7) выяснить, соседствуют ли две заданные вершины с третьей.

Для заданного орграфа:

8) для данной вершины вывести на экран все "выходящие" соседние вершины;

9) для данной вершины вывести на экран все "входящие" соседние вершины;
10) удалить дугу, соединяющую вершины a и b;
11) добавить дугу, соединяющую вершину a и b;
12) исключить данную вершину;
13) добавить новую вершину.

Для взвешенного графа:

14) найти все вершины графа, недостижимые из данной;
15) определить, существует ли путь длиной не более L между двумя данными вершинами;
16) найти все вершины, из которых существует путь в данную;
17) найти все истоки графа;
18) найти все стоки графа;
19) найти вершину, наиболее удаленную от центральной вершины графа;
20) найти вершину, наиболее приближенную к центральной вершине графа.

***Задание 2***

Во входном файле задается:

1) в первой строке N – количество городов;
2) начиная со второй строки через пробел названия N-городов;
3) с новой строки матрица смежности взвешенного графа, описывающая схему дорог.

Например, входной файл для Рис. 35 может выглядеть следующем образом:

```
5
Березовка Еремеевка Октябрьское Рузаевка Сосновка
0  70 0  0  20
70 0  75 25 15
0  75 0  40 60
0  25 40 0  0
20 15 60 0  0
```

Остальные данные, необходимые для решения задачи, вводятся с клавиатуры.

1. Найти кратчайший путь, соединяющий города A и B и проходящий только через заданное множество городов.
2. Определить, существует ли город, из которого можно добраться до каждого их остальных городов, проезжая не более N км.
3. N-периферией называется множество городов, расстояние от которых до выделенного города (столицы) больше N. Определить N-периферию для заданной столицы и значения N.
4. Определите, между какими городами нужно построить дорогу, что бы расстояние между любыми городами не превышало N км.
5. Определите, какое наименьшее количество дорог нужно закрыть (и между какими городами), чтобы из города A нельзя было попасть в город B.

***Задание 3***

Модифицировать рассмотренные алгоритмы обходов графа и алгоритмы Дейкстры и Флойда, реализовав граф в виде списка смежности.

EPAM Systems