МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С. П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)

Е.В. СИМОНОВА

СТРУКТУРЫ ДАННЫХ В C#

Часть II. НЕЛИНЕЙНЫЕ ДИНАМИЧЕСКИЕ СТРУКТУРЫ

Рекомендовано редакционно-издательским советом федерального государственного автономного образовательного учреждения высшего образования «Самарский национальный исследовательский университет имени академика С.П. Королева» в качестве учебного пособия для студентов, обучающихся по основной образовательной программе высшего образования по направлению подготовки 09.03.01 Информатика и вычислительная техника

ISBN 978-5-7883-1291-0 (Ч.II)
ISBN 978-5-7883-1294-1

С А М А Р А

Издательство Самарского университета

2018

УДК 004.9(075)
ББК 32.97я7
С 375

Рецензенты: канд. техн. наук, доц. Л. С. З е л е н к о;
д-р техн. наук, проф. С. В. С м и р н о в

Симонова, Елена Витальевна
С 375 **Структуры данных в C#. Часть II. Нелинейные динамические структуры**: учеб. пособие / *Е.В. Симонова.* – Электрон. текст. и граф. дан. (2,1 Мб). – Самара: Издательство Самарского университета, 2018. – 1 опт. компакт-диск (CD-ROM). – Систем. требования: ПК Pentium, Adobe Acrobat Reader. – Загл. с титул. экрана.

ISBN 978-5-7883-1291-0 (Ч.II)
ISBN 978-5-7883-1294-1

Учебное пособие включает разделы, которые подробно описывают рекурсивные алгоритмы обработки структур данных, иерархические структуры данных (деревья и графы), принципы организации множества объектов с заданным отношением порядка на основе хеширования. Теоретический материал иллюстрируется большим количеством программных фрагментов, реализующих алгоритмы обработки различных структур данных. Содержит контрольные вопросы и упражнения по всем разделам.

Предназначено для студентов направления подготовки 09.03.01 Информатика и вычислительная техника.

Подготовлено на кафедре информационных систем и технологий.

УДК 004.9 (075)
ББК 32.97я7

№ госрегистрации 0321900458

Редактор М.С. Сараева

Компьютерная вёрстка А.В. Ярославцевой

Подписано для тиражирования 16.11.2018.

Объем издания 2,1 Мб.

Количество носителей 1 диск.

Тираж 10 экз.

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ ИМЕНИ АКАДЕМИКА С. П. КОРОЛЕВА»
(САМАРСКИЙ УНИВЕРСИТЕТ)

443086, САМАРА, МОСКОВСКОЕ ШОССЕ, 34.

Изд-во Самарского университета.
443086, Самара, Московское шоссе, 34.

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

В учебном пособии описаны структуры данных и алгоритмы, которые широко используются при решении разнообразных задач в широком спектре предметных областей.

Учебное пособие предназначено для студентов, обучающихся по направлению 09.03.01 – Информатика и вычислительная техника.

Содержание учебного пособия соответствует разделам рабочей программы по дисциплине «Программирование» федерального компонента ГОС подготовки бакалавров по направлению 09.03.01 – Информатика и вычислительная техника.

Глава 1 посвящена рассмотрению рекурсии как метода организации множества объектов и вычислительных процессов. Подробно описывается структура рекурсивного вычислительного процесса, особенности его реализации. Приводится большое количество примеров рекурсивных алгоритмов, нашедших широкое применение при решении задач различной природы.

В главе 2 описываются рекурсивные иерархические древовидные структуры. Рассматриваются алгоритмы обработки наиболее распространенных древовидных структур, таких, как сбалансированные деревья, дихотомические деревья, деревья выражений.

В главе 3 подробно рассматриваются особенности организации структур графов. Особое внимание уделено рассмотрению алгоритмов решения распространных в практике задач обхода графов и нахождения кратчайших путей в графе.

Глава 4 посвящена рассмотрению принципов организации множеств объектов с заданным отношением порядка на основе хеширования. Описываются методы выбора функции преобразования и разрешения коллизий.

Примеры программ, описывающих алгоритмы обработки структур данных различных типов, реализованы на языке C# версии 3.0 с использованием Visual Studio 2008 .NET Framework 3.5.

Программы, реализующие алгоритмы обработки структур данных различных типов, соответствуют современному уровню информационных технологий. Разделы, рассмотренные в пособии, имеют большое учебно-методическое значение и необходимы при самостоятельной работе студентов во время выполнения ими домашних заданий, лабораторного практикума и курсовой работы.

В конце каждой главы приведены упражнения и контрольные вопросы, с помощью которых можно проверить усвоение изложенного материала.

В учебном пособии содержатся сведения, недостаточно освещенные в учебной литературе, а также реализуются новые методики преподавания, активизирующие самостоятельную работу студентов.

Учебное пособие может быть полезно широкому кругу читателей, практикующихся в области компьютерных наук.

ВВЕДЕНИЕ

Для описания сложных отношений между объектами используются иерархические нелинейные структуры данных.

Деревья представляют собой иерархическую структуру некоторого множества объектов. Древовидные структуры широко используются для организации хранения информации в системах управления базами данных и для представления синтаксических структур в компиляторах программ. Специальные методы представления множеств основаны на использовании структур деревьев двоичного поиска, нагруженных и сбалансированных деревьев.

Во многих задачах, возникающих в технических дисциплинах, в математике и в компьютерных науках часто требуется наглядно представить отношения между какими-либо объектами. Наиболее адекватной моделью для представления таких отношений являются ориентированные и неориентированные графы. В пособии рассматриваются основные структуры данных, которые применяются для представления ориентированных и неориентированных графов, а также описываются основные алгоримы определения связности ориентированных графов, построения минимальных остовных деревьев и нахождения кратчайших путей в графе.

Существуют методы организации упорядоченных множеств объектов, для которых продолжительность поиска не зависит от количества объектов. При этом объекты организованы в виде таблицы, а для определения местоположения каждого объекта следует использовать ключ. К подобным методам относится хеширование

Две части пособия, посвященные рассмотрению линейных и нелинейных структур данных, охватывают широкий спектр представления объектов различной природы, используемых при решении разнообразных практических задач.

1. РЕКУРСИВНЫЕ АЛГОРИТМЫ ОБРАБОТКИ СТРУКТУР ДАННЫХ

1.1. Итерация и рекурсия в программировании

1.1.1. Понятие рекурсии

Рекурсия – есть метод определения множества объектов или процесса в терминах самого себя.

Рекурсивные определения

Любое рекурсивное определение содержит две части: ***базисную часть*** и собственно ***рекурсивную***. Например, понятие нечетного целого числа определяется следующим образом:

- базисная часть: число 1 является нечетным целым числом;
- рекурсивная часть: если какое либо число К является нечетным целым числом, то нечетными целыми будут числа, определяемые выражениями К-2 и К+2.

Это определение состоит из двух независимых частей: базисной (или базы) и рекурсивной. Базисная часть является нерекурсивным утверждением, которое задает определение для некоторой фиксированной группы объектов. Рекурсивная часть определения записывается таким образом, чтобы при цепочке повторных применений утверждение из рекурсивной части приводилось бы к базисной части. Таким образом, базисная часть задает один или более случаев, которые удовлетворяют определению, а рекурсивная часть показывает, как применить определение, чтобы проверить, удовлетворяют ли ему другие случаи.

Например, докажем, что число К = 7 является нечетным целым числом. Для этого применим рекурсивную часть определения:

число К = 7 является нечетным целым числом, если нечетным целым является число К - 2 = 7 - 2 = 5;

число К = 5 является нечетным целым числом, если нечетным целым является число К - 2 = 5 - 2 = 3;

число К = 3 является нечетным целым числом, если нечетным целым является число К - 2 = 3 - 2 = 1;

число К = 1 является нечетным целым числом. Таким образом, рекурсивное утверждение удалось привести к базе, следовательно, первоначальное утверждение о том, что число К = 7 - нечетное целое число является истинным.

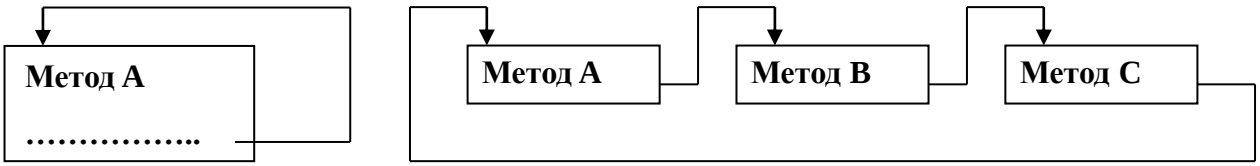
Рекурсивные алгоритмы

Рекурсивный алгоритм реализует какое-либо рекурсивное определение посредством разбиения решаемой задачи на подзадачи меньшей размерности, выполняемые с помощью одного и того же алгоритма. Процесс

разбиения завершается при достижении простейших возможных решаемых задач минимальной размерности, которые называются условиями завершения. Таким образом, рекурсивное определение алгоритма включает две части:

- ***условия завершения*** (одно или несколько), которые могут быть вычислены для определенных параметров. Условия завершения соответствуют базисной части рекурсивного определения;
- ***шаг рекурсии***, в котором текущие значения некоторых переменных в алгоритме могут быть определены с использованием их предыдущих значений. В конечном итоге шаг рекурсии должен приводить к выполнению условий завершения.

Рекурсивные алгоритмы реализуются через ***рекурсивные методы (функции)***. Рекурсивным называется метод, который в процессе выполнения явно или неявно вызывает сам себя. ***Прямая (явная) рекурсия*** характеризуется наличием в теле метода оператора обращения к нему же самому (рис. 1.а). В случае ***косвенной (неявной) рекурсии*** один метод обращается к другому, который (возможно через цепочку вызовов других методов) вновь обращается к первому методу (рис. 1.б). Далее будем рассматривать только прямую рекурсию.



а **б**

Рис. 1. Схема прямой и косвенной рекурсии:
а – прямая рекурсия; б – косвенная рекурсия

1.1.2. Итеративная и рекурсивная схема организации вычислительного процесса

Для того чтобы лучше понять особенности рекурсивных алгоритмов, полезно сопоставить итеративную и рекурсивную огранизацию процесса вычислений в программе. Особенности итеративного и рекурсивного вычислительного процесса рассмотрим на примере вычисления значения факториала некоторого натурального числа N.

Итеративная схема организации вычислительного процесса

Итеративный процесс можно проиллюстрировать с помощью схемы, приведенной на рис. 2. Этот процесс состоит из четырех блоков: инициализации, принятия решения (о продолжении вычислений), вычисления и модификации.

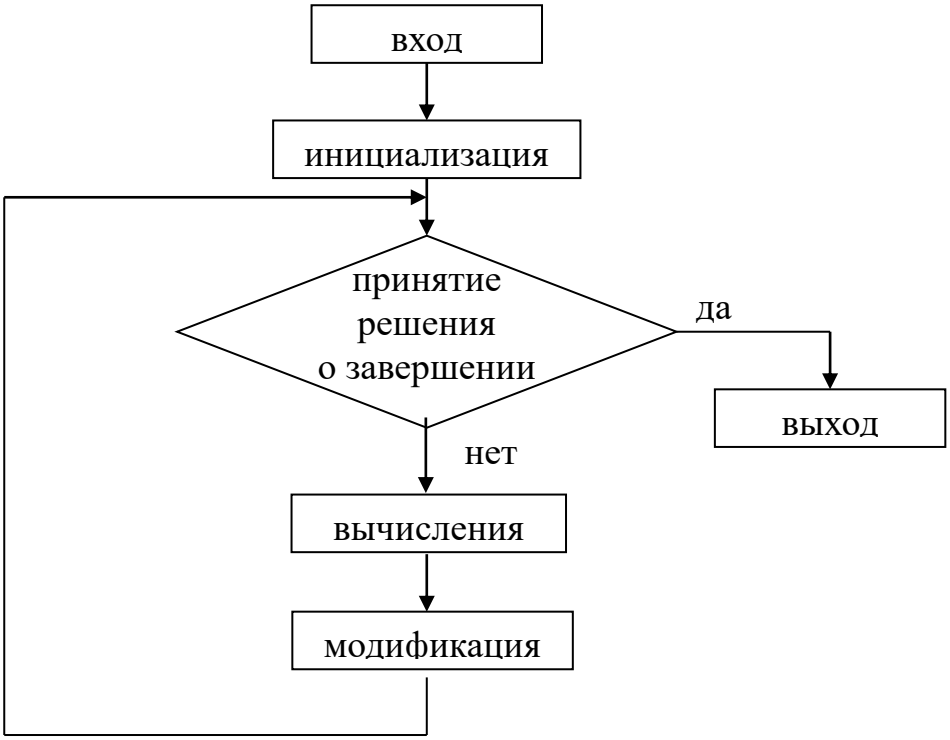


Рис. 2. Схема итеративного вычислительного процесса

В основе итеративного вычислительного процесса лежит ***итеративный цикл*** *While (с предусловием или постусловием)*, *For*. Наиболее универсальным является цикл *While*:

While < условие цикла > < тело цикла >;

Итеративная схема вычисления факториала:

N! = 1 * 2 * 3 * … * N.

Метод, реализующий итеративную схему вычисления факториала, приведен ниже:

```
using System;
namespace Iteration
{
  class Iteration
  {
     public static long Iter_fact( int n )
    {
       int i =1;  long f =1;                          // инициализация
       while ( i < = n )                              // решение о завершении
      {
        f = f * i;                                    // вычисления
```

```
        i++;                                                    // модификация
      }
      return f;
    }

    static void Main()
    {
      Console.Write( “Введите целое число:  “ );
      int x = int.Parse( Console.ReadLine() );
      long y = Iter_fact( x);
      Console.WriteLine( “Итеративный факториал = {0}”, y );
    }
  }
}
```

Существует два важных положения, известных в математике и в программировании, определяющих соотношение между итерацией и рекурсией:

1. любой итеративный цикл может быть заменен рекурсией;
2. рекурсия не всегда может быть заменена итерацией.

Рекурсивная схема организации вычислительного процесса

Общая схема рекурсивного вычислительного процесса представлена на рис. 3.

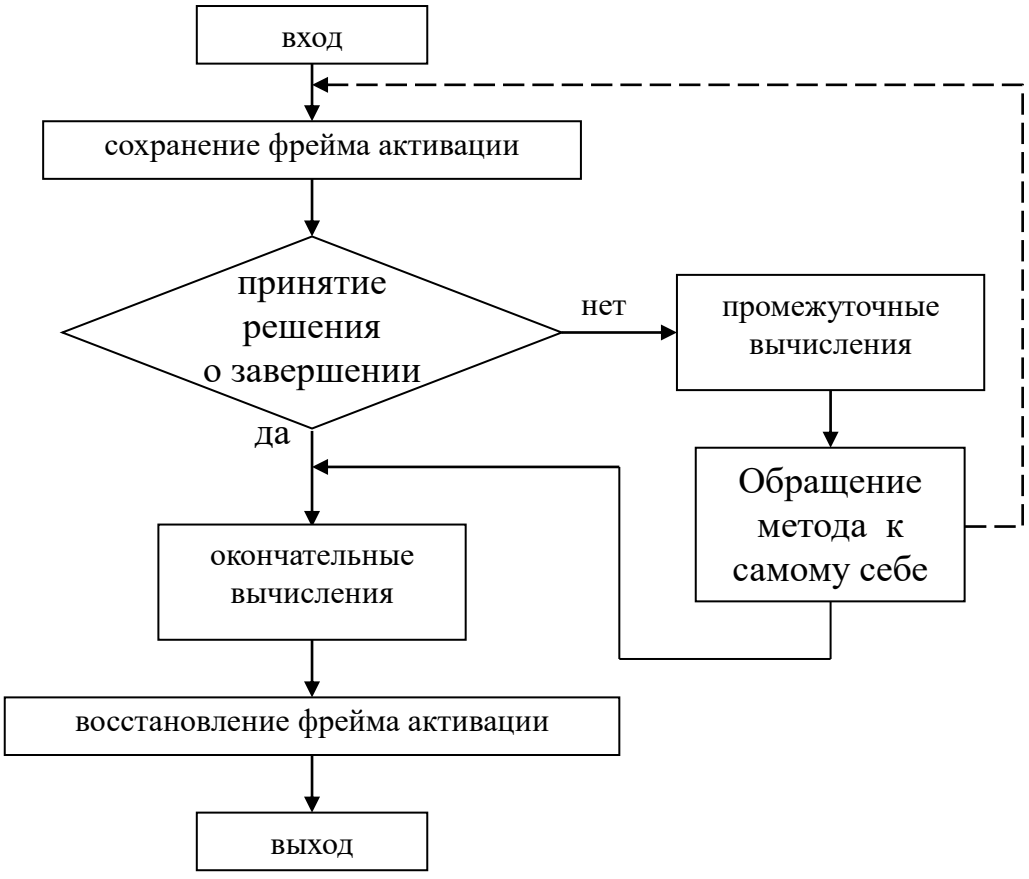


Рис. 3. Схема рекурсивного вычислительного процесса

Так как обращаться к рекурсивному методу можно как из него самого, так и извне, каждое обращение к рекурсивному методу вызывает его независимую активацию. При каждой активации образуются копии всех локальных переменных и формальных параметров рекурсивного метода, в которых “оставляют следы” операторы текущей активации. Таким образом, для рекурсивного метода может одновременно существовать несколько активаций. Для обеспечения правильного функционирования рекурсивного метода необходимо сохранять адреса возврата в таком порядке, чтобы возврат после завершения каждой текущей активации выполнялся в точку, соответствующую оператору, непосредственно следующему за оператором рекурсивного вызова. Совокупность локальных переменных, значений фактических параметров, подставляемых на место формальных параметров рекурсивного метода, и адреса возврата однозначно характеризует текущую активацию и образует ***фрейм активации***. Фрейм активации необходимо сохранять при очередной активации и восстанавливать после завершения текущей активации.

В блоке принятия решения (о продолжении вычислений) производится проверка, являются ли значения входных параметров такими, для которых возможно вычисление значений выходных параметров в соответствии с базисной частью рекурсивного определения. На основании этой проверки принимается решение о выполнении промежуточных или окончательных вычислений. Блок промежуточных вычислений можно объединить с блоком обращения к методу, если промежуточные вычисления очень просты. В блоке окончательных вычислений производится явное определение параметров-переменных метода для конкретных значений входных параметров, соответствующих текущей активации метода.

В основе рекурсивного вычислительного процесса лежит ***рекурсивный цикл***, который реализуется через вызов рекурсивного метода, причем каждая активация рекурсивного метода эквивалентна одному проходу итеративного цикла *While*.

Общая схема рекурсивного цикла:

```
[спецификаторы] тип Recursion_Cycle (...)
{
  if  < условие цикла >
  {
     < тело рекурсивного цикла; >
     Recursion Cycle (…);
   }
}
```

В теле рекурсивного цикла (в блоке промежуточных вычислений) обязательно должны содержаться операторы, изменяющие значения переменных, от которых зависит условие завершения рекурсивного цикла. Напомним, что выполнение условия завершения рекурсивного цикла

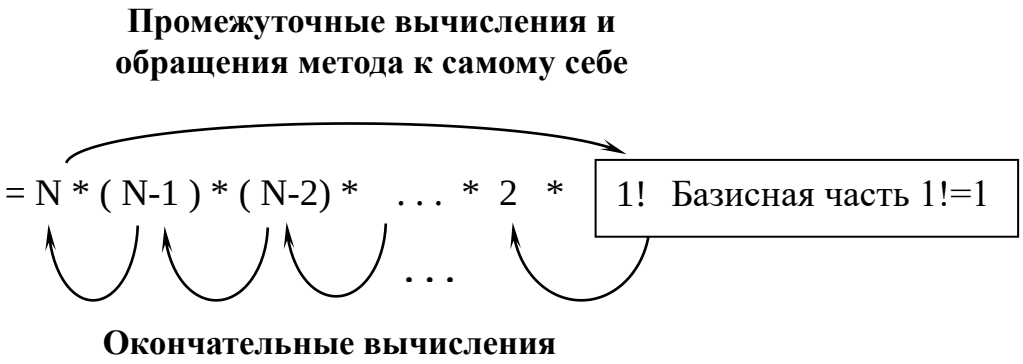
соответствует достижению базиса рекурсивного определения. Если значения этих переменных не успевают измениться в теле цикла до очередной активации рекурсивного метода, то возникает ***бесконечный рекурсивный цикл***.

Общая схема бесконечного рекурсивного цикла:

```
[спецификаторы] тип Infinite Recursion_Cycle (...)
{
  if  < условие цикла >
  {
    Infinite Recursion_Cycle (…);
   < тело рекурсивного цикла; >
  }
}
```

Рекурсивная схема вычисления факториала:

Базисная часть:
0! = 1; 1! = 1;
Рекурсивная часть:
N! = N * (N-1)! = N * (N-1) * (N-2)! = N * (N-1) * … * (N – (N-1))! =



Метод, реализующий рекурсивную схему вычисления факториала:

```
using System;
namespace Recursion
{
  class Recursion
  {
     public static long Recurs_fact( int n )
     {
       long f;
       if  ( n == 0 ||  n == 1)              // принятие решения о завершении  вычислений:
         f = 1;                          // да – окончательные вычисления для базисной части
       else
       {
```

```
        f = Recurs_fact ( n-1);          // нет – промежуточные вычисления и
                                            обращение метода к самому себе
        f = f * n;              // окончательные вычисления для рекурсивной части { 1 }
       }
        return f;
     }                                                                  // { 2 }

    static void Main()
    {
      Console.Write( “Введите целое число:  “ );
      int x = int.Parse( Console.ReadLine() );
      long y = Recurs_fact( x );
      Console.WriteLine( “Рекурсивный факториал = {0}”, y );
    }
  }
}
```

{ 1 } – адрес возврата после завершения активации,

{ 2 } – завершение активации.

С каждым обращением к рекурсивному методу ассоциируется ***номер уровня рекурсии*** (номер фрейма активации). Считается, что при первоначальном вызове рекурсивного метода из основной программы номер уровня рекурсии равен единице. Каждый следующий вход в метод имеет номер уровня на единицу больше, чем номер уровня метода, из которого производится это обращение.

Другой характеристикой рекурсивного метода является ***глубина рекурсии***, определяемая максимальным уровнем рекурсии в процессе вычисления при заданных аргументах. В общем случае эта величина неочевидна, исключение составляют простые рекурсивные методы, например, при вычислении значения N! глубина рекурсии равна N.

Так как при выходе из текущей активации самым первым должен быть восстановлен фрейм, который был позже всех сохранен, для хранения фреймов используется область системного стека ([11]: с.31). Таблица 1 и рис. 4 поясняют механизм рекурсивных вычислений.

Таблица 1. Трасса вычисления 3!

№ уровня рекурсии	Знач-е вход. пар-ра n	Содержимое стека		Знач-е выход. пар-ра F	Выход из уровня
		До вызова N,F,(*)возврата	После вызова N,F,(*)возврата		
1	3	–,–,–	3, □ ,(*1*)		
2	2	3, □ ,(*1*)	2, □ ,(*1*) 3, □ ,(*1*)		
3	1	2, □ ,(*1*) 3, □ ,(*1*)	1, □ ,(*1*) 2, □ ,(*1*) 3, □ ,(*1*)		
		До возврата из уровня	После возврата из уровня		
3	1	1, □ ,(*1*) 2, □ ,(*1*) 3, □ ,(*1*)	2, □ ,(*1*) 3, □ ,(*1*)	1	(*2*)
2	2	2, □ ,(*1*) 3, □ ,(*1*)	3, □ ,(*1*)	2	(*2*)
1	3	3, □ ,(*1*)	–,–,–	6	(*2*)

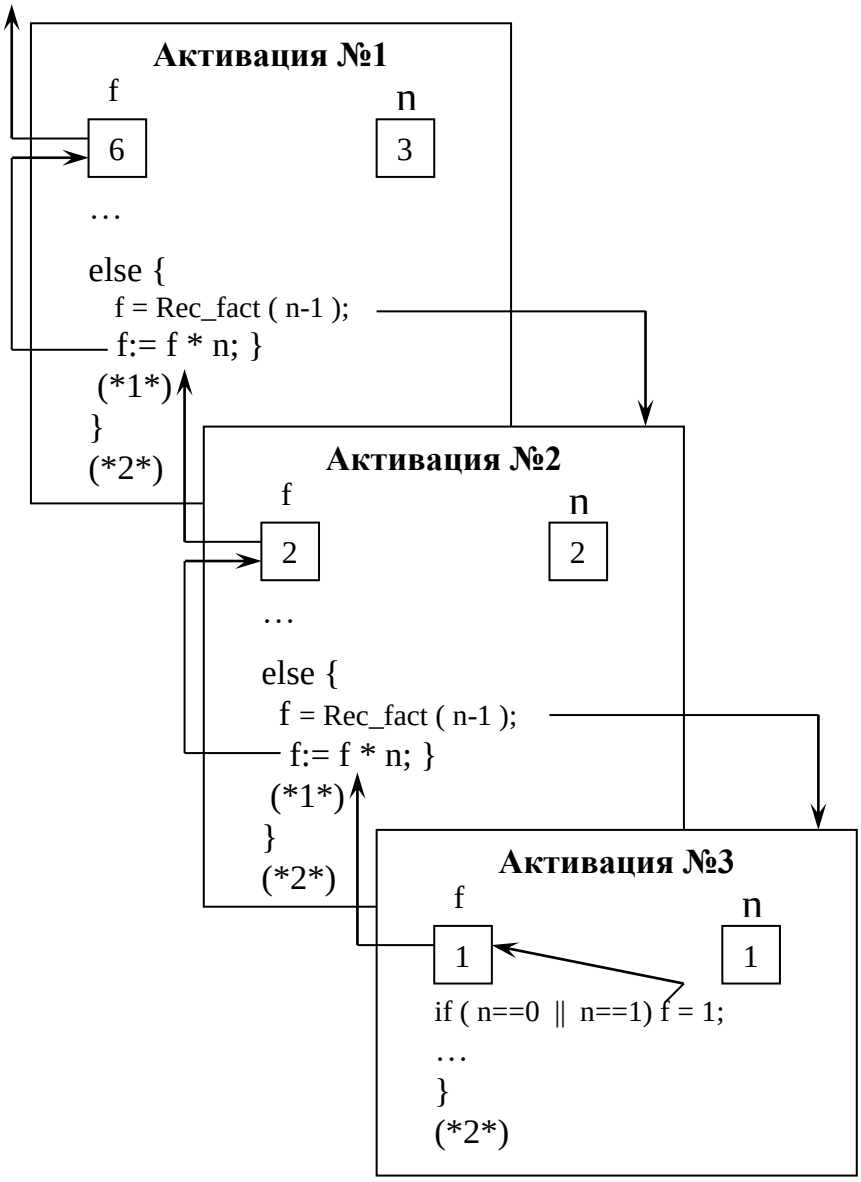


Рис. 4. Фреймы активации при вычислении 3!

1.2 Виды рекурсивных алгоритмов

1.2.1. Вычислительные алгоритмы

а) Вычисление n-го по счету члена числовой последовательности.

Примером числовой последовательности являются числа Фибоначчи. Числа Фибоначчи составляют последовательность, очередной элемент которой вычисляется по двум предыдущим значениям:

$$F_n = F_{n-1} + F_{n-2}.$$

Нулевое и первое значения должны быть заданы и равны единице. Последовательности такого рода применяются, например, в программных генераторах случайных чисел:

```
using System;
namespace Fibonacci;
{
  class Fibonacci
  {
    public static long Fib( int n )
    {
      if ( n == 0  ||  n == 1 ) return 1;                    // условие завершения
      else return  Fib( n-2 ) + Fib( n-1 )                   // шаг рекурсии
     }

   static void Main()
   {
     Console.WriteLine("Введите значение n ");
     int n = int.Parse( Console.ReadLine() );
     Console.WriteLine("{0}", Fib( n )  );
    }
  }
}
```

Каждое обращение при $n > 1$ приводит к двум дальнейшим обращениям, т.е. общее число обращений растет экспоненциально. Очевидно, что числа Фибоначчи более эффективно можно вычислять по итеративной схеме.

Однако существуют такие числовые последовательности, для которых применение рекурсии – единственный способ решения. Например:

$$a(1) = 1;\ a(n) = n – a (a (n-1)),\ n > 1.$$

б) Вычисление значений полиномов порядка n в заданной точке x. Например, полиномы Чебышева определяются следующим образом:

$$T_0 (x) = 1; \qquad T_1 (x) = x; \qquad T_n (x) = 2xT_{n-1} (x) – T_{n-2} (x).$$

Таким образом, текущее значение полинома определяется по двум предыдущим значениям.

в) Решение разностных уравнений. Рассмотрим, например, уравнение следующего вида:

$$A(0) = 1; \qquad A(n) = A(n \text{ div } 2) + A (n \text{ div } 3).$$

Простого и очевидного способа итерационного программирования функции *A(n)* не существует. Решение разностных уравнений выполняется с помощью рекурсии.

г) Сортировки.

В качестве примера рассмотрим ***алгоритм быстрой сортировки.***

1. Выбрать в массиве некоторый элемент, называемый опорным элементом, желательно, чтобы опорный элемент разбивал множество элементов массива на две примерно равные части.
2. Выполнить операцию разделения массива таким образом, чтобы все элементы, меньшие или равные опорного элемента, оказались слева от него, а все элементы, большие опорного – справа от него. Это достигается путем просмотра массива попеременно с обоих концов, причем каждый элемент сравнивается с опорным. Элементы из левой части, не меньшие опорного, меняются местами с элементами из правой части, не большими опорного. После завершения процедуры разделения опорный элемент оказывается на своем окончательном месте. Все элементы, которые меньше опорного, будут стоять слева от него, а все элементы, которые больше опорного, - справа от него.
3. Выполнить пункты 1 и 2 над частями массива слева и справа от опорного элемента.

Базой рекурсии являются массивы, состоящие из одного или двух элементов, которые уже отсортированы. Алгоритм всегда завершается, поскольку за каждую итерацию по крайней мере один элемент размещается на его окончательном месте.

д) Бинарный поиск элемента с заданным значением в упорядоченном множестве объектов (массиве).

При бинарном поиске берется некоторый ключ и просматривается упорядоченный массив из *N* элементов на предмет совпадения с этим ключом (элементы индексируются от *1* до *N*). Результатом поиска является индекс совпавшего с ключом элемента или *0* при отсутствии такового. Алгоритм бинарного поиска может быть описан рекурсивно. Пусть упорядоченный массив *A* характеризуется нижним индексом *low* и верхним индексом *high*. Поиск совпадения с ключом начинается в середине массива (индекс *mid*):

mid = (low + high) / 2; сравнить *A[mid]* с ключом.

Если совпадение произошло, условие останова достигнуто, что позволяет прекратить поиск и возвратить индекс *mid*. Если совпадения не происходит, можно воспользоваться тем фактом, что массив упорядочен, и ограничить диапазон поиска “нижним подмассивом” (слева от *mid*) или “верхним подмассивом” (справа от *mid*).

Если ключ меньше *A[mid]*, совпадение может произойти только в левой половине массива в диапазоне индексов от *low* до *mid – 1*. Если ключ больше *A[mid]*, совпадение может произойти только в правой половине массива в диапазоне индексов от *mid + 1* до *high*.

Шаг рекурсии направляет бинарный поиск для продолжения в один из подмассивов. Рекурсивный процесс просматривает массивы все меньшего размера. Поиск заканчивается неудачей, если подмассивы исчезли, т.е. если нижняя граница превосходит верхнюю. Условие *low > high* – второе условие останова рекурсивного процесса. В этом случае алгоритм возвращает *0*.

1.2.2. Перебор с возвратами

Перебор с возвратами (бектрекинг - back tracking – обратное следование) – это способ поиска решения, когда при возврате после рассмотрения "пробного" варианта решения на один шаг назад все переменные программы восстанавливают свои значения. Классическим примером перебора с возвратами является ***алгоритм прохождения лабиринта***, идея которого состоит в следующем. Лабиринт есть множество перекрестков, связанных между собой. Предположим, что каждый перекресток имеет три атрибута, определяющих возможность перемещения налево, прямо и направо. Если значение некоторого атрибута равно единице, то движение в данном направлении возможно. Нуль показывает, что движение в данном направлении заблокировано. Три нулевых значения атрибутов перемещения из некоторого перекрестка представляют тупик. Проход по лабиринту считается успешным при достижении точки "Выход".

Процесс поиска выхода включает ряд рекурсивных шагов. В каждом перекрестке необходимо исследовать возможные варианты. Если возможно, сначала следует пойти налево. Достигнув очередного перекрестка, необходимо снова рассмотреть варианты и попытаться пойти налево. Если выбор левого направления заводит в тупик, следует отступить на один шаг назад и выбрать движение прямо, если это направление не заблокировано. Если этот выбор заводит в тупик, вновь необходимо отступить на один шаг и пойти направо. Если все альтернативы движения из данного перекрестка ведут в тупики, следует вернуться к предыдущему перекрестку и сделать новый выбор. Если произошел возврат к исходной точке, и из нее нет новых путей, задача поиска выхода из лабиринта не имеет решения. Рассмотрим варианты в лабиринте, изображенном на рис. 5.

Перекресток	Выбор	Результирующий перекресток
1	прямо	2
2	налево	3
3	налево	7
7	тупик: вернуться к 3	3
3	прямо	4
4	прямо	6
6	тупик: вернуться к 4	4
4	направо	5

5	тупик: вернуться к 4, 3, 2	2
2	прямо	8
8	налево	9
9	прямо	10 - выход

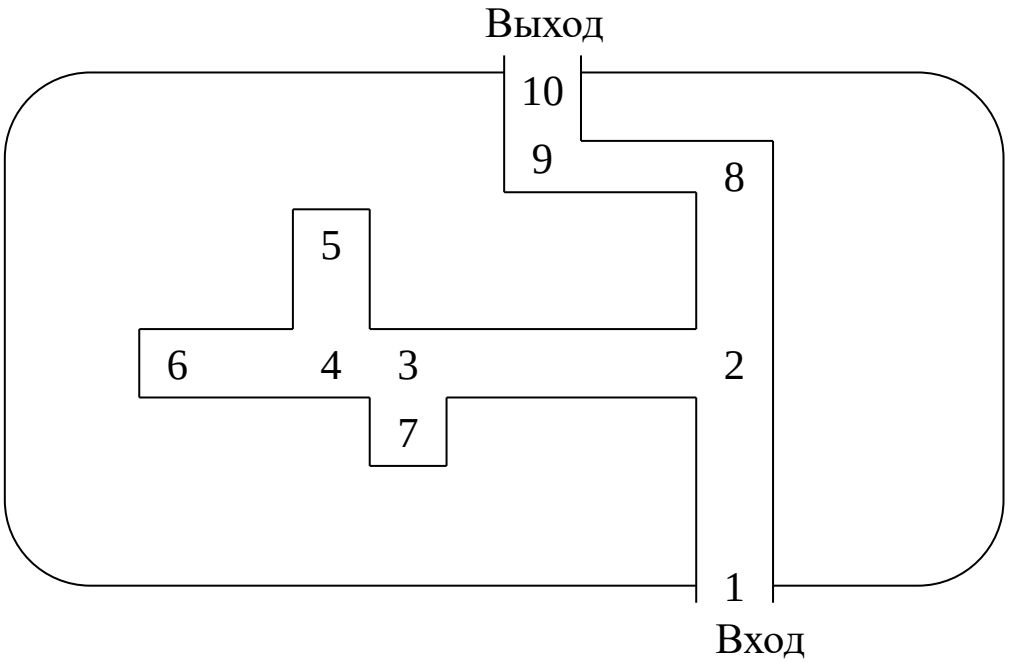


Рис. 5. Лабиринт

С помощью алгоритма бектрекинга решаются разнообразные ***шахматные задачи***. Наиболее известными являются следующие из них:

- задача о расстановке восьми ферзей. Требуется найти все возможные способы расстановки восьми ферзей на шахматной доске так, чтобы ни один из них не нападал на любого другого;
- задача о расстановке ладей. Требуется определить все возможные способы расстановки *n* ладей на шахматной доске размером *n* * *n* так, чтобы они не угрожали друг другу;
- задача обхода шахматной доски размером 8 * 8 ходом коня так, чтобы конь побывал на каждом поле один раз.

1.2.3. Комбинаторика

Рекурсия находит широкое применение в комбинаторике – разделе математики, занимающемся подсчетом объектов.

Перестановка из N элементов (1,2, …, N) есть упорядоченное расположение этих элементов. Для N = 3 (1 3 2), (3 2 1), (1,2,3) – различные перестановки. В комбинаторике установлено, что число перестановок равно N!. Для позиции 1 существуют N вариантов, т.к. все N элементов доступны. Для позиции 2 имеются N – 1 вариантов, т.к. один элемент зафиксирован в позиции 1. Число вариантов уменьшается на 1 по мере продвижения по позициям. Общее число перестановок есть произведение числа вариантов в каждой позиции.

$$Permulation(N) = N * (N-1) * (N-2) * \ldots * 2 * 1 = N!$$

Рекурсивный алгоритм генерирует перечень всех перестановок из N элементов для N > 1. Сформируем 24 (4!) перестановки из четырех элементов. Перестановки с одинаковыми первыми элементами записываются в отдельный столбец. Каждый столбец затем делится на три пары с одинаковыми вторыми элементами:

1	**2**	**3**	**4**
1 2 3 4	2 1 3 4	3 1 2 4	4 1 2 3
1 2 4 3	2 1 4 3	3 1 4 2	4 1 3 2
1 3 2 4	2 3 1 4	3 2 1 4	4 2 1 3
1 3 4 2	2 3 4 1	3 2 4 1	4 2 3 1
1 4 2 3	2 4 1 3	3 4 1 2	4 3 1 2
1 4 3 2	2 4 3 1	3 4 2 1	4 3 2 1

Алгоритм вычисления числа перестановок иллюстрируется иерархическим деревом (рис. 6), которое содержит упорядоченные пути, соответствующие перестановкам. В исходном положении имеется четыре варианта – 1, 2, 3 и 4, соответствующие четырем столбцам. По мере продвижения вниз по дереву нижние уровни разделяются на 3, 2 и 1 элемент, соответственно. Алгоритм генерации всех перестановок моделирует проход по путям дерева. Продвигаясь от уровня к уровню, мы тем самым указываем очередную позицию в перестановке. Этот процесс представляет собой шаг рекурсии.

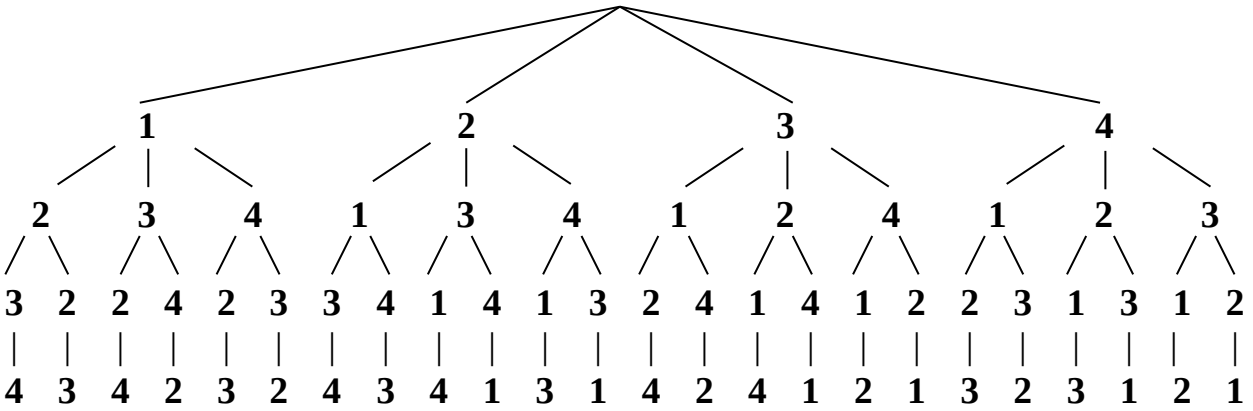


Рис. 6. Иллюстрация работы алгоритма перестановок

Шаг рекурсии выполняется следующим образом. Представим перестановку в виде массива из N элементов. Индекс элемента массива – это номер позиции перестановки. В каждом элементе массива хранится значение, соответствующее позиции перестановки, определяемой индексом элемента.

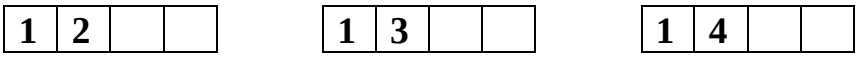
Итерационный процесс проверяет все возможные значения первого элемента перестановки. В нашем примере существует N=4 возможных значения первого элемента:

Индекс 1:

1 1 1 1

На следующем уровне дерева каждый узел дает начало N-1 перестановкам, содержащим N-1 отличных от первого элементов. Например, узел 1 соответствует перестановкам, которые начинаются с единицы в первой (индекс 1) позиции. Путь, ведущий к узлу 2 следующего уровня, соответствует перестановкам со значениями 1 и 2 в первых двух позициях и т.д. Можно итерационно определить второй элемент перестановки перебором узлов 2, 3 и 4:

Индекс 2:



1 2 1 2 1 2

На следующем уровне дерева каждый узел разветвляется на два пути, которые представляют перестановки из двух элементов. Например, если во второй позиции (индекс 2) находится элемент 2, в третьей позиции (индекс 3) могут быть только элементы 3 или 4:

Индекс 3:

1 2 3 1 2 3

Так как третий элемент перестановки зафиксирован, последний элемент определяется однозначно, поскольку перестановка не допускает повторяющиеся значения:

Индекс 4:

1 2 3 4 1 2 3 4

Определение всех N элементов перестановки является ***условием завершения*** работы рекурсивного алгоритма.

Алгоритм перестановок можно запрограммировать, оперируя массивом A[1] ,…, A[n], в котором хранится перестановка чисел 1..n. Рекурсивный метод [11] распечатывает все перестановки (t позиций), в которых зафиксирована первая позиция.

```
using System;
namespace Permulation
{
```

```
class Permulation
{
  public static void Generate( int A[], int t )          // Перестановки чисел 1..N
  {
    int i, j, k;
    if  ( t == A.Length-1 )                               // Условие завершения шага
    {                                                     // Перестановка готова
      for ( i=0; i<A.Length; i++ ) Console.Write( "{0}", A[i]);
      Console.ReadLine(); Console.WriteLine();
    }
  else
    for ( j = t+1; j<A.Length; j++)                       // Формирование перестановки: t < N
    {
      k = A[t+1]; A[t+1] = A[j]; A[j] = k;                // Поменять местами a[t+1] и a[j]
      Generate( A, t+1 );
      k = A[t+1]; A[t+1] = A[j]; A[j] = k;                // Поменять местами a[t+1] и a[j]
    }
  }

 static void Main()
 {
   Console.WriteLine("Введите целое число: ");
   int n = int.Parse( Console.ReadLine() );
   int[] Array = new int[n];                              // Массив для хранения перестановок
    for ( int i=0; i<A.Length; i++) Array[i] = i;         // Заполнение массива значениями
   int t = 0;
   Generate( Array,t );
  }
 }
}
```

1.2.4. Игры и головоломки: задача о "ханойских башнях"

Согласно легенде, у жрецов храма Брахмы в горах Тибета есть медная платформа с тремя алмазными стержнями A, B и C. На одном стержне A нанизано 64 золотых диска, каждый из которых немного меньше того, что под ним. Конец света наступит, когда жрецы переместят диски со стержня A на стержень C. Задача имеет следующие условия:

- за один раз можно перемещать только один диск,
- больший диск нельзя класть на меньший,
- снятый диск нельзя отложить, его необходимо сразу же надеть на другой стержень.

Несомненно, жрецы все еще работают, т.к. задача включает в себя $2^{64} - 1$ ходов. Если тратить по одной секунде на ход, то потребуется примерно 500 миллиардов лет.

Решение данной задачи носит рекурсивный характер. Задача разбивается на последовательность подзадач одного и того же типа, но меньшей размерности. Условием завершения является выполнение простой задачи перемещения одного диска. Сформулируем алгоритм решения данной задачи для N дисков. Обозначим стержни: начальный (*start*), промежуточный (*temp*), конечный или целевой (*destination*). На начальный стержень нанизано N дисков в порядке возрастания размера, т.е. самый большой диск лежит внизу. Цель задачи – переместить все N дисков с начального стержня на конечный, следуя определенным выше условиям, и распечатать перечень ходов. Предполагается, что промежуточный стержень будет использоваться для временного хранения дисков.

Иллюстрация решения задачи для N = 1 диска приведена на рис. 7, для N = 2 дисков - на рис. 8, для N = 3 дисков – на рис. 9. Алгоритм требует 2^N – 1 ходов.

Если N = 1, выполняется условие завершения (базисное условие рекурсивного алгоритма), которое может быть обработано путем перемещения единственного диска с начального стержня на конечный и распечатки соответствующего хода. Для N > 1 выполняется трехшаговый процесс перемещения N дисков с начального стержня на конечный, причем стержень A является начальным (*A – start*), стержень B – промежуточным (*B – temp*), стержень C – конечным (*C – dest*).

На первом шаге алгоритма перемещаются N – 1 дисков с начального стержня на промежуточный с использованием конечного стержня для временного хранения. При этом стержень A выполняет роль начального (*A – start*), стержень B выполняет роль конечного (*B – dest*). Конечный стержень C используется как промежуточный (*C – temp*).

На втором шаге самый большой диск просто перемещается с начального стержня на конечный: *start -> dest*.

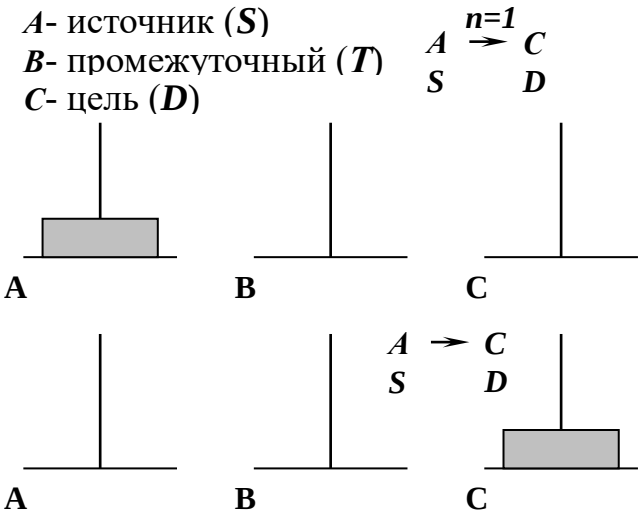


Рис. 7. Решение задачи о “ханойских башнях” для N = 1 диска

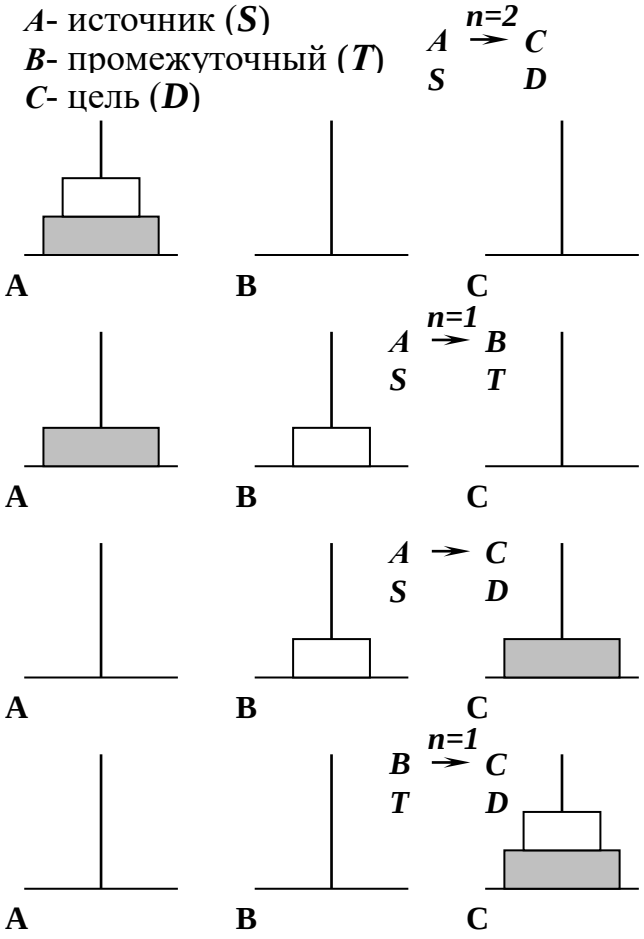


Рис. 8. Решение задачи о “ханойских башнях” для N = 2 дисков

На третьем шаге N – 1 дисков перемещаются со среднего стержня на конечный с использованием начального стержня для временного хранения. При этом стержень B выполняет роль начального (*B – start*), стержень C

выполняет роль конечного (*C – dest*). Начальный стержень A используется как промежуточный (*A – temp*).

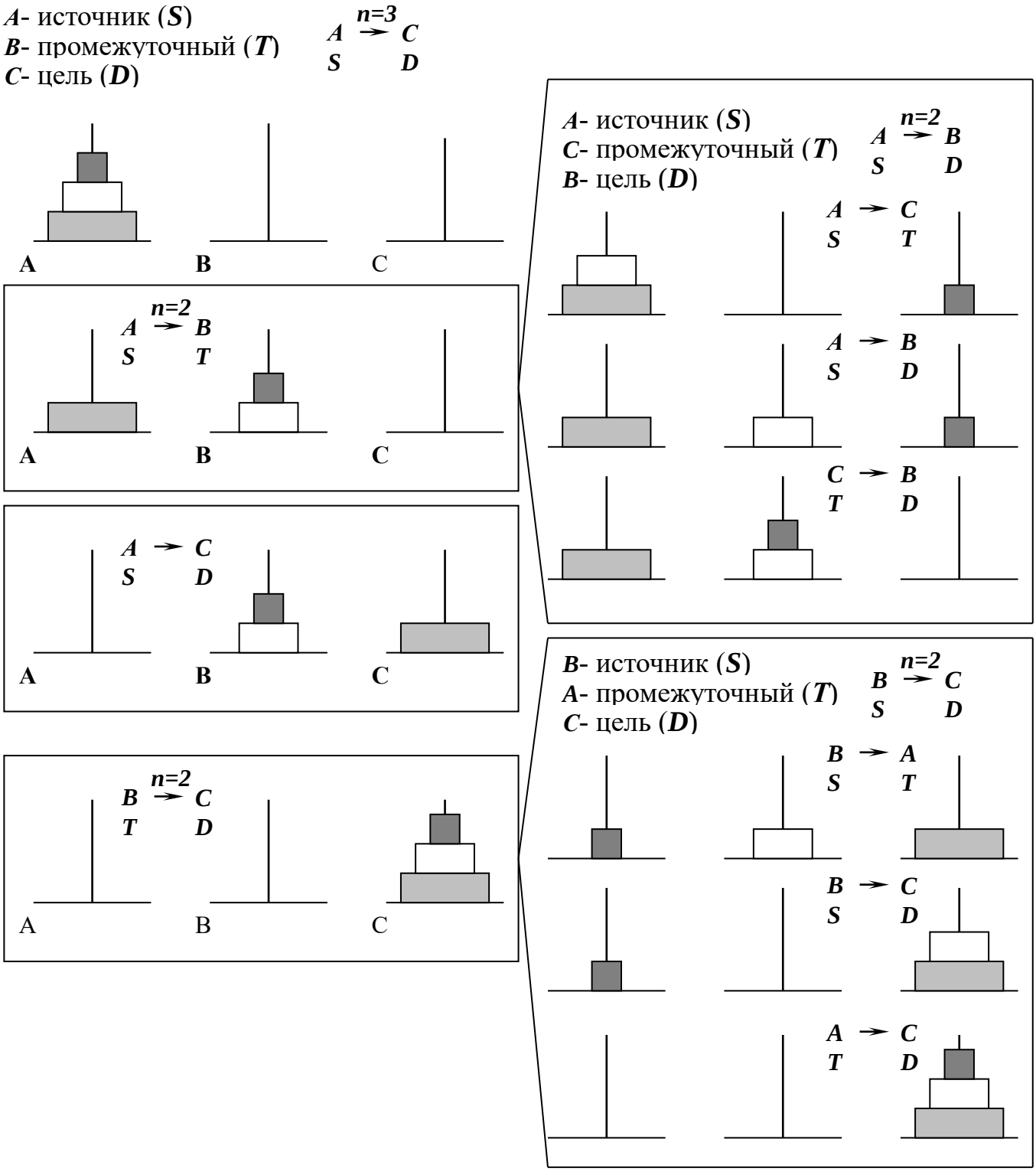


Рис. 9. Решение задачи о “ханойских башнях” для N = 3 дисков

Программная реализация алгоритма решения задачи о “ханойских башнях” следует ниже:

```
// перенести N дисков с начального стержня на конечный, используя промежуточный
//стержень для временного хранения дисков
Using System
{
  namespace Hanoi
  {
     public static void Hanoi( int n, char start, char tmp, chat dest )
     {
       if  ( n ==1 )                          // условие завершения: перемещение одного диска
         Console.WriteLine( "{0}, '-->', {1}", start, dest );
       else
       {
        // перенести N–1 дисков с начального стержня на промежуточный, используя
           конечный  стержень для временного хранения дисков
        Hanoi( n-1, start, dest, temp );
        // перенести нижний диск с начального стержня на конечный
        Console.WriteLine( "{0}, '-->', {1}", start, dest );
        // перенести N–1 дисков с промежуточного стержня на конечный, используя
           начальный  стержень для временного хранения дисков
        Hanoi( n-1, temp, start, dest )
       }
     }

    static void Main()
    {
       Console.WriteLine(" Введите количество дисков");
       int n = int.Parse( Console.ReadLine() );
       Hanoi( n, 'a', 'b', 'c');
       Console.ReadLine();
    }
  }
}
```

Более короткий вариант решения задачи о "ханойских башнях":

```
// перенести N дисков с начального стержня на конечный, используя промежуточный
//стержень для временного хранения дисков
     public static void Hanoi( int n, char start, char tmp, chat dest )
     {
       if  ( n>0 )                                                    // условие продолжения }
       {
        // перенести N–1 дисков с начального стержня на промежуточный, используя
           конечный  стержень для временного хранения дисков
        Hanoi( n-1, start, dest, temp );
        // перенести нижний диск с начального стержня на конечный
        Console.WriteLine( "{0}, '-->', {1}", start, dest );
```

```
        // перенести N–1 дисков с промежуточного стержня на конечный, используя
           начальный  стержень для временного хранения дисков
        Hanoi( n-1, temp, start, dest )
      }
    }
```

1.2.5. Арифметические выражения – пример рекурсивной структуры данных

При описании арифметических выражений рекурсия означает возможность вложенности, т.е. использование в выражениях в качестве операндов подвыражений, заключенных в круглые скобки. Синтаксис языков программирования принято описывать с помощью рекурсивной нотации, называемой формой Бэкуса-Наура (БНФ). Она состоит из правил подстановки, определяющих, каким образом нетерминальный символ (т.е. символ, требующий дальнейшего уточнения) может быть замещен другим нетерминальным или терминальным символом (т.е. символом, не требующим уточнения). В правилах подстановки используется знак «|» для разделения альтернативных подстановок. Нетерминальные символы заключаются в угловые скобки «<>». Символ «::=» означает «это есть».

Определение простейшего арифметического выражения с помощью БНФ выглядит следующим образом:

<арифметическое выражение>::=<операнд><знак операции><операнд>
<операнд> ::= <идентификатор> | (<арифметическое выражение>)
<знак операции> :: = + | - | * | /
<идентификатор> ::= a | b | … | z | A | B | … | Z

Эти правила указывают, что арифметическое выражение состоит из двух операндов, разделенных знаком операции. В свою очередь, любой операнд может представлять собой однобуквенный идентификатор или арифметическое выражение, заключенное в круглые скобки. Это означает, что арифметическое выражение определяется в терминах самого себя, следовательно, данное определение рекурсивно.

Примеры простейших арифметических выражений, соответствующих данному определению: *X+Y X*(Y+Z) (Y*Z)-X (X+Y)*(Z-W) (X/(Y+Z))*W* и т.п.

Рекурсивные алгоритмы наиболее эффективны и удобны для обработки рекурсивных структур данных.

1.3. Рекурсивные алгоритмы обработки динамических линейных структур данных на примере списков

Рекурсивное определение списка: список – это пустая структура или структура, состоящая из особого элемента (первого) и списка, на который указывает особый элемент. Рекурсивную интерпретацию линейного односвязного списка иллюстрирует рис. 10.

Как следует из рекурсивного определения списка, условием завершения его обработки является достижение пустого списка в процессе применения шагов рекурсии.

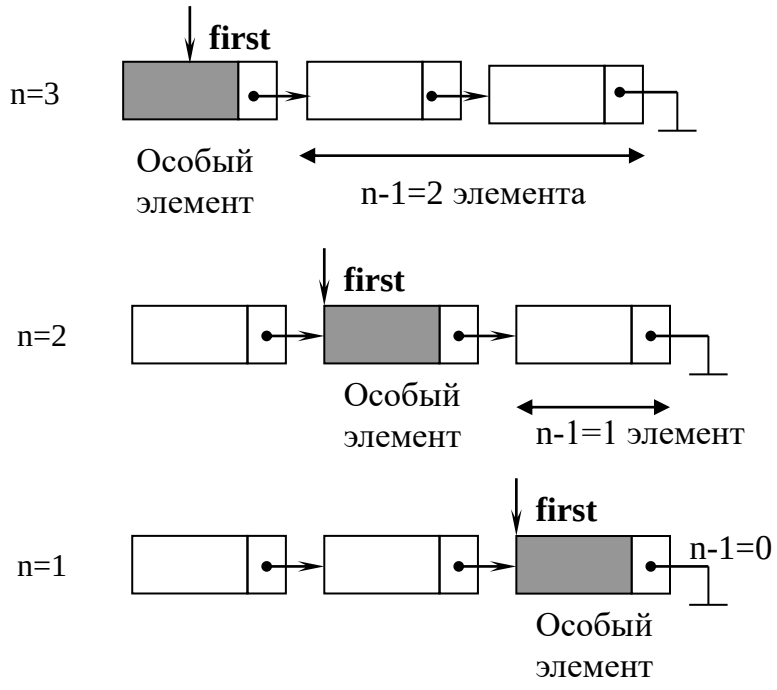


Рис. 10. Рекурсивная интерпретация списка

Рассмотрим два рекурсивных метода, один из которых распечатывает содержимое информационных полей линейного односвязного списка в прямом порядке (*Print_Front*), а второй – в обратном порядке (*Print_Back*):

```
public void Print_Front( Node first )          // распечатка содержимого списка
{                                              // в прямом направлении
  if  ( first != null )
  {
    Console.WriteLine( first.Info );
    Print_Front( first.Link )                  // хвостовая рекурсия
  }
}
```

```
public void Print_Back( Node first )                    // распечатка содержимого списка
{                                                       // в обратном направлении
  if  ( first != null )
  {
    Print_Back( first.Link )
    Console.WriteLine( first.Info );
  }
}
```

1.4. Эффективность рекурсивных вычислений

Так как для хранения фреймов активации рекурсивного метода используется стек, при обработке данных нерекурсивной природы следует использовать итеративные алгоритмы, не требующие дополнительного расхода памяти, если только рекурсивные алгоритмы не оказываются более ясными и понятными. Например, при вычислении значения факториала натурального числа N несомненно следует предпочесть итеративный метод.

Если метод содержит единственный рекурсивный вызов и он является последним действием метода, то говорят, что имеет место ***хвостовая рекурсия*** (*tail recursion*) (см. метод *Print_Front*, приведенный в разделе 1.3). Этот рекурсивный **вызов требует затрат на создание фреймов активации и запоминание** их в стеке. Когда рекурсивный вычислительный процесс доходит до условия завершения, выполняется серия возвратов, выталкивающих фреймы активации из стека. Если при наличии хвостовой рекурсии фреймы активации не используются для окончательных вычислений, следует предпочесть итеративную реализацию (см. метод *Print*, приведенный в [11]: с.52). Рекурсивный метод *Print_Back*, не содержащий хвостовой рекурсии, имеет более эффективную реализацию, чем итеративный метод, выполняющий те же действия.

1.5. Контрольные вопросы к главе 1

1. В чем заключаются особенности работы с автоматической памятью?

2. Из каких частей состоит рекурсивное определение?

3. Из каких частей состоит рекурсивный алгоритм?

4. Что такое прямая и косвенная рекурсия?

5. Сравните итеративную и рекурсивную организацию вычислительного процесса. Чем они отличаются?

6. Что такое бесконечная рекурсия? Какова причина ее возникновения?

7. Что такое уровень рекурсии? Что такое глубина рекурсии?

8. Каким образом используются фреймы активации в процессе работы рекурсивного алгоритма?

9. Что такое бектрекинг? Как реализуются алгоритмы поиска с возвратом?

10. В чем особенности рекурсивной реализации комбинаторных алгоритмов?

11. Что такое “хвостовая” рекурсия? Почему при реализации алгоритмов ее следует избегать?

1.6. Упражнения к главе 1

1. Реализуйте рекурсивный алгоритм вычисления последовательности *n* вложенных корней

$$x(m,i)=\sqrt{m+\sqrt{m+...+\sqrt{m}}}, \quad m\ge 0,\ i=\overline{1,...,n}.$$

2. Реализуйте рекурсивный алгоритм вычисления суммы *n* слагаемых. *i* – количество синусов в каждом слагаемом.

$$y(x,i)=\sin x+\sin(\sin x)+...+\sin(\sin...(\sin x)),\quad i=\overline{1,...,n}.$$

3. Реализуйте рекурсивный алгоритм вычисления суммы n первых членов ряда

$$x+\frac{x^3}{2!}+\frac{x^5}{3!}+\frac{x^7}{4!}+...+\frac{x^{2n-1}}{n!}+...$$

4. Пусть в алгебраической записи выражения имеется единственная операция – умножение, обозначаемое обычным образом (два сомножителя следуют непосредственно друг за другом). Выражение состоит из строки символов и скобок- ограничителей: ( ) [ ] { }. Реализуйте рекурсивный алгоритм, выполняющий проверку соответствия открывающих и закрывающих скобок. Например, запрещены выражения вида (ab] или a(b[c])d].

5. Задана строка S из N символов. Реализуйте рекурсивный алгоритм проверки, является ли симметричной часть строки, начинающаяся i-м и заканчивающаяся j-м ее элементом.

6. Реализуйте рекурсивный алгоритм, распечатывающий различные представления заданного натурального числа N в виде суммы не менее двух натуральных слагаемых. Представления, отличающиеся лишь порядком слагаемых, различными не считаются.

7. Реализуйте рекурсивный алгоритм, распечатывающий по одному разу в лексикографическом порядке все последовательности длины N, составленные из натуральных чисел 1..K.

8. Реализуйте рекурсивный алгоритм вычисления значения многочлена вида

$$P_n(x) = a_0 * x^n + a_1 * x^{n-1} + a_2 * x^{n-2} + ... + a_{n-1} * x + a_n$$

в заданной целочисленной точке согласно формуле

$$P_n(x) = x * P_{n-1}(x) + a_n, \quad \text{где}$$

$$P_{n-1}(x) = a_0 * x^{n-1} + a_1 * x^{n-2} + ... + a_{n-2} * x + a_{n-1}.$$

9. Реализуйте рекурсивный алгоритм нахождения наибольшего общего делителя последовательности N натуральных чисел.

10. Реализуйте рекурсивный алгоритм двоичного поиска элемента с заданным значением в упорядоченном целочисленном массиве.

2. ИЕРАРХИЧЕСКИЕ НЕЛИНЕЙНЫЕ СТРУКТУРЫ ДАННЫХ. ДЕРЕВЬЯ

2.1. Деревья общего вида (произвольной степени)

Нелинейные структуры данных выражают более сложные отношения порядка между объектами, чем отношения предшествования и следования. Наиболее важным видом нелинейных структур являются **деревья**. Древовидные структуры позволяют определить такие отношения, как предок, потомок, брат и т.п.

Дерево (непустое) – конечное множество объектов *T*, состоящее из одного или более узлов, для которых выполняются следующие условия:

- имеется один специально выделенный узел, называемый корнем данного дерева;
- остальные узлы (исключая корень) содержатся в *m* попарно непересекающихся множествах *T1, ...,Tm,* каждое из которых в свою очередь является деревом. Деревья *T1, ...,Tm* называются ***поддеревьями*** данного корня. Структура дерева общего вида представлена на рис. 11.

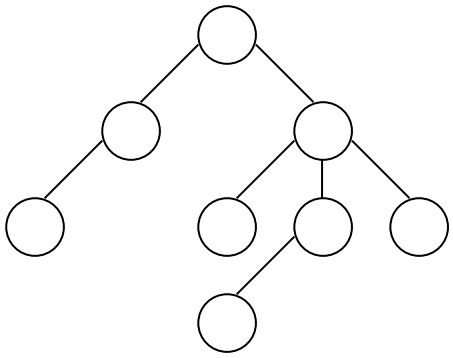


Рис. 11. Дерево общего вида

Пустым называется дерево, не содержащее узлов.

Число поддеревьев данного корня (узла) называется ***степенью*** этого ***узла***. Максимальная степень всех узлов дерева называется ***степенью дерева*** (обозначим *d*). Узел с нулевой степенью называется ***терминальным узлом или листом***. ***Уровень узла*** – выраженная в числе ребер длина пути, ведущего из узла в корень дерева, плюс единица. Считается, что корень дерева находится на уровне 1. Если некоторый узел A располагается на уровне *i*, то находящийся непосредственно ниже его на уровне *i+1* узел B называется ***непосредственным потомком*** узла A, а узел A называется ***непосредственным предком*** узла B. Максимальный уровень всех узлов дерева называется ***высотой или глубиной дерева*** (обозначим *h*). Высота пустого дерева равна нулю, высота дерева, состоящего из одного корня, равна 1. Дерево, содержащее максимальное число узлов, называется ***полным***.

В полном дереве у всех узлов, за исключением терминальных, число непосредственных потомков равно степени дерева. Количество узлов в полном дереве степени *d* высотой *h* вычисляется по формуле (*i* – номер уровня без единицы)

$$N_d(h) = \sum_{i=0}^{h-1} d^i .$$

Основные свойства деревьев общего вида:

- корень не имеет предков;
- каждый узел, за исключением корня, имеет только одного предка;
- каждый узел связан с корнем единственным путем, т.е. в деревьях отсутствуют замкнутые контуры (циклы).

Если в определении дерева имеет значение относительный порядок поддеревьев *T1,…,Tm*, дерево является ***упорядоченным***. Поэтому два упорядоченных дерева на рис. 12 – это разные, отличные друг от друга объекты.

Рис. 12. Два различных дерева

2.2. Бинарные деревья

Особенно важное практическое значение имеют упорядоченные деревья второй степени. Их называют ***двоичными или бинарными деревьями***. ***Бинарное дерево*** – это конечное множество узлов, которое или пусто, или состоит из корня и двух непересекающихся бинарных деревьев, называемых левым и правым поддеревьями данного корня. Примеры бинарных деревьев приведены на рис. 13.

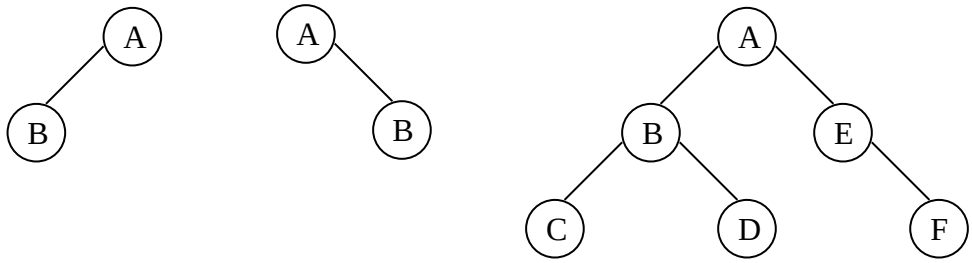


Рис. 13. Примеры бинарных деревьев

Максимальное число узлов в бинарном дереве высотой *h (d=2)*:

$$N_2(h) = \sum_{i=0}^{h-1} 2^i = 2^h - 1 \ .$$

Максимальное число узлов на уровне *i* в бинарном дереве:

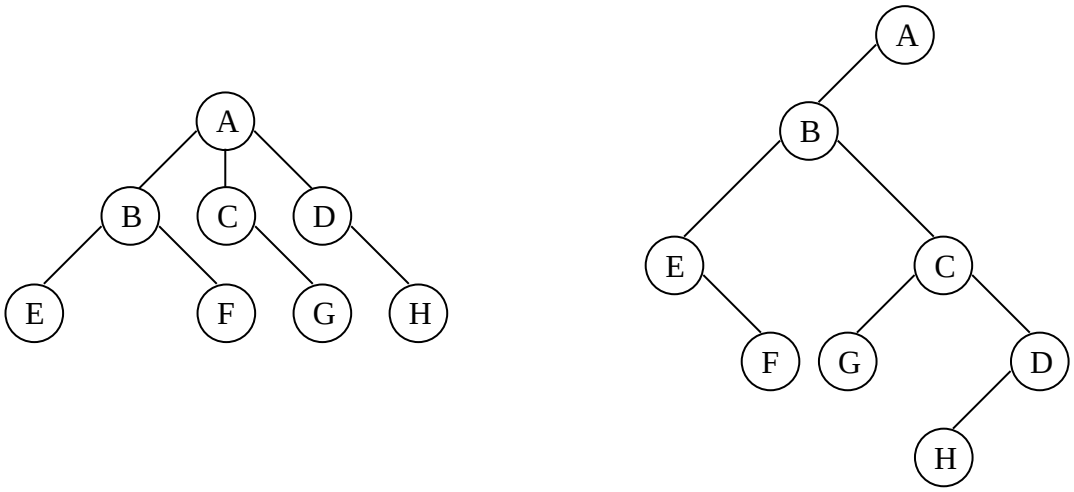
$$n_i = 2^{i-1}.$$

Высота полного бинарного дерева, содержащего *N* узлов:

$$h = \lfloor \log_2 N \rfloor + 1,$$

где $\lfloor \ \rfloor$ означает взятие целой части числа.

Существуют различные алгоритмы преобразования дерева произвольной степени, т.е. дерева общего вида, к виду бинарного. ***Левосторонний алгоритм*** формулируется следующим образом: у каждого узла дерева произвольной степени необходимо сохранить самую левую связь, а узлы – потомки одного и того же узла следует соединить правой связью. На рис. 14.а представлено исходное дерево произвольной степени, а на рис. 14.б – эквивалентное бинарное дерево.



а **б**

Рис. 14. Преобразование дерева произвольной степени к виду бинарного дерева: а – дерево произвольной степени; б – бинарное дерево

2.3. Представление бинарных деревьев

2.3.1. Представление бинарных деревьев в памяти с последовательной организацией

Если известен максимальный размер дерева (т.е. высота, а следовательно, и количество узлов, соответствующих полному дереву), то структуру дерева можно хранить в виде массива. При этом корень дерева будет храниться в элементе массива с индексом [1]. Для каждого узла с номером *k* его левый потомок будет храниться в элементе с индексом [2 * k], а правый – в элементе с индексом [2 * k + 1] (см. рис. 15).

Достоинством представления бинарных деревьев в памяти с последовательной организацией является простота доступа как от предка к потомку, так и от потомка к предку, а недостатком – то, что если дерево не является полным, в массиве появляется большое число пустых элементов. Ограниченный размер массива затрудняет включение в дерево новых узлов, т.к. при этом требуется изменять размер массива. Удаление узла из дерева и соответствующее изменение его структуры также потребует модификации содержимого массива.

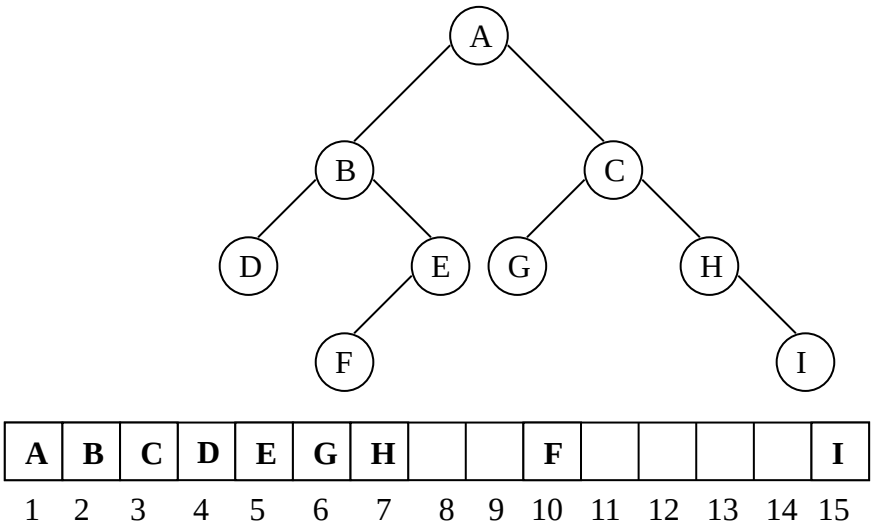


Рис. 15. Представление бинарного дерева в памяти с последовательной организацией

2.3.2. Связанное представление бинарных деревьев

Элемент хранения узла бинарного дерева состоит из одного или нескольких информационных полей и двух полей связи, указывающих соответственно на левое и правое поддеревья данного узла:

```
public class TreeNode                          // Класс «Узел бинарного дерева»
{
   private char info;                                  // информационное поле
   private TreeNode left;                      // ссылка на левое поддерево
   private TreeNode right;                     // ссылка на правое поддерево

   public char Info {…}                                // свойства
   public TreeNode Left{…}
   public TreeNode Right{…}

  public TreeNode () {}                                // конструкторы
  public TreeNode (char info)
  {
     Info = info;
  }
  public TreeNode (char info, TreeNode left, TreeNode right )
```

```
  {
    Info = info;  Left = left; Right = right;
  }
}

public class BinaryTree                    // Класс «Бинарное дерево произвольного вида»
{
  private TreeNode root;                                  // ссылка на корень дерева
  public TreeNode Root                     // свойство, открывающее доступ к корню дерева
  {
    get { return root; }
    set { root = value; }
  }
  public BinaryTree ()                                    // создание пустого дерева
   {
      root = null;
    }
   …
}
```

Дерево задается ссылкой на его корень. Если дерево пусто, ссылка на его корень равна *null*. Связанное представление бинарного дерева иллюстрирует рис. 16.

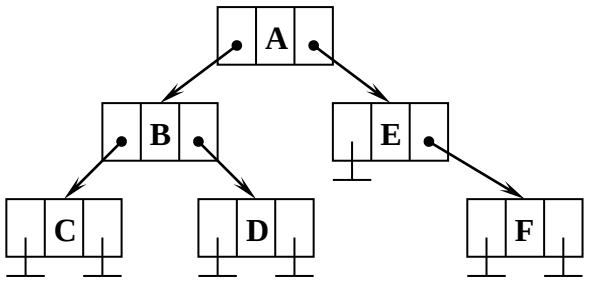


Рис. 16. Связанное представление бинарного дерева

2.4. Алгоритмы обхода бинарных деревьев

Наиболее типичная операция, выполняемая над бинарными деревьями, – обход дерева. ***Обход*** – это операция, при выполнении которой каждый узел обрабатывается ровно один раз одинаковым образом. Обход дерева заключается в разбиении дерева на корень, левое и правое поддеревья и применении к каждому из поддеревьев соответствующей операции обработки до тех пор, пока в процессе разбиения не будет получено пустое дерево. Полный обход дерева дает линейную расстановку узлов, что облегчает выполнение многих алгоритмов. Существует несколько принципов упорядочения, которые естественно вытекают из структуры дерева. Как и саму древовидную структуру, их удобно представить с

помощью рекурсии. Различным принципам упорядочения соответствуют три левосторонних алгоритма обхода в глубину (и три симметричных правосторонних алгоритма): нисходящий, восходящий, смешанный, а также обход в ширину.

2.4.1. Алгоритмы обхода в глубину

Рассмотрим три ***левосторонних*** алгоритма обхода.

Нисходящий (прямой) обход выполняется согласно алгоритму ***“корень-левый-правый” (К-Л-П)***:

1. обработать корневой узел;
2. обойти левое поддерево в нисходящем порядке;
3. обойти правое поддерево в нисходящем порядке.

Трасса нисходящего обхода дерева рис. 16: *A B* ***C D*** *E* ***F***.

Восходящий (концевой) обход выполняется согласно алгоритму **“*левый-правый-корень” (Л-П-К)***:

1. обойти левое поддерево в восходящем порядке;
2. обойти правое поддерево в восходящем порядке;
3. обработать корневой узел.

Трасса восходящего обхода дерева рис. 16: ***C D*** *B* ***F*** *E A* .

Смешанный (обратный) обход выполняется согласно алгоритму **“*левый-корень-правый” (Л-К-П)***.

1. обойти левое поддерево в смешанном порядке;
2. обработать корневой узел;
3. обойти правое поддерево в смешанном порядке.

Трасса смешанного обхода дерева рис. 16: ***C*** *B* ***D*** *A E* ***F*** .

Заметим, что при различных алгоритмах обхода порядок обработки терминальных узлов не изменяется.

Ниже приведен метод, распечатывающий трассу нисходящего обхода бинарного дерева – последовательность информационных полей узлов дерева.

```
public void KLP( TreeNode root )          // root – ссылка на корень дерева и любого из
                                                                  поддеревьев
        {
           if  ( root !=null )                                    // дерево не пусто?
          {                                    // распечатать информ. поле корневого узла
             Console.WriteLine( root.Info );
              KLP( root.Left );          // обойти левое поддерево в нисходящем порядке
(* 1 *)      KLP( root.Right )          // обойти правое поддерево в нисходящем порядке
(* 2 *)    }
(* 3 *)  }
```

Для иллюстрации работы этого метода и построения трассы состояний стека будем использовать следующие обозначения:

^A – адрес узла дерева, в информационном поле которого хранится символ “A”;

(* 1 *) и (* 2 *) – адреса возврата из уровня рекурсивного метода, номер которого на 1 больше текущего уровня;

(* 3 *) – адрес возврата из текущего уровня рекурсивного метода.

Трасса нисходящего обхода дерева, приведенного на рис. 16, содержится в таблице 2.

Таблица 2. Трасса состояний стека при работе метода нисходящего обхода бинарного дерева

Номер входа в метод	Номер уровня рекурсии	Значение фактического параметра	Стек		Выходная трасса	Выход из уровня
			Исходное состояние	Конечное состояние		
1	1	^A	(-,-)	(^A, *1*)	A	
2	2	^B	(^A, *1*)	(^B, *1*) (^A, *1*)	B	
3	3	^C	(^B, *1*) (^A, *1*)	(^C, *1*) (^B, *1*) (^A, *1*)	C	
4	4	null	(^C, *1*) (^B, *1*) (^A, *1*)	(null,*1*) (^C, *1*) (^B, *1*) (^A,*1*)		
	4	null	(null,*1*) (^C, *1*) (^B, *1*) (^A,*1*)	(^C, *1*) (^B, *1*) (^A, *1*)		(*3*)
4	3	^C	(^C, *1*) (^B, *1*) (^A, *1*)	(^C, *2*) (^B, *1*) (^A, *1*)		
5	4	null	(^C, *2*) (^B, *1*) (^A, *1*)	(null,*2*) (^C, *2*) (^B, *1*) (^A, *1*)		
	4	null	(null,*2*) (^C, *2*) (^B, *1*) (^A, *1*)	(^C, *2*) (^B, *1*) (^A, *1*)		(*3*)
	3	^C	(^C, *2*) (^B, *1*) (^A,*1*)	(^B, *1*) (^A, *1*)		(*3*)
	2	^B	(^B, *1*) (^A,*1*)	(^B, *2*) (^A, *1*)		
6	3	^D	(^B,*2*) (^A,*1*)	(^D, *1*) (^B, *2*) (^A, *1*)	D	

Номер входа в метод	**Номер уровня рекур-сии**	**Значение фактичес-кого параметра**	**Исходное состояние стека**	**Конечное состояние стека**	**Выход-ная трасса**	**Выход из уровня**
7	4	null	(^D, *1*) (^B, *2*) (^A, *1*)	(null,*1*) (^D, *1*) (^B, *2*) (^A, *1*)		
	4	null	(null,*1*) (^D, *1*) (^B, *2*) (^A, *1*)	(^D, *1*) (^B, *2*) (^A, *1*)		(*3*)
	3	^D	(^D, *1*) (^B, *2*) (^A, *1*)	(^D, *2*) (^B, *2*) (^A, *1*)		
8	4	null	(^D, *2*) (^B, *2*) (^A, *1*)	(null,*2*) (^D, *2*) (^B, *2*) (^A, *1*)		
	4	null	null,*2*) (^D, *2*) (^B, *2*) (^A, *1*)	(^D, *2*) (^B, *2*) (^A, *1*)		(*3*)
	3	^D	(^D, *2*) (^B, *2*) (^A, *1*)	(^B, *2*) (^A, *1*)		(*3*)
	2	^B	(^B, *2*) (^A, *1*)	(^A, *1*)		(*3*)
	1	^A	(^A, *1*)	(^A, *2*)		
9	2	^E	(^A, *2*)	(^E, *1*) (^A, *2*)	E	
10	3	null	(^E, *1*) (^A, *2*)	(null,*1*) (^E, *1*) (^A, *2*)		
	3	null	(null,*1*) (^E, *1*) (^A, *2*)	(^E, *1*) (^A, *2*)		(*3*)
	2	^E	(^E, *1*) (^A, *2*)	(^E, *2*) (^A, *2*)		

11	3	^F	(^E, *2*) (^A, *2*)	(^F, *1*) (^E, *2*) (^A, *2*)	F	
Номер входа в метод	**Номер уровня рекурсии**	**Значение фактического параметра**	**Исходное состояние стека**	**Конечное состояние стека**	**Выходная трасса**	**Выход из уровня**
12	4	null	(^F, *1*) (^E, *2*) (^A, *2*)	(null, *1*) (^F, *1*) (^E, *2*) (^A, *2*)		
	4	null	(null, *1*) (^F, *1*) (^E, *2*) (^A, *2*)	(^F, *1*) (^E, *2*) (^A, *2*)		(*3*)
12	3	^F	(^E, *2*) (^A, *2*)	(^F, *2*) (^E, *2*) (^A, *1*)		
13	4	null	(^F, *2*) (^E, *2*) (^A, *2*)	(null, *2*) (^F, *2*) (^E, *2*) (^A, *2*)		
	4	null	(null, *2*) (^F, *2*) (^E, *2*) (^A, *2*)	(^F, *2*) (^E, *2*) (^A, *2*)		(*3*)
	3	^F	(^F, *2*) (^E, *2*) (^A, *2*)	(^E, *2*) (^A, *2*)		(*3*)
	2	^E	(^E, *2*) (^A, *2*)	(^A, *2*)		(*3*)
	1	^A	(^A, *2*)	(-, -)		(*3*)

2.4.2. Алгоритм обхода в ширину

При обходе в ширину узлы посещаются уровень за уровнем, начиная с корня, причем каждый уровень обходится слева направо.

Для реализации данного алгоритма используется структура очереди, в которой хранятся ссылки на поддеревья каждого узла. Для структуры очереди определены операции: *In_Queue* – включить в очередь; *Out_Queue* – исключить из очереди. Ниже приведена схема алгоритма обхода дерева в ширину: Q – ссылка на голову очереди, Root – ссылка на корень дерева.

```
…
 In_Queue( Q, Root );                    // Занести в очередь ссылку на корень дерева
 while ( Q != null )                     // Выполнять до тех пор, пока очередь не пуста
 {
    Out_Queue( Q, p );                   // p – ссылка на узел, извлеченный из очереди
    Work( p );                           // Обработать узел p
    if  ( p.Left != null ) In_Queue( Q,p.Left );   // Если узел p имеет левого
                                         потомка, занести в очередь ссылку на левого потомка
```

```
    if  ( p.Right != null ) In_Queue( Q,p.Right );          // Если узел p имеет правого
                                                потомка, занести в очередь ссылку на правого потомка
}
```

Трасса обхода в ширину дерева рис. 67: *A B E* ***C D F***.

2.5. Виды бинарных деревьев

2.5.1. Деревья произвольного вида

Структура бинарного дерева ***произвольного вида*** не зависит ни от количества узлов в дереве, ни от каких-либо специальных признаков узлов, хранящихся в полях узлов. Бинарное дерево произвольного вида создается, начиная с корня, с помощью последовательного включения узлов либо в левое, либо в правое поддерево корневого узла. Пример структуры бинарного дерева произвольного вида показан на рис. 16, 15.

2.5.2. Сбалансированные деревья

Дерево называется ***идеально сбалансированным***, если для каждой вершины число узлов в ее правом и левом поддеревьях отличается не более чем на единицу (далее будем называть такие деревья сбалансированными). ***Сбалансированное дерево***, состоящее из *N* узлов, имеет минимальную высоту среди всех бинарных деревьев. Высоту сбалансированного дерева можно определить по формуле:

$$h = \lfloor \log_2 N \rfloor + 1.$$

Минимальная высота при заданном числе узлов достигается, если на всех уровнях, кроме терминального, размещается максимально возможное число узлов. Это происходит за счет равномерного размещения узлов поровну слева и справа от каждого узла.

Правило равномерного размещения для *N* узлов (*правило 1*) формулируется с помощью рекурсии:

- создать один узел в качестве корня;
- по *правилу 1* построить левое поддерево с числом узлов *nl = N/2;*
- по *правилу 1* построить правое поддерево с числом узлов *nr = N - nl - 1*.

Структура сбалансированного дерева из *N* узлов определяется количеством узлов (рис. 17).

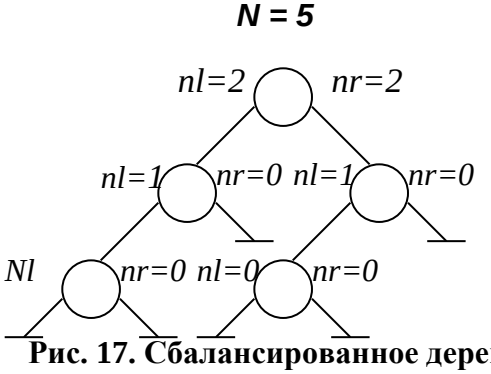


Рис. 17. Сбалансированное дерево

Метод построения сбалансированного дерева из *N* узлов:

```
public TreeNode Create_Balanced( int n)              // n – количество узлов в дереве
{                                                    // root – ссылка на корень дерева и
  char x; TreeNode root;                             на корень любого из поддеревьев
  if ( n == 0 ) root = null;                         // если n == 0, построить пустое дерево
  else
  {                                                  // заполнить информационное поле корня
       Console.WriteLine("Введите значение инф. поля узла ");
       x = Char.Parse( Console.ReadLine() );
       root = new TreeNode( x );                     // создать корень дерева
       root.Left = CreateBalanced( n/2 );            // построить левое поддерево
(*1*)  root.Right = CreateBalanced( n – n/2 – 1);    // построить правое поддерево
      }
(*2*) return root;
(*3*) }
...
BinaryTree T = new BinaryTree();     // объявление объекта класса "Бинарное дерево"
T.Root = T.Create_Balanced( 7 );     // создание сбалансированного дерева из 7 узлов
T.KLP(T.Root);                       // трасса нисходящего обхода дерева
```

Для иллюстрации работы этого метода и построения трассы состояний стека будем использовать следующие обозначения:

^A – адрес узла дерева, в информационном поле которого хранится символ "A";

(* 1 *) и (* 2 *) – адреса возврата из уровня рекурсивного метода, номер которого на 1 больше текущего уровня;

(* 3 *) – адрес возврата из текущего уровня рекурсивного метода.

Трасса построения сбалансированного дерева, содержащего 3 узла, приведена в таблице 3.

Сбалансированное дерево, как и дерево любого другого вида, можно построить в процессе нисходящего обхода. Особенность работы метода построения дерева заключается в том, что сначала создаются корневые узлы, адреса которых запоминаются в стеке. Установить ссылки из корневых узлов дерева на их поддеревья можно только после того, как эти поддеревья будут созданы и адреса их корневых узлов возвращены в точки вызова. Точками вызова являются поля ссылок на левое поддерево (*root.Left*) или правое поддерево (*root.Right*) корневого узла (в таблице установка этих ссылочных связей показана стрелками).

Таблица 3. Трасса состояний стека при работе метода построения сбалансированного дерева

Номер уровня рекур-сии	Знач-я входного парам-ра n	Стек		Знач-я вых. пар-ра			Выход из уровня
		Исходное состояние (n,root, (.) возврата)	Конечное состояние (n,root, (.) возврата)	root	root. left	root. right	
1	3	(-, -, -)	(3,^A, *1*)	^A	-	-	-
2	1	(3,^A, *1*)	(1, ^B, *1*) (3, ^A, *1*)	^B	-	-	-
3	0	(1, ^B, *1*) (3, ^A, *1*)	(0,null ,*1*) (1, ^B, *1*) (3, ^A, *1*)	null	-	-	-
	0	(0,null ,*1*) (1, ^B, *1*) (3, ^A, *1*)	(1, ^B, *1*) (3, ^A, *1*)	null	-	-	(* 3 *)
2	1	(1, ^B, *1*) (3, ^A, *1*)	(1, ^B, *2*) (3, ^A, *1*)	^B	null	-	-
3	0	(1, ^B, *2*) (3, ^A, *1*)	(0,null,*2*) (1, ^B, *2*) (3, ^A, *1*)	null	-	-	-
	0	(0,null,*2*) (1, ^B, *2*) (3, ^A, *1*)	(1, ^B, *2*) (3, ^A, *1*)	null	-	-	(* 3 *)
2	1	(1, ^B, *2*) (3, ^A, *1*)	(3, ^A, *1*)	^B	null	null	(* 3 *)
1	3	(3, ^A, *1*)	(3, ^A, *2*)	^A	^B	-	-
2	1	(3,^A, *2*)	(1, ^C, *1*) (3, ^A, *2*)	^C	-	-	-
3	0	(1, ^C, *1*) (3, ^A, *2*)	(0,null,*1*) (1, ^C, *1*) (3, ^A, *2*)	null	-	-	-
	0	(0,null,*1*) (1, ^C, *1*) (3, ^A, *2*)	(1, ^C, *1*) (3, ^A, *2*)	null	-	-	(* 3 *)
2	1	(1, ^C, *1*) (3, ^A, *2*)	(1, ^C, *2*) (3, ^A, *2*)	^C	null		
3	0	(1, ^C, *2*) (3, ^A, *2*)	(0,null,*2*) (1, ^C, *2*) (3, ^A, *2*)	null	-	-	-
	0	(0,null,*2*) (1, ^C, *2*) (3, ^A, *2*)	(1, ^C, *2*) (3, ^A, *2*)	null	-	-	(* 3 *)
2	1	(1, ^C, *2*) (3, ^A, *2*)	(3, ^A, *2*)	^C	null	null	(* 3 *)
1	3	(3, ^A, *2*)	(-, -, -)	^A	^B	^C	(* 3 *)

2.5.3. Дихотомические деревья (деревья поиска)

Бинарные деревья часто используются для представления множества объектов, среди которых идет поиск по уникальному значению некоторого атрибута, называемого ***ключом***. При этом на множестве объектов определено особое отношение порядка – ***дихотомия***, заключающееся в поэтапном разбиении множества информационных полей объектов на два непересекающихся подмножества. ***Дихотомическим деревом (деревом поиска)*** называется бинарное дерево, организованное так, что для каждого узла, имеющего ключ *K*, справедливо утверждение о том, что в его левом поддереве содержатся узлы с ключами, меньшими *K*, а в его правом поддереве содержатся узлы с ключами, большими *K*.

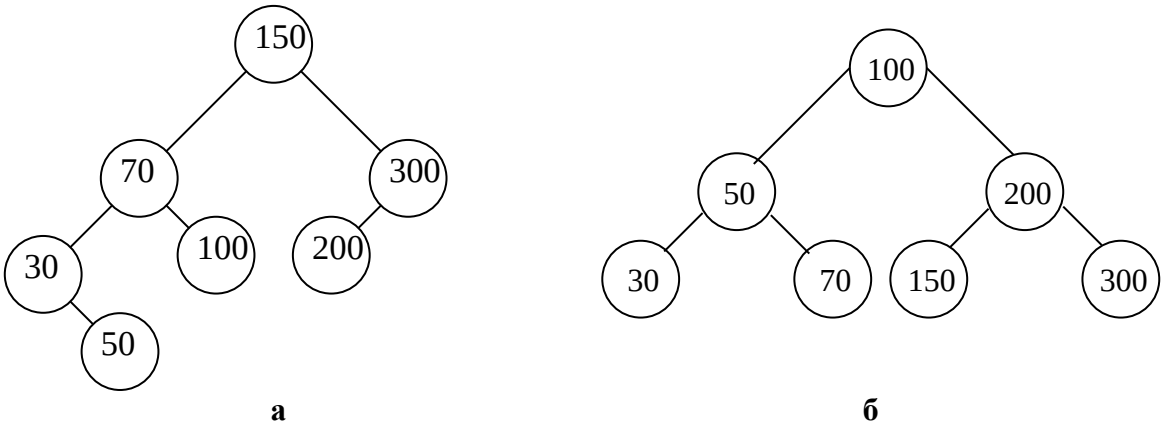
Описание элемента хранения узла дихотомического дерева:

```
public class DTreeNode                          // Класс «Узел дихотомического дерева»
{
   private char info;                                   // информационное поле
   private int key;                                     // поле ключа
   private DTreeNode left;                              // ссылка на левое поддерево
   private DTreeNode right;                             // ссылка на правое поддерево

   public char Info {…}                                 // свойства
   public int Key {…}
   public DTreeNode Left{…}
   public DTreeNode Right{…}

  public DTreeNode () {}                                // конструкторы
  public DTreeNode (char info, int key)
  {
     Info = info;  Key = key;
  }
  public DTreeNode (char info, int key, DTreeNode left, DTreeNode right )
  {
    Info = info;  Key = key; Left = left; Right = right;
  }
}

public class DichotomyTree                      // класс «Дихотомическое дерево»
{
  private DTreeNode root;                       // ссылка на корень дихотомического дерева

  public DTreeNode Root                         // свойство, открывающее доступ к корню дерева
  {
    get { return root; }
    set { root = value; }
  }
```

```
  public DichotomyTree ()                    // создание пустого дерева
   {
      root = null;
    }
   …
}
```

Структура дихотомического дерева определяется последовательностью задания ключей. Последовательности задания ключей для

- дерева рис. 18 а) – 150, 70, 300, 100, 30, 200, 50;
- дерева рис. 18 б) – 100, 50, 30, 70, 200, 150, 300;
- дерева рис. 19 – 300, 200, 150, 100, 70, 50, 30.



а **б**

Рис. 18. Дихомические деревья:
а – несбалансированное; б – сбалансированное

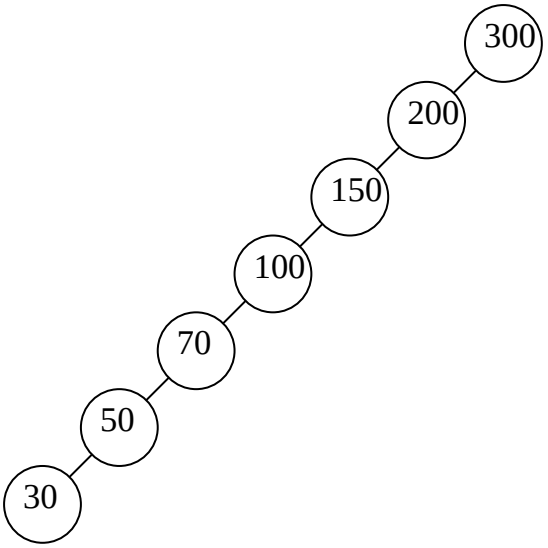


Рис. 19. Вырожденное дихотомическое дерево

Поиск в дихотомическом дереве узла с заданным значением ключа осуществляется на основе сравнения заданного ключа с ключом корня. Единственное сравнение позволяет перейти к левому или к правому

поддереву корня. Если дихотомическое дерево является сбалансированным, поиск узла с заданным значением ключа требует не более чем $\lfloor log_2 N \rfloor + 1$ сравнений, где N – количество узлов дерева. В частном случае, когда множество ключей является линейно упорядоченным, дихотомическое дерево фактически вырождается в линейный список (рис. 19). При этом поиск среди N узлов выполняется максимум за N сравнений, а в среднем – за $N / 2$ сравнений:

Включение в дихотомическое дерево узла с заданным значением ключа производится следующим образом: если поиск узла привел в тупик (т.е. к пустому поддереву, обозначенному ссылкой со значением *null*), то новый узел необходимо включить в дерево на место пустого поддерева. Таким образом, узел включается в дихотомическое дерево всегда в качестве листа.

```
public DTreeNode Ins_DNode(DTreeNode root, int k);
                                                    / root – ссылка на корень дерева
{                                                   // k – ключ вставляемого узла
 if  ( root == null )                  // дерево пусто – вставка узла в качестве листа
                                      // создать и инициализировать элемент хранения узла
    root = new DTreeNode(‘ ‘, k, null, null);
 else                                          // дерево не пусто – продолжить поиск:
 {                                                        // поиск в левом поддереве
   if  (k < root.Key) root.Left = Ins_DNode( root.Left,k )
   else                                                  // поиск в правом поддереве
     if  (k > root.Key) root.Right = Ins_DNnode( root.Right,k )
     else                                          // ключ должен быть уникальным
       Console.WriteLine(“узел с ключом {0} уже есть в дереве”, k)
 }
 return root;
}
```

Если корневой узел не содержит искомого ключа, метод рекурсивно вызывает сам себя для продолжения поиска либо в левом, либо в правом поддереве. В том случае, если достигнут конец ветви и не обнаружен искомый ключ, создается новый узел с этим значением ключа. Рекурсия заканчивается, когда искомый ключ найден (при этом выдается сообщение о невозможности повторного включения узла) или достигнут конец ветви (при этом новый узел включается в дерево).

??? *Какой алгоритм обхода лежит в основе метода включения узла в дихотомическое дерево?*

При ***построении дихотомического дерева*** из N узлов на каждом из N шагов цикла осуществляется вставка узла с заданным значением ключа

(вызов метода *Ins_DNode*). Поэтому сложность создания дихотомического дерева из N узлов оценивается как $N * log_2 N$ операций**.**

При ***удалении узла*** с заданным значением ключа из дихотомического дерева различают три случая:

1. узла с заданным значением ключа в дереве нет;
2. узел с заданным значением ключа имеет не более одного потомка. В этом случае исключаемый узел заменяется на своего потомка;
3. узел с заданным значением ключа имеет двух потомков. В этом случае исключаемый узел заменяется либо на *самый правый узел его левого поддерева*, имеющий не более одного потомка, либо на *самый левый узел его правого поддерева*, имеющий не более одного потомка.

Пример исключения узлов из дихотомического дерева приведен на рис. 20.

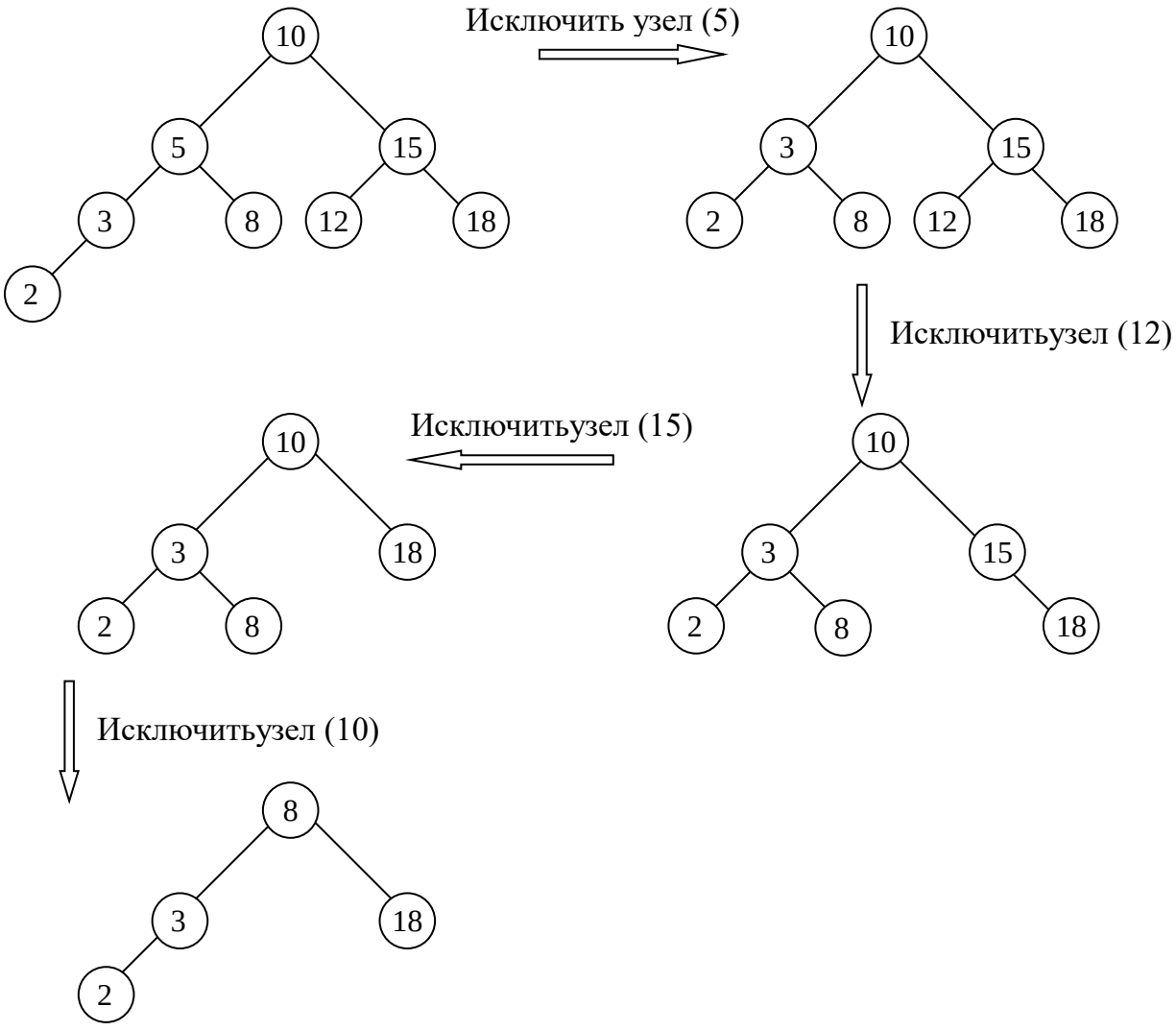


Рис. 20. Исключение узлов из дихотомического дерева

Разрушение бинарного дерева любого вида заключается в присвоении *null* ссылке на корень дерева. В результате разрушения дерево становится пустым.

2.5.4. Деревья выражений

Дерево выражений – бинарное дерево, в корневых узлах которого хранятся признаки операций, а в терминальных узлах – операнды выражения (переменные или константы). Дерево выражений представлено на рис. 21.

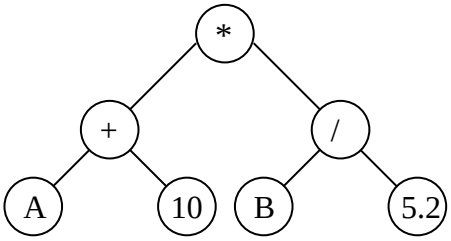


Рис. 21. Дерево выражений

Различные алгоритмы обхода дерева выражений соответствуют различной структуре представления выражения в виде строки.

Нисходящему обходу соответствует ***префиксная форма*** представления, т. к. в ней знак операции предшествует операнду:

** + A 10. / B 5.2 .*

Восходящему обходу соответствует ***постфиксная форма*** представления, т. к. в ней знак операции находится после операндов:

*A 10. + B 5.2 / * .*

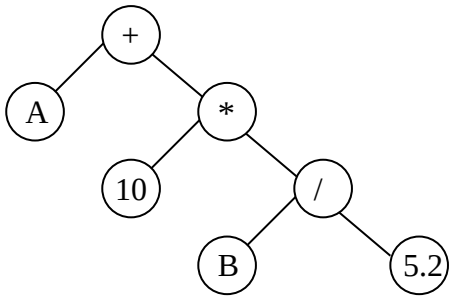
Смешанному обходу соответствует ***инфиксная форма*** представления, т. к. в ней знак операции находится между операндами:

*A + 10. * B / 5.2 .*

Для того чтобы задать приоритеты операций, используется абсолютно скобочная форма, в которой каждое подвыражение заключается в круглые скобки:

*((A + 10.) * (B / 5.2)) .*

На рис. 22 приведено дерево, смешанный обход которого позволяет получить бесскобочную инфиксную форму, эквивалентную дереву рис. 21, а скобочная инфиксная форма задает совсем другие приоритеты операций. Заметим, что подобная проблема не может возникнуть при использовании префиксной или постфиксной формы представления выражения.



*(A + (10. * (B / 5.2)))*

Рис. 22. Дерево выражений

??? *Какой алгоритм обхода необходимо использовать для того, чтобы подсчитать значение арифметического выражения, представленного в виде дерева?*

2.6. Демонстрационная программа, реализующая операции создания, обработки, просмотра содержимого бинарного дерева (на примере сбалансированного дерева)

```
public class TreeNode                          // описание узла бинарного дерева
{
   private char info;                          // информационное поле
   private TreeNode left;                      // ссылка на левое поддерево
   private TreeNode right;                     // ссылка на правое поддерево

   public  char Info {…}                       // свойства
   public TreeNode Left{…}
   public TreeNode Right{…}

  public TreeNode () {}                        // конструкторы
  public TreeNode (char info)
  {
     Info = info;
  }

  public TreeNode (char info, TreeNode left, TreeNode right )
  {
    Info = info;  Left = left; Right = right;
  }
}

public class BalancedTree                      // Класс “Сбалансированное дерево ”
{
  private TreeNode root;                       // ссылка на корень дерева
  public TreeNode Root                         // свойство, открывающее доступ к корню дерева
  {
```

```
    get { return root; }
    set { root = value; }
  }

  public BalancedTree ()                                  // создание пустого дерева
   {
      root = null;
    }
                                                          // создание дерева из n узлов
  public TreeNode Create_Balanced( int n)                 // n – количество узлов в дереве
 {                                                        // root – ссылка на корень дерева и
    char x; TreeNode root;                                на корень любого из поддеревьев
    if ( n == 0 ) root = null;                            // если n == 0, построить пустое дерево
    else
   {                                                      // заполнить информационное поле корня
        Console.WriteLine("Введите значение инф. поля узла ");
        x = Char.Parse( Console.ReadLine() );
        root = new TreeNode( x );                         // создать корень дерева
        root.Left = CreateBalanced( n/2 );                // построить левое поддерево
        root.Right = CreateBalanced( n – n/2 – 1);        // построить правое поддерево
    }
    return root;
  }

   public void KLP(TreeNode root)                         //  нисходящий обход
  {
     if (root != null)
    {
        Console.WriteLine(root.Info);                     //  обработать корень
        KLP(root.Left);                 // в нисходящем порядке обойти левое поддерево
        KLP(root.Right);                // в нисходящем порядке обойти правое поддерево
    }
  }

  public int Count( )                   // подсчет кол-ва узлов в дереве - восходящий обход
  {
     int s;
     if  ( root == null ) s = 0;
     else s = Work( root^.left ) + Work( root^.right + 1;
     return s;
  }

   public void Destroy( );                                { разрушение дерева }
  {
     root = null;
```

```
  }

public class Program
{
   static void Main()
   {
     BalancedTree T = new BalancedTree();                    // конструктор
     T.Root = T.Create();                    // создание сбалансированного дерева
     Console.ReadLine();
     T.KLP();                                  // нисходящий обход дерева
     Console.Write ( “Количество узлов в дереве = “ );
     Console.WriteLine ( T.Count() );                        // обработка дерева
     Console.ReadLine();
     T.Destroy();                                            // разрушение дерева
    }
}
```

2.7. Контрольные вопросы к главе 3

1. Дайте определение дерева общего вида.

2. Что такое степень дерева и глубина дерева?

3. Перечислите свойства деревьев общего вида.

4. Дайте определение бинарного дерева.

5. Сформулируйте алгоритм преобразования дерева произвольного вида к виду бинарного дерева.

6. Каким образом бинарное дерево представляется в памяти с последовательной и связанной организацией?

7. Какие алгоритмы обхода бинарных деревьев Вы знаете?

8. Дайте определение сбалансированного дерева. В чем отличительные особенности сбалансированных деревьев? Сформулируйте алгоритм построения сбалансированного дерева.

9. Дайте определение дихотомического дерева. Приведите примеры применения дихотомических деревьев.

10. Сформулируйте алгоритм создания дихотомического дерева.

11. Сформулируйте алгоритм исключения узла из дихотомического дерева.

12. Дайте определение дерева выражений.

13. Какой алгоритм обхода необходимо использовать для вычисления значения выражения, представленного в виде дерева?

14. Какой алгоритм обхода необходимо использовать для разрушения дерева?

2.8. Упражнения к главе 2

1. Создать сбалансированное дерево. Подсчитать количество узлов дерева с положительными и отрицательными значениями информационных полей.

2. Создать сбалансированное дерево. Подсчитать количество узлов дерева с заданным значением информационных полей.

3. Создать дерево поиска. Подсчитать сумму значений информационных полей узлов дерева.

4. Создать два дерева поиска. Скопировать в линейный список информационные поля элементов с совпадающими в первом и втором деревьях значениями ключей.

5. Создать дерево поиска. Построить копию этого дерева.

6. Создать дерево поиска. Скопировать в линейный список узлы дерева, выбранные по заданным значениям ключей.

7. Создать дерево поиска. Скопировать в новое дерево поиска узлы дерева, выбранные по заданным значениям ключей.

8. Создать два сбалансированных дерева. Определить эквивалентность двух деревьев.

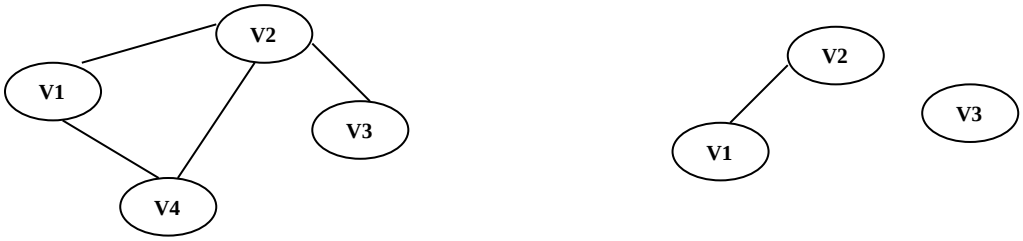
9. Создать дерево поиска. Определить глубину дерева.

10. Создать дерево поиска. Написать процедуру исключения произвольного узла из дерева.

3. ИЕРАРХИЧЕСКИЕ НЕЛИНЕЙНЫЕ СТРУКТУРЫ ДАННЫХ. ГРАФЫ

3.1. Основные понятия и определения

Граф G = (V, E) включает конечное множество вершин V и конечное множество ребер E, соединяющих эти вершины. Любые две вершины, соединенные в графе ребром, называются смежными. Ребро графа, направленное от одной вершины к другой, называется ориентированным. Ребро, не имеющее определенного направления, является неориентированным. Граф, в котором все ребра ориентированы, называется ***орграфом***. Ребро графа, соединяющее некоторую вершину саму с собой, называется петлей. Граф, в котором между любой парой вершин существует не более одного ребра (не более одного ребра данного направления для орграфа), называется ***простым***. Примеры неориентированных и ориентированных графов приведены на рис. 23 и 24.



а **б**

Рис. 23. Неориентированные графы
а – граф G1; б – граф G2

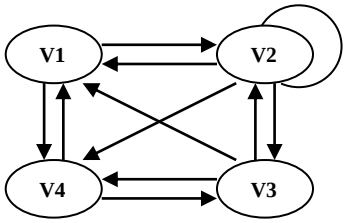


Рис. 24. Ориентированный граф G3

Пусть G = (V, E) – ***простой граф*** (простой орграф). Любая последовательность ребер графа (ориентированных ребер орграфа) такая, что конечная вершина любого ребра этой последовательности является начальной вершиной следующего ребра, задает ***путь*** P в графе. Число ребер в последовательности, задающей путь, является ***длиной пути***.

$P = ((v_1, v_2), (v_2, v_3), \ldots, (v_{K-2}, v_{K-1}), (v_{K-1}, v_K))$ – путь в графе.

$P = (v_1, v_2, \ldots, v_{K-1}, v_K)$ - сокращенное обозначение пути в графе.

Пути в графе G_1: $P_1 = (1,4,2,3)$; $P_2 = (1,4,2,1)$; $P_3 = (2,3,2)$; $P_4 = (1,4,2,1,4,1)$.
Различные пути из вершины 2 в вершину 4 в графе G_3: $P_1 = (2,4)$; $P_2 = (2,3,4)$; $P_3 = (2,3,1,4)$; $P_4 = (2,1,4)$; $P_5 = (2,3,2,4)$; $P_6 = (2,2,4)$;

Путь в графе, все ребра которого различны, называется ***простым путем***. Путь в графе, все вершины которого различны, называется ***элементарным путем***. Все элементарные пути являются также и простыми. Для графа G_3 пути P_1 , P_2 , P_3 , P_5 – простые, а пути P_4 , P_6 - простые, но не элементарные.

Простой путь, начинающийся и заканчивающийся в одной вершине, называется ***циклом***. В неориентированном графе длина цикла больше или равна 3. Для графа G1: путь C_1 = (1,2,4,1) – цикл; путь P = (2, 3,2) – не цикл. В орграфе длина цикла больше или равна 2. Простой граф без цикла называется ***ациклическим***.

Граф называется ***связным***, если для любой его вершины i существует путь к любой другой вершине j: P = (i = v_1 , v_2 , …, v_{k-1} , v_k = j). Если в графе какие либо два подмножества вершин не соединены ребрами, граф не является связным. Например, граф G_2 не является связным.

Граф G' = (V', E') является ***подграфом*** графа G = (V, E), если V' ⊂ V, E' ⊂ E. Максимальный связный подграф графа называется его ***компонентой связности***. Ориентированный граф называется ***сильно связным***, если для каждой пары вершин i и j существует хотя бы один ориентированный путь из i в j и хотя бы один ориентированный путь из j в i.

Дерево – ациклический связный граф, одна вершина которого (корень) не имеет входящих ребер, а все другие вершины имеют единственное входящее ребро. В дереве n вершин, (n-1) ребер. Каждому ребру графа можно сопоставить некоторое число, называемое ***весом***, физический смысл которого определяется предметной областью решаемой задачи. Такой граф называется ***взвешенным***.

3.2. Представление графов

3.2.1. Матричное представление графов

Пусть для простого графа G = (V, E) задан некоторый вид упорядоченности вершин {v_1 , v_2 , …, v_n }, такой, что некоторая вершина названа первой, другая – второй и т.д., где n – количество вершин.

Матрица A (n, n) = $|a_{ij}|$ i =1..n , j = 1.. n такая, что

$$a_{ij} = \begin{cases} 1 - \text{ребро } (v_i, v_j) \text{ из вершины i в вершину j существует,} \\ 0 - \text{ребро } (v_i, v_j) \text{ из вершины i в вершину j не существует,} \end{cases}$$

называется ***матрицей смежности***. Для простых неориентированных графов матрица смежности симметрична относительно главной диагонали. Для графа без петель главная диагональ содержит только нулевые элементы.

Описание графа с помощью матрицы смежности:

```
public class Graph
{
  private int size;                          // количество вершин в графе
  private bool [,] adjacency;                // матрица смежности графа
}
```

Граф G4 (рис. 25) представлен матрицей смежности (рис. 26). Элементы i-й строки матрицы определяются ребрами, исходящими из вершины v_i. Элементы j-го столбца определяются ребрами, входящими в вершину v_j.

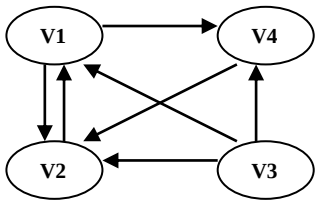


Рис. 25. Граф G4

	V1	V2	V3	V4
V1	0	1	0	1
V2	1	0	0	0
V3	1	1	0	1
V4	0	1	0	0

Рис. 26. Матрица смежности графа G4

Для взвешенных графов задается ***весовая матрица*** W.

Матрица $W(n, n) = |w_{ij}|$ $i = 1..n$, $j = 1..n$ такая, что

$$w_{ij} = \begin{cases} \text{вес} - \text{ребро } (v_i, v_j) \text{ из вершины i в вершину j существует,} \\ \infty - \text{ребро } (v_i, v_j) \text{ из вершины i в вершину j не существует,} \end{cases}$$

называется ***весовой матрицей.***

Описание графа с помощью весовой матрицы:

```
public class Graph
{
  private int size;                          // количество вершин в графе
  private int [,] weight;                    // весовая матрица графа
}
```

В матрице смежности единица на пересечении i-й строки и j-го столбца матрицы A означает наличие ребра (v_i, v_j), т.е., прямого пути длины 1 из вершины i вершину j. Прямой путь длины 1 между вершинами графа существует не всегда. Поэтому необходимо определить, существует ли в графе какой либо путь (не обязательно простой) длины, не превышающей n, из вершины i в вершину j. Это можно определить с помощью матрицы достижимости P.

Матрица $P(n, n) = |p_{ij}|$ i =1..n , j = 1.. n такая, что

$$p_{ij} = \begin{cases} 1 - \text{существует путь длины} \le n \text{ из вершины } v_i \text{ в } v_j, \\ 0 - \text{не существует путь длины} \le n \text{ из вершины } v_i \text{ в } v_j, \end{cases}$$

называется ***матрицей достижимости***.

Для определения матрицы достижимости используются булевы матричные операции над квадратными матрицами размера n * n: булева сумма (∨) и булево произведение (∧). Для вычисления матрицы P задается выражение:

$$P = A \vee A^2 \vee A^3 \vee \ldots \vee A^{n-1} \vee A^n = \bigcup_{k=1}^{n} A^k.$$

Каждый элемент матрицы A^K, k =1,…,n, находящийся на пересечении i-й строки и j –го столбца, равен 1, если существует не обязательно простой путь длины k, ведущий из вершины i в вершину j. Например, элементы матрицы A^2 на пересечении строки i и столбца j задают наличие или отсутствие пути длины 2 между v_i и v_j. Элементы этой матрицы определяются выражением

$$A^2 = |a^2_{ij}| = \left|\bigcup_{k=1}^{n}(a_{ik} \cap a_{kj})\right| \quad i,j = 1,\ldots,n.$$

Для любого k условие $a_{ik} \wedge a_{kj} = 1$ выполняется тогда и только тогда, когда оба элемента a_{ik} и a_{kj} равны 1, т.е., в графе имеются ребра (v_i, v_k) и (v_k, v_j). Таким образом,

$$A^2 = A \wedge A;\ A^3 = A \wedge A^2;\ A^4 = A \wedge A^3;\ \ldots\ A^n = A \wedge A^{n-1}.$$

Для графа G4 матрица достижимости вычисляется следующим образом.

$$A^2 = \begin{vmatrix} 1\,1\,0\,0 \\ 0\,1\,0\,1 \\ 1\,1\,0\,1 \\ 1\,0\,0\,0 \end{vmatrix} \quad A^3 = \begin{vmatrix} 1\,1\,0\,1 \\ 1\,1\,0\,0 \\ 1\,1\,0\,1 \\ 0\,1\,0\,1 \end{vmatrix} \quad A^4 = \begin{vmatrix} 1\,1\,0\,1 \\ 1\,1\,0\,1 \\ 1\,1\,0\,1 \\ 1\,1\,0\,0 \end{vmatrix} \quad P = \begin{vmatrix} 1\,1\,0\,1 \\ 1\,1\,0\,1 \\ 1\,1\,0\,1 \\ 1\,1\,0\,1 \end{vmatrix}$$

Диагональные элементы в матрицах A^K, k =1,…,n указывают наличие цикла длины k в вершине v_i, i = 1,…,n. Диагональные элементы в матрице P указывают наличие цикла длины, не превышающей n, в вершинах v_i, i=1,…,n. Для ациклических графов диагональные элементы равны 0.

Матрица $Q(n, n) = |q_{ij}|$ i =1..n , j = 1.. n такая, что

$$q_{ij} = \begin{cases} 1 – \text{существует путь длины} \le n \text{ из вершины } v_j \text{ в } v_i, \\ 0 – \text{не существует путь длины} \le n \text{ из вершины } v_j \text{ в } v_i, \end{cases}$$

называется ***матрицей контрдостижимости***. Столбец j матрицы Q совпадает со строкой j матрицы P, т.е., матрица контрдостижимости является транспонированной к матрице достижимости $Q = P^T$.

3.2.2. Представление графа в виде списка смежности

Представление графа G4 (рис. 27) в виде списка смежности приведено на рис. 30.

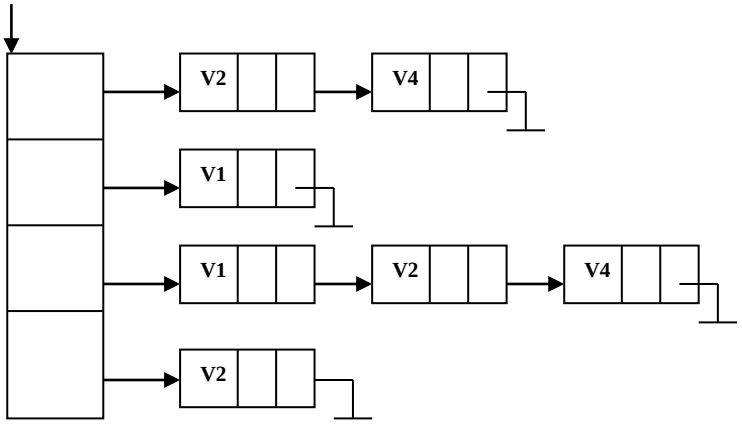


Рис. 27. Представление графа в виде списка смежности

Описание графа с помощью списка смежности:

```
public class AdjListNode                          // Класс "Узел списка вершин"
{
   private int id;                                // идентификатор смежной вершины
   private int weight;                            // вес ребра
   private AdjListNode link;                      // ссылка на соседний элемент списка вершин

   public int Id {…}                              // свойства
   public int Weight {…}
   public AdjListNode Link {…}

   public AdjListNode ( ) {}                      // конструкторы
   public AdjListNode (int id, int weight)
   {
      Id = id; Weight = weight;
   }
}
public class  Graph                               // Класс "Граф"
```

```
{
  private int size;                                  // количество вершин в графе
  private  AdjListNode [] adjList;          // ссылка на список смежности вершин графа

  public int Size {…}                                // свойства
  public  AdjListNode [] AdjList {…}

  public Graph (int size)                            // конструктор
  {                          // создание и инициализация списка смежности вершин графа
     AdjList = new AdjListNode [size];
     for ( int i=0; i<=size; i++ ) AdjList[i] = null;
  }
  . . .                                              // методы
}
```

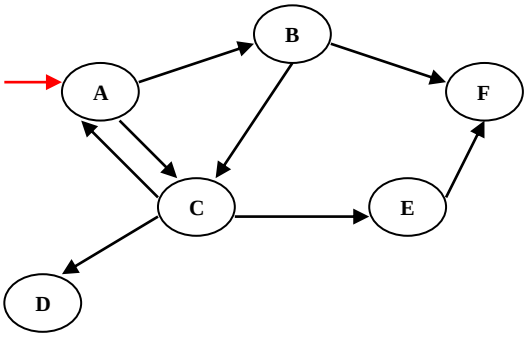
Выбор представления графа в виде списка смежности или матрицы смежности зависит от того, “разреженный” граф или “плотный”. Плотным называется граф, у которого количество ребер приближается к числу (n^2-n), где n – количество вершин. Разреженным называется граф, у которого количество ребер намного меньше числа вершин. Для представления плотных графов рекомендуется использовать матрицы смежности, а для представления разреженных графов – списки смежности.

3.3. Алгоритмы обхода графов

Обход графа легко выполнить, если принять одну из его вершин в качестве первой посещаемой, т.е. “корня”. Затем следует рекурсивно обрабатывать все остальные вершины, достижимые от первой. Существует два основных алгоритма обхода графа.

3.3.1 Алгоритм обхода в глубину

Этот алгоритм обобщает алгоритм нисходящего обхода деревьев. Рассмотрим алгоритм обхода в глубину на примере графа G5 (рис. 28), представленного матрицей смежности.



	A	B	C	D	E	F
A	0	1	1	0	0	0
B	0	0	1	0	0	1
C	1	0	0	1	1	0
D	0	0	0	0	0	0
E	0	0	0	0	0	1
F	0	0	0	0	0	0

Рис. 28. Граф G5, представленный матрицей смежности

В качестве корневой выберем вершину A. Обработав корневую вершину A, находим в строке матрицы смежности этой вершины первый элемент не равный нулю. Он соответствует вершине B, поэтому далее рассматриваем строку матрицы, соответствующую вершине B.

Обрабатываем вершину B. Для нее первым ненулевым элементом в строке матрицы смежности является вершина C. Переходим к строке, соответствующей вершине C, обрабатываем вершину C. Для вершины C первым ненулевым элементом в строке матрицы смежности является A, но вершина A уже обработана, значит, для вершины C выбираем следующий элемент не равный нулю, т.е. D. Строка матрицы смежности, соответствующая вершине D, пуста. Поэтому опять возвращаемся к предыдущей вершине C и находим в строке матрицы смежности следующий элемент не равный нулю, он соответствует вершине E. От вершины E переходим к вершине F. В результате, получилась следующая трасса обхода графа G5 в глубину, начиная с вершины A: ***A B C D E F***. Чтобы при обходе графа избегать зацикливаний, которые могут возникнуть из–за многократного посещения вершин, имеющих несколько исходящих ребер, обработанные вершины следует помечать. В результате обхода графа в глубину, пройденные ребра вместе с обработанными вершинами образуют одно или несколько деревьев (если граф имеет несколько компонент связности). Для графа G5 дерево, полученное в результате обхода в глубину, показано на рис. 29:

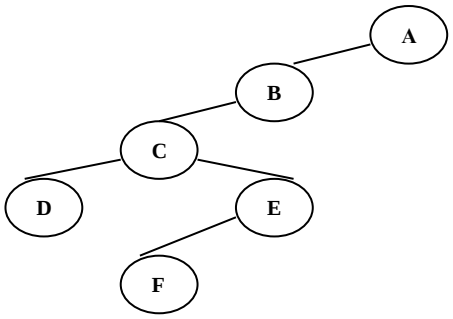


Рис. 29. Дерево, полученное в результате обхода графа G5 в глубину

Для реализации обхода графа в глубину необходимо использовать структуру стека. Реализация алгоритма обхода графа в глубину из заданной вершины приведена ниже. Граф задан с помощью матрицы смежности. Используется вектор посещенных вершин, в котором отмечаются обработанные вершины. Распечатываются номера вершин в порядке их посещения:

```
public class Graph                                  // класс «Графы»
{
  private int size;                                 // количество вершин в графе
  private bool [,] adjacency;                       // матрица смежности графа
  private bool [] vector;                           // вектор посещенных вершин
```

```
  // свойства Size, Adjacency, Vector

  public Graph(int size, bool[,] G)                    // конструктор класса «Графы»
  {
    Adjacency = new bool[size,size];                   // инициализация матрицы смежности
    Adjacency = G;
    for (int i=0; i<size; i++) Vector[i] = false;      // иниц-я вектора посещенных вершин
    Size = size;
  }

public void Depth (int i);                             { i – вершина, с которой начинается обход
{
  Vector[i]:=true;                                     // отметить вершину i как обработанную
   Console.Write( "{0}"  + ' ', i );                   // распечатать номер посещенной вершины
   for (int k=0; k<Size; k++)                          // найти первую встретившуюся ранее не
                                                       посещенную вершину k, смежную с вершиной i
     if ( Adjacency[i,k] && ! (Vector[k]) )
       Depth( k );                                     // перейти к обработке вершины k
}

 static void Main()
 {
   bool[,] M = new bool[4,4]
   {
     {false, true, false, true},                       // матрица смежности графа G4
     {true, false, false, false},
     {true, true, false, true},
     {false, true, false, false}
   };
   Graph G = new Graph( 4,M );
   G.Depth( 1 );                                       // обход графа из вершины номер 1
   Console.ReadLine();
  }
}
```

3.3.2 Алгоритм обхода в ширину

Этот вид обхода соответствует прохождению узлов в горизонтальном порядке (слева направо), в соответствии с матрицей смежности в порядке возрастания длины пути от корня к вершине. Трасса обхода в ширину графа G6 (рис. 30), начиная с вершины A, – ***A B C G H D E F I L J K***.

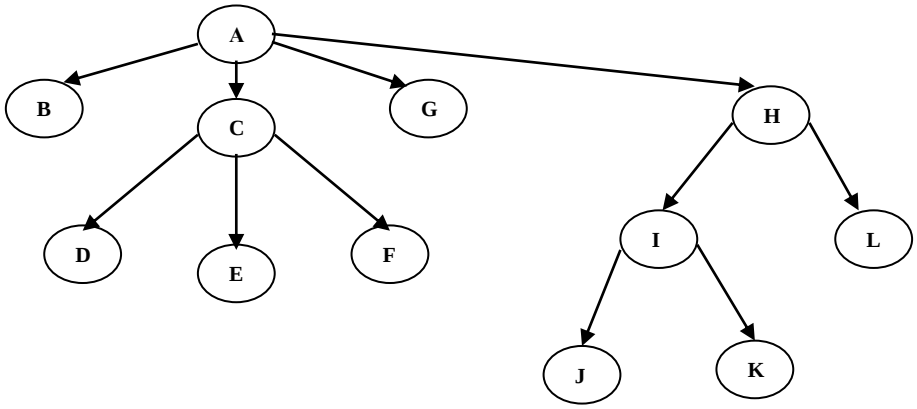


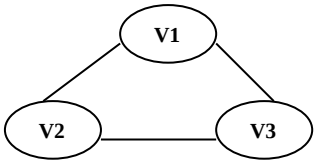
Рис. 30. Граф G6

Для реализации обхода графа в ширину используется структура очереди, в которой должны временно храниться вершины, имеющие смежные вершины на более низком уровне, чтобы на более низком уровне вершины также можно было бы обрабатывать слева направо.

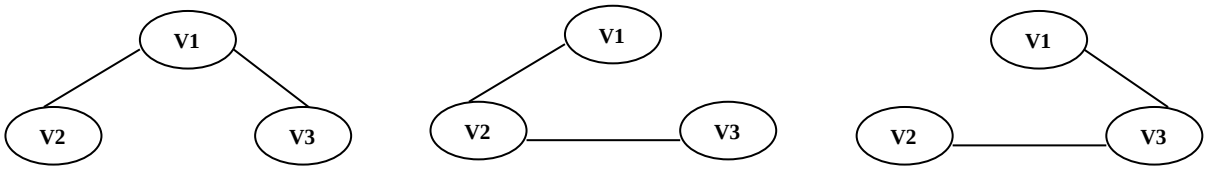
3.4. Остовные деревья

Остовным деревом (или каркасом) G' = (V', E') графа G = (V, E) называется подграф графа G, который является деревом и содержит все вершины графа G, т.е., V' = V, E' ⊂ E. Число различных каркасов полного, связного, неориентированного графа с n вершинами равно n^{n-2}. На рис. 31 показаны различные остовные деревья графа G7:

- v1 – v2; v1 – v3;
- v2 – v1; v2 – v3;
- v3 – v1; v3 – v2.



а



б

Рис. 31. Граф G7 и его остовные деревья
а – граф G7; б – остовные деревья графа G7

Алгоритм построения остовного дерева G ' = (V ', E ') неориентированного графа G = (V, E)

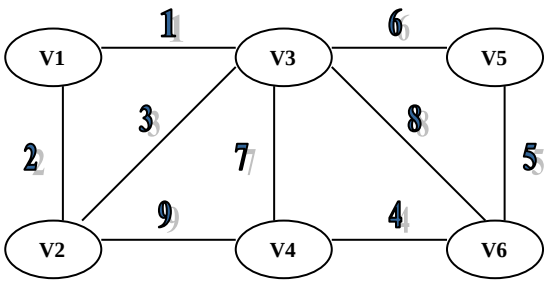
1. Пусть v ∈V – произвольная вершина. V' = [v], E' = [ ] ;
2. Если V’ = V, то G ‘ = (V’, E’) – остовное дерево построено, иначе переход к пункту 3.
3. Найти ребро (i,j)∈ E такое, что (i∈ V') && (j∈ V-V').
   Обновление V' и E': V' = V' ∨ [j]; E' = E' ∨ [(i,j)]. Переход к пункту 2.

Данный алгоритм реализуется на основе обхода графа в глубину с использованием матрицы смежности. Вершины, подключаемые к остовному дереву, помечаются, чтобы избежать зацикливаний. Остовное дерево всегда имеет n вершин и (n-1) ребро. Для хранения остовного дерева, следует использовать структуру двумерного массива следующего вида, позволяющую хранить ребра остовного дерева:

```
int [,] cover = new int[size-1,2];
```

3.4.1 Остовные деревья минимального веса

Пусть задан связный неориентированный граф G = (V, E). W = $|w_{ij}|$ i=1..n, j=1..n. В весовой матрице обозначим бесконечно большим числом отсутствие ребра между вершинами i и j. ***Остовным деревом минимального веса*** называется остовное дерево с минимальной суммой весов ребер. Существует два алгоритма построения остовных деревьев минимального веса, оба они выполняются за (n-1) итерацию. Рассмотрим алгоритмы построения остовных деревьев на примере графа G8 (рис. 32).



	V1	V2	V3	V4	V5	V6
V1	0	2	1	∞	∞	∞
V2	2	0	3	9	∞	∞
V3	1	3	0	7	6	8
V4	∞	9	7	0	∞	4
V5	∞	∞	6	∞	0	5
V6	∞	∞	8	4	5	0

Рис. 32. Граф G8, представленный матрицей смежности

Алгоритм Краскала

Построение начинается одновременно со всех вершин, которые рассматриваются в качестве отдельных фрагментов. Затем какие либо два фрагмента последовательно соединяются ребром, имеющим минимальный вес среди всех ребер, добавление которых не создает цикл (рис. 33).

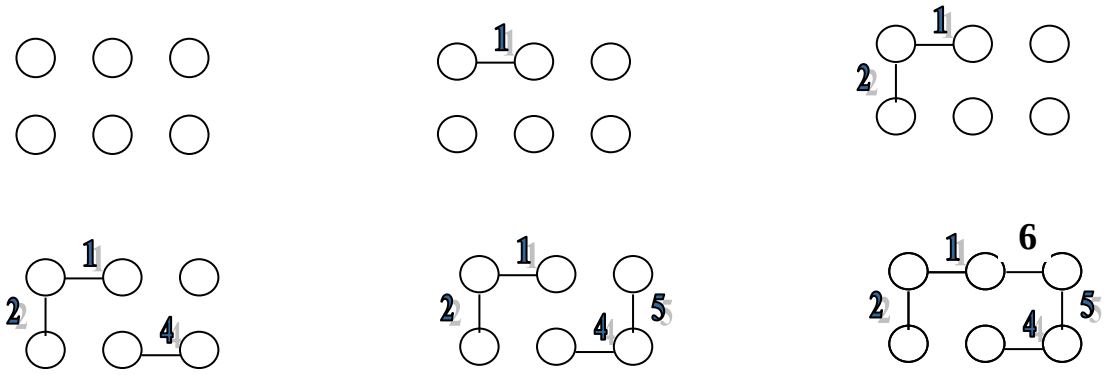


Рис. 33. Алгоритм Краскала

Алгоритм Прим – Дейкстра

Этот алгоритм начинает работу с произвольно выбранной вершины, которая затем соединяется с ближайшей вершиной. Соединенные вершины образуют связное множество, остальные вершины – несвязное множество. На каждом шаге в множество связных вершин добавляется вершина, находящаяся на кратчайшем расстоянии к любой из вершин связного множества. Множества связных и несвязных вершин корректируются, алгоритм заканчивает работу, когда все вершины попадут в множество связных вершин (рис. 34). Построение остовного дерева начнем с вершины V5.

Если веса некоторых ребер совпадают, предпочтение отдается ребру, исходящему из вершины с меньшим номером, а, если ребра исходят из одной и той же вершины, предпочтение отдается ребру, входящему в вершину с меньшим номером.

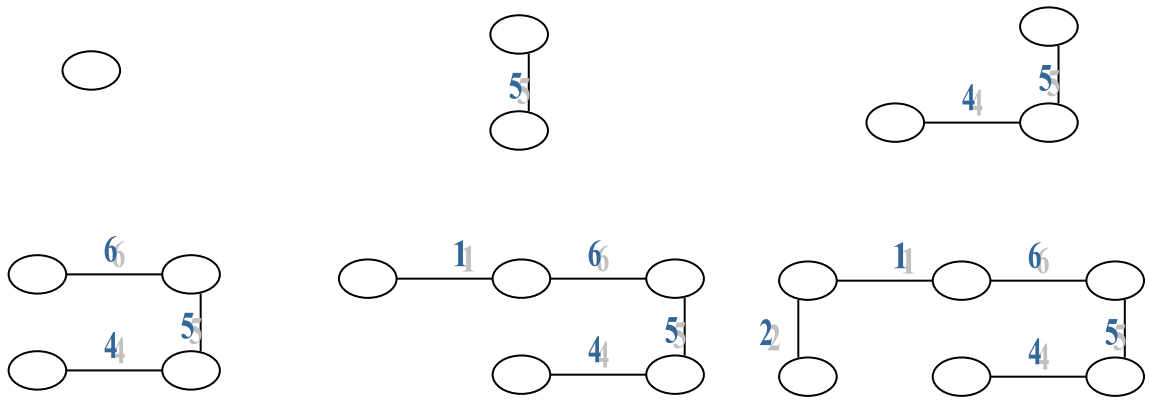


Рис. 34. Алгоритм Прим-Дейкстра

3.5. Алгоритмы нахождения кратчайших путей в графе

Взвешенный орграф G = (V, E) задан весовой матрицей W = $|w_{ij}|$ i=1..n, j=1..n. Заданы начальная и конечная вершины s, t ∈ V. Длина любого

ориентированного пути P = (s, i, j,…, k, t) между вершинами s и t определяется как сумма весов ребер, принадлежащих этому пути:

$$D = w_{si} + w_{ij} + \ldots + w_{kt}.$$

Кратчайший путь между вершинами s и t имеет минимальную длину пути между заданными вершинами s и t. Ребра (i,j) принадлежат пути от вершины s к вершине t:

$$D_{min} = \min \sum_{(i,j) \in P} w_{ij}$$

Все алгоритмы нахождения кратчайших путей определяют оценки от начальной вершины до всех остальных вершин графа и осуществляют построение матрицы кратчайших путей.

Матрица $D(n, n) = |d_{ij}|$ $i = 1..n$, $j = 1..n$ такая, что

$$d_{ij} = \begin{cases} \text{длина кратчайшего пути – путь из вершины i в вершину j существует,} \\ \infty \text{ – путь из вершины i в вершину j не существует,} \end{cases}$$

называется ***матрицей кратчайших путей.***

Наиболее широко распространены два алгоритма нахождения кратчайших путей в графе: алгоритм Флойда и алгоритм Дейкстра.

3.5.1. Алгоритм Флойда

Алгоритм Флойда состоит в последовательном поиске длин кратчайших путей из вершины i в вершину j для всех пар (i,j) при ограничении, что промежуточной вершиной может быть только вершина 1, затем при ограничении, что промежуточными могут быть только вершины 1 и 2 и т.д. Таким образом, поиск кратчайших путей выполняется в порядке увеличения количества промежуточных вершин:

1. [Инициализация]. Установить D = W;
2. [Выполнение прохода]. Повторять шаги 3,4 при k = 1,2,…n;
3. [Обработка строк]. Повторять шаг 4 при i = 1,2,…n;
4. [Обработка столбцов]. Повторять при j = 1,2,…n:
   Установить $d_{ij} = \min [d_{ij}, d_{ik} + d_{kj}]$.
   Два варианта соответствуют использованию и неиспользованию вершины k в качестве промежуточной;
5. [Завершение алгоритма].

Так как каждый из n шагов алгоритма выполняется для каждой пары вершин, объем вычислений составляет $O(n^3)$.

3.5.2. Алгоритм Дейкстра

Алгоритм Дейкстра выполняет поиск длины кратчайшего пути от некоторого выделенного узла (узла 1) до всех остальных узлов графа. Пути определяются в порядке возрастания их длины. Обозначим через D_i длину пути из узла 1 в узел i, S – множество связных вершин, d_{ij} – длину прямого пути из вершины i в вершину j.

Перед началом работы алгоритма $S = [1]$; $D1 = 0$; $Dj = d_{1j}$ для $\forall j \neq 1$.

1. [Поиск следующего ближайшего узла]. Найти вершину $i \notin S$ такую, что $D_i = \min D_j$, $j \notin S$;
   [Обновление множества связных вершин]. $S = S \vee [i]$. Если $S = V$, то кратчайший путь найден, иначе переход к пункту 2;
2. [Обновление длин кратчайших путей]. Для $\forall j \notin S$
   $$D_j = \min [D_j , D_i + d_{ij}].$$
   Переход к пункту 1.

Так как в алгоритме Дейкстра число операций, выполняемых на каждом шаге, пропорционально количеству вершин n, а каждый шаг выполняется (n-1) раз, объем вычислений составляет $O(n^2)$. Применение алгоритма Дейкстра для нахождения всех кратчайших путей в графе потребует объема вычислений $O(n^3)$.

3.6. Контрольные вопросы к главе 3

1. Дайте определение графа.

2. Какие способы представления графов Вам известны?

3. Дайте определение матрицы смежности, весовой матрицы, матрицы достижимости.

4. Дайте определение дерева как частного случая графа.

5. Сформулируйте алгоритм обхода графа в глубину.

6. Сформулируйте алгоритм обхода графа в ширину.

7. Дайте определение остовного дерева графа.

8. Сформулируйте алгоритм Краскала построения остовного дерева графа.

9. Сформулируйте алгоритм Прим-Дейкстра построения остовного дерева графа.

10. Приведите общую постановку задача нахождения кратчайшего пути в графе.

11. Сформулируйте алгоритм Флойда нахождения кратчайшего пути в графе.

12. Сформулируйте алгоритм Дейкстра нахождения кратчайшего пути в графе.

13. Сравните эффективность алгоритмов Флойда и Дейкстра нахождения кратчайшего пути в графе.

3.7. Упражнения к главе 3

1. На основании заданной матрицы смежности графа постройте матрицу достижимости этого графа.

2. Найдите все вершины графа, недостижимые от заданной его вершины.

3. **Обод** - это граф, вершины которого V_0, V_1, ... , V_n (n >= 2) можно занумеровать так, что для всех i (1 <= i <= n-1) вершина V_i соединена ребрами с V_{i-1} и V_{i+1}, вершина V_0 - с V_n, а других ребер нет. Определите, является ли ободом заданный граф.

4. Граф называется ***связным***, если для любой пары вершин существует соединяющий их путь (не обязательно прямой). Определите, является ли связным заданный граф.

5. Определите, является ли заданный граф ***ациклическим*** (т.е., не содержащим циклов).

6. ***Источником*** орграфа называется вершина, от которой достижимы все другие вершины; ***стоком*** - вершина, достижимая от всех других вершин. Найдите все источники и стоки данного орграфа.

7. Реализуйте алгоритм обхода графа в глубину из всех вершин графа.

8. Реализуйте алгоритм обхода графа в ширину из заданной вершины.

9. Реализуйте алгоритм построения остовных деревьев заданного графа.

10. Напишите программу, реализующую алгоритм Флойда поиска в графе длин кратчайших путей, ведущих из вершины i в вершину j для всех пар (i,j).

11. Напишите программу, реализующую алгоритм Дейкстра поиска длин кратчайших путей, ведущих из заданной вершины во все остальные вершины.

4. ОРГАНИЗАЦИЯ МНОЖЕСТВА ОБЪЕКТОВ С ЗАДАННЫМ ОТНОШЕНИЕМ ПОРЯДКА. ХЕШИРОВАНИЕ

4.1. Основные понятия и определения

Пусть задано множество объектов, характеризуемых некоторым уникальным атрибутом, который называется ключом (на множестве ключей задано отношение порядка). Требуется организовать множество объектов так, чтобы поиск объекта с заданным ключом потребовал как можно меньше затрат. Известны способы организации упорядоченных множеств объектов в виде списков и деревьев. Поиск объекта по заданному значению ключа в списке требует в среднем *N/2* операций, где *N* – количество объектов в списке. Поиск в сбалансированном дихотомическом дереве выполняется за $log_2 N$ операций, где *N* – количество объектов в дереве. В каждом из этих методов продолжительность поиска зависит от количества объектов в упорядоченном множестве, а алгоритмы поиска основаны на различных способах организации самих объектов.

Существуют методы организации упорядоченных множеств объектов, для которых продолжительность поиска не зависит от количества объектов *N*. При этом объекты должны быть организованы в виде таблицы, а для определения местоположения каждого объекта следует использовать ключ. Задача представления упорядоченного множества фактически сводится в этом случае к задаче определения отображения *H: K –> A*, где *K* – пространство или множество ключей, которые могут идентифицировать объекты в таблице с прямым доступом, *A* – адресное пространство (множество индексов таблицы), обозначим его 0,1,2, …, m-1. Отображение множества ключей во множество адресов называется преобразованием ключа или **хешированием** (*hashing*), а функция *H*, выполняющая такое преобразование, – **хеш-функцией** (рис. 35).

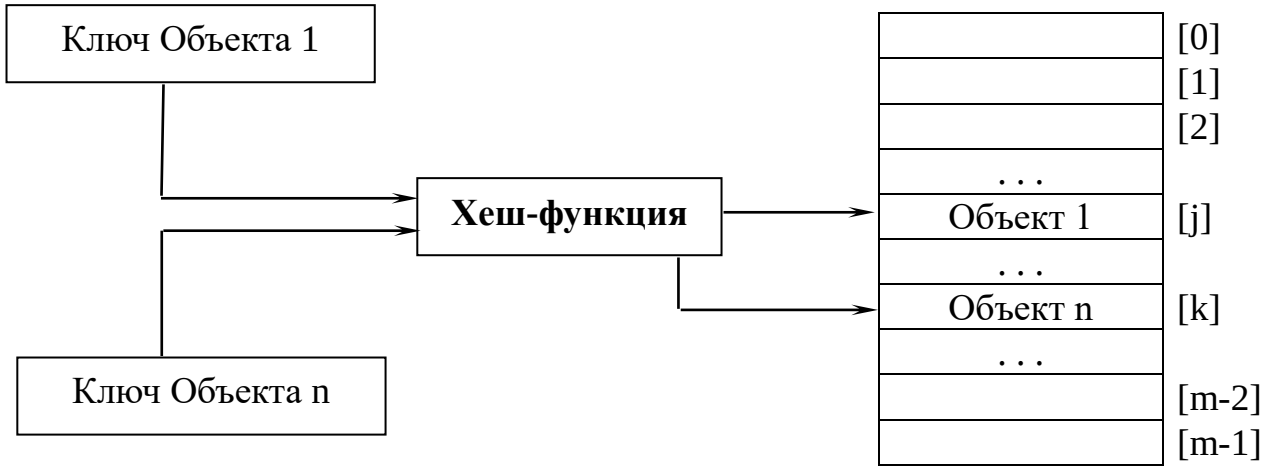


Рис. 35. Графическое представление хеш-функции

Основная трудность преобразования ключей состоит в том, что множество возможных значений ключей намного больше, чем множество адресов таблицы (индексов массива). Например, пусть слова, состоящие из 10 букв английского алфавита, используются для идентификации конкретного человека во множестве из 1000 человек. При этом имеются 26^{10} возможных ключей, которые должны отображаться в 10^3 возможных индексов. Следовательно, *H* – преобразование «многие к одному».

Алгоритмы хеширования состоят из двух компонентов: функции хеширования, определяющей преобразование пространства ключей в адресное пространство, и средств разрешений коллизий, устраняющих конфликты, возникающие вследствие преобразования нескольких ключей в один адрес.

4.2. Выбор функции преобразования

Каждый элемент пространства ключей *K* является цифровым, алфавитным или алфавитно-цифровым идентификатором. Основное требование к хорошей функции преобразования состоит в том, чтобы она распределяла ключи как можно более равномерно по шкале значений индексов. Желательно, чтобы распределение производило впечатление совершенно случайного. Хеш-функция должна эффективно вычисляться, т.е., состоять из небольшого количества основных арифметических и логических операций.

Наиболее широко распространенная функция хеширования основывается на методе деления. Пусть имеется функция *ord(k)*, которая определяет порядковый номер ключа *k* во множестве всех возможных значений ключей. Индексы таблицы находятся в диапазоне 0..m-1, где m – размер таблицы. Тогда значение хеш-функции вычисляется следующим образом:

$$H(k) = ord(k) \% m. \qquad (1)$$

Если ключ *k* является последовательностью символов, то порядковые номера символов предварительно суммируются, а затем выполняется деление. При отображении ключей в адреса до некоторой степени сохраняется существующая во множестве ключей равномерность распределения. Ключи с близкими значениями отображаются в уникальные адреса. Например, при делителе 10 ключи 2000, 2001, … ,2009 отобразятся в адреса 0,1, … ,9. Но ключи 3010, 3011, …, 3019 также отобразятся в адреса 0,1, … ,9, т.к. ключи из этих двух групп совпадают по модулю *m*. Если *m* является простым числом с большим значением, ключи не будут совпадать по модулю *m*, поэтому в качестве делителя следует выбирать простое число.

4.3. Разрешение коллизий

При добавлении элемента в хеш-таблицу коллизия создает основную проблему всей операции. Без коллизии добавляемый элемент размещается по индексу, непосредственно вычисленному с помощью хеш-функции. При наличии коллизии нужный индекс уже занят, поэтому необходимо найти некоторую другую ячейку для размещения объекта, т.е., вычислить другой индекс *h*, который однозначно получается на основе данного ключа. Подобное корректирующее действие называется **разрешением коллизии** (*collision resolution*).

Существует несколько стратегий разрешения коллизии:

- метод открытой адресации,
- метод цепочек.

4.3.1. Метод открытой адресации

Согласно методу открытой адресации, если ключ *k* отображается в табличный индекс *i*, а он уже занят, в таблице просматриваются другие индексы до тех пор, пока не будет найдено свободное место для объекта, приведшего к коллизии (*G(i)* – некоторая функция):

$$i = i + 1;\ h = H(k) + G(i); \qquad (2)$$

Для определения функции *G(i)* можно использовать

- линейную проверку (*linear probing*),
- квадратичную проверку (*quadratic probing*),
- повторное хеширование (*rehashing*), называемое также двойным хешированием (*double hashing*).

При **линейной проверке** используется следующая последовательность индексов в предположении, что таблица круговая:

$$i, i+1, \ldots, m-1, 0, 1, 2, \ldots, i-1.$$

Такая последовательность получается, если функция *G(i) = i*, индексы h_i, используемые для поиска, имеют вид:

$$h_0 = H(k), \qquad (3)$$
$$h_i = (h_0 + i) \% m,\ i = 1, \ldots, m-1.$$

Линейная проверка, хотя и проста в реализации, не является хорошей стратегией разрешения коллизий, т.к. ведет к **кластеризации**, при которой вокруг первичных ключей (ключей, для которых конфликта при занесении в таблицу не было) возникают все более длинные последовательности занятых подряд индексов. Например, пусть надо добавить в таблицу Объект 1 с

ключом 1234, Объект 2 с ключом 4674, Объект 3 с ключом 6827, Объект 4 с ключом 3235, Объект 5 с ключом 7165, размер таблицы *m =10*, хеш-функция вычисляется по формуле (3). Объект 1 будет размещен в ячейке с индексом 4. Объект 2 также должен быть размещен в элементе с индексом 4, но этот индекс уже занят, поэтому Объект 2 занимает следующий свободный элемент с индексом 5. Объект 3 добавляется в элемент с индексом 7. Объект 4 должен быть добавлен в элемент с индексом 5, но этот индекс уже занят, поэтому Объект 4 занимает следующий свободный элемент с индексом 6. Объект 5 должен быть добавлен в элемент с индексом 5, но этот индекс уже занят, следующие элементы с индексами 6 и 7 также заняты, поэтому Объект 5 занимает свободный элемент с индексом 8.

Коллизии усложняют не только процесс вставки объекта в хеш-таблицу, но также и процесс поиска объекта по ключу. Например, если необходимо найти информацию об Объекте 5, то по его ключу 7165 вычисляется значение хеш-функции, равное 5. Но в элементе с индексом 5 находится Объект 2. Линейный поиск следует продолжать до тех пор, пока не будет найден объект с заданным ключом или пустой элемент таблицы. Если в таблице найден пустой элемент, это означает, что в ней не содержится объект с заданным ключом.

Избежать кластеризации позволяет **квадратичная проверка**, выполняющая просмотр индексов таблицы на квадратичном расстоянии (функция $G(i) = i^2$):

$$h_0 = H(k), \qquad (4)$$
$$h_i = (h_0 + i^2) \% m, \; i = 1, \ldots, m-1.$$

Однако даже квадратичная проверка может привести к кластеризации. Кроме того, недостаток этой стратегии заключается в том, что при добавлении объекта можно пропустить свободный элемент. Фактически при квадратичной проверке используется половина таблицы.

Повторное хеширование работает следующим образом: пусть имеется набор различных хеш-функций $H_1, \ldots, H_n$ и при вставке или извлечении объекта из хеш-таблицы первоначально используется функция H_1. Если это приводит к коллизии, то производится попытка использования хеш-функции H_2 и так далее до H_n при необходимости. При повторном хешировании очень важно, чтобы каждый элемент хеш-таблицы просматривался ровно один раз, когда делается m просмотров, где m – размер таблицы. Это означает, что функции H_i и H_j не должны выдавать один и тот же индекс таблицы. Такой результат достигается, если значения, получаемые при вычислении функций H_i и H_j, являются взаимно простыми числами, что гарантировано, если размер таблицы m – простое число. Повторное хеширование обеспечивает лучшую технику избежания коллизий, чем линейная или квадратичная проверка.

Для анализа методов устранения коллизий используется вероятностная модель, на основе которой получаются формулы для средней длины успешного и неуспешного поиска. Предполагается, что каждый ключ может быть отображен в каждый из элементов хеш-таблицы с равной вероятностью *1 / m*. Следовательно, существует m^N различных вариантов отображения значений ключей в адресное пространство, где *N* – количество объектов. Средняя длина поиска (*ALoS – Average Length of Searching*) зависит от **коэффициента заполнения** таблицы $\alpha = N / m$, определяемого как отношение занятого и общего адресного пространства хеш-таблицы. Средняя длина поиска при линейной проверке вычисляется по формуле (5).

$$\mathrm{ALoS}_{\text{linear probing}} \cong \begin{cases} \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) & \text{при удачном поиске} \\ \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right) & \text{при неудачном поиске} \end{cases} \qquad (5)$$

4.3.2. Реализация хеш-таблицы (класс *Hashtable*)

.Net Framework Base Class Library содержит реализацию хеш-таблицы в виде класса *System.Collections.Hashtable*, который предоставляет коллекцию пар «ключ-значение», упорядоченных по хеш-коду ключа. И ключ, и значение могут быть любого типа. В классе *Hashtable* реализованы стандартные операции по работе с хеш-таблицей (Таблица 4).

Таблица 4. Методы и свойства класса *Hashtable*

Сигнатура	Описание
public Hashtable(int capacity);	создает и инициализирует новый экземпляр класса *Hashtable* с заданным количеством элементов. *Capacity* – минимальное число элементов, которые должен содержать экземпляр класса *Hashtable*
public Hashtable();	создает и инициализирует новый экземпляр класса *Hashtable* с количеством элементов по умолчанию
public virtual void Add(object key, object value);	добавляет в текущий экземпляр класса *Hashtable* элемент с ключом *key* и значением *value*
public virtual void Clear();	удаляет все элементы из текущего экземпляра класса *Hashtable*

public virtual object Clone();	создает объект класса *Object*, который является копией текущего экземпляра класса *Hashtable*
public virtual bool Contains(object key);	определяет, содержит ли текущий экземпляр класса *Hashtable* заданный ключ *key*. Возвращает *true*, если текущий экземпляр класса *Hashtable* содержит заданный ключ *key*
public virtual bool ContainsKey(object key);	определяет, содержит ли текущий экземпляр класса *Hashtable* элемент с заданным ключом *key*. Возвращает *true*, если текущий экземпляр класса *Hashtable* содержит элемент с заданным ключом *key*
public virtual bool ContainsValue(object value);	определяет, содержит ли текущий экземпляр класса *Hashtable* элемент с заданным значением *value*. Возвращает *true*, если текущий экземпляр класса *Hashtable* содержит элемент с заданным значением *value*
public virtual void CopyTo(Array array, int arrayIndex);	копирует элементы текущего экземпляра класса *Hashtable* в одномерный массив *array*, начиная с индекса *arrayIndex*
protected virtual int GetHash(object key);	генерирует хеш-код для заданного ключа *key* текущего экземпляра класса *Hashtable*
public virtual void Remove(object key);	удаляет элемент с заданным ключом *key* из текущего экземпляра класса *Hashtable*
public virtual int Count { get; }	свойство возвращает число пар ключ/значение, содержащихся в текущем экземпляре класса *Hashtable*
public virtual ICollection Keys { get; }	свойство возвращает коллекцию *ICollection*, содержащую ключи элементов текущего экземпляра класса *Hashtable*
ICollection IDictionary.Keys { get; }	свойство реализует поддержку интерфейса *IDictionary*, возвращает пары ключ/значение
public virtual ICollection Values { get; }	свойство возвращает коллекцию *ICollection*, содержащую значения элементов текущего экземпляра класса *Hashtable*

ICollection IDictionary.Values { get; }	свойство реализует поддержку интерфейса *IDictionary*, возвращает пары ключ/значение

Объекты в хеш-таблицу добавляются с помощью метода *Add()*. Для извлечения объекта из хеш-таблицы в качестве индекса используется ключ, как и в случае индексирования массива порядковым значением. Метод *ContainsKey()* возвращает логическое значение, указывающее, был ли найден данный ключ в хеш-таблице. Класс *Hashtable* содержит также свойство *Keys*, которое возвращает набор ключей, используемых в хеш-таблице. Это свойство может использоваться для перечисления элементов в хеш-таблице. Порядок, в котором добавляются объекты, и порядок ключей в коллекции *Keys* не обязательно совпадают. Индекс элемента таблицы, в котором хранится объект, зависит от хеш значения ключа и стратегии разрешения коллизий.

Пример работы с объектом класса *Hashtable* приведен ниже.

Функция хеширования класса *Hashtable*

Функция хеширования реализована в методе *GetHashCode()*, который определен в классе *System.Object*. Реализация метода в классе *Object* возвращает уникальное целое число, которое не изменяется за время жизни объекта. Т.к. каждый объект является наследником, прямым или непрямым, класса *Object*, все объекты имеют доступ к этому методу. Поэтому объект любого типа может быть представлен уникальным целым числом.

Функция хеширования класса *Hashtable* определена следующим образом (6):

$$H(key) = [\ GetHash(key) + 1 + (((GetHash(key) >> 5) + 1)\ \%\ (\ hashsize - 1\ ))\ ]\ \%\ hashsize \qquad (6)$$

GetHash(key) – по умолчанию, результат, возвращаемый вывовом *GetHashCode()*. *GetHash(key) >> 5* вычисляет хеш-код для ключа *key* и затем сдвигает результат на 5 разрядов вправо, что эквивалентно выполнению операции деления на 32. Оператор *%* выполняет модульную арифметику, *hashsize* – общее число элементов хеш-таблицы. Благодаря операции взятия остатка от деления, *H(key)* всегда будет целым числом в диапазоне от 0 до *hashsize-1*. Т.к. *hashsize* – общее число элементов хеш-таблицы, результирующее хеш-значение всегда будет указывать на допустимый индекс таблицы.

Разрешение коллизий в классе *Hashtable*

Для разрешения коллизий в классе *Hashtable* используется повторное хеширование. Функция H_1 определяется по формуле (6). Другие хеш-

функции отличаются от (6) только коэффициентом умножения *k*. В общем случае хеш-функция определяется следующим образом (7):

$$H(key) = [\ GetHash(key) + k * (\ 1 + (((GetHash(key) >> 5) + 1)\ \%\ (\ hashsize - 1\ )))\ ]\ \%\ hashsize \quad (7)$$

Коэффициент заполнения и расширение хеш-таблицы

Класс *Hashtable* содержит скрытый член класса, называемый *LoadFactor*, который задает коэффициент заполнения хеш-таблицы α (значение в диапазоне от 0.1 до 1.0). Коэффициент заполнения, равный 0.5, означает, что хеш-таблица может содержать до половины своих элементов незаполненными. Всякий раз, когда в хеш-таблицу добавляется новое значение, производится проверка, не приведет ли эта операция к превышению максимального значения коэффициента заполнения. Если операция добавления объекта приведет к такому увеличению, хеш-таблица будет расширена согласно следующему алгоритму:

1. Число элементов хеш-таблицы приблизительно удваивается. Более точно, число элементов увеличивается от текущего значения в виде простого числа до следующего простого числа.
2. Т.к. хеш-значение каждого элемента хеш-таблицы зависит от общего числа элементов хеш-таблицы, все хеш-значения необходимо повторно вычислить.

Коэффициент заполнения влияет на ожидаемое количество просмотров таблицы при возникновении коллизии. Более высокий коэффициент заполнения, который позволяет иметь относительно плотную хеш-таблицу, требует меньше памяти, но большего числа просмотров при возникновении коллизии, чем разреженная хеш-таблица.

Расширение хеш-таблицы является дорогой операцией. Поэтому, если имеется предварительная оценка количества объектов, которые должны храниться в хеш-таблице, можно задать начальную емкость хеш-таблицы в конструкторе (*capacity*), чтобы избежать нежелательных увеличений ее размера впоследствии.

4.3.3. Метод цепочек

Согласно методу цепочек (*chaining*), для разрешения коллизий все множество ключей разбивается на несколько непересекающихся подмножеств, называемых классами эквивалентности. Два ключа попадают в один и тот же класс эквивалентности, если хеш-функция преобразует их в один и тот же индекс таблицы. Каждый класс эквивалентности хранится в виде отдельного списка. В случае коллизии объект, приводящий к коллизии, добавляется в конец соответствующего списка. Структура, представляющая хеш-таблицу в виде связанных списков объектов, называется ***словарем*** (*Dictionary*). Структура словаря, использующего функцию хеширования (1), где размер таблицы m = 10, показана на рис. 36.

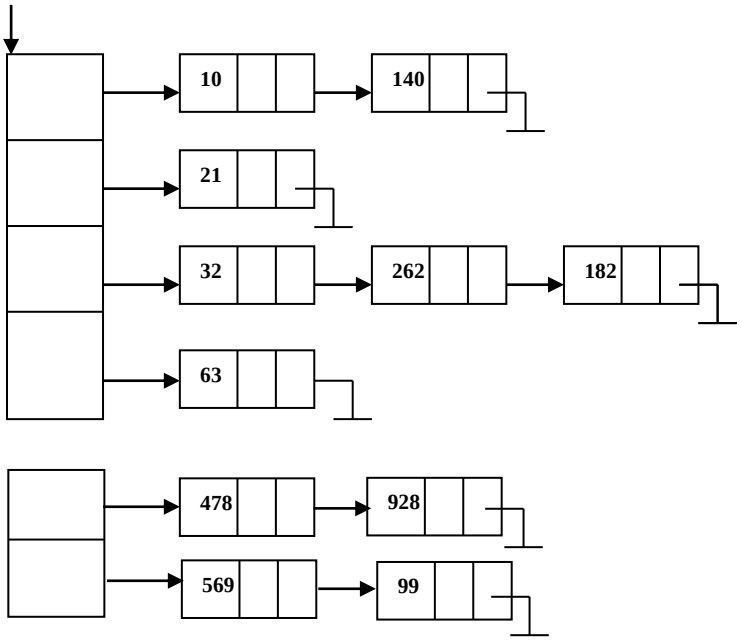


Рис. 36. Структура словаря

Средняя длина поиска при использовании метода цепочек определяется по формуле (8).

$$\mathrm{ALoS}_{\mathrm{chaining}} \cong \begin{cases} 1+\dfrac{\alpha}{2} & \text{при удачном поиске} \\ \alpha + e^{-\alpha} & \text{при неудачном поиске} \end{cases} \qquad (8)$$

4.3.4. Реализация словаря (класс *Dictionary*)

.Net Framework Base Class Library содержит реализацию хеш-таблицы в виде класса *System.Collections.Generic.Dictionary*. Хеш-таблица (*Hashtable*) является слабо типизированной структурой данных, т.к. в хеш-таблицу можно добавлять ключи и значения любого типа. Класс *Dictionary* имеет сильную типизацию как для ключей, так и для значений. При создании экземпляра класса *Dictionary* необходимо указать типы данных и для ключа, и для значения следующим образом:

```
Dictionary<Tkey, TValue>  variableName =
new Dictionary<Tkey, TValue> ();
```

Объект класса *Dictionary* относится к типу *KeyValuePair<(Of<(Tkey, TValue>)>)*, представляющему пару из типа ключа и типа значения. В классе *Dictionary* реализованы операции по работе с хеш-таблицей (Таблица 5).

Таблица 5. Методы класса *Dictionary*

Сигнатура	Описание
Add	добавляет указанные ключ и значение в словарь
Clear	удаляет все ключи и значения из словаря *Dictionary<(Of <(TKey, TValue>)>)*
ContainsKey	определяет, содержится ли указанный ключ в словаре *Dictionary<(Of <(TKey, TValue>)>)*
ContainsValue	определяет, содержит ли коллекция *Dictionary<(Of <(TKey, TValue>)>)* указанное значение
Equals	определяет, равен ли заданный объект *Object* текущему объекту *Object* (унаследовано от *Object*)
Finalize	позволяет объекту *Object* попытаться освободить ресурсы и выполнить другие операции очистки, перед тем как объект *Object* будет утилизирован в процессе сборки мусора (унаследовано от *Object*)
GetEnumerator	возвращает перечислитель, осуществляющий перебор элементов словаря *Dictionary<(Of <(TKey, TValue>)>)*
GetHashCode	играет роль хеш-функции для определенного типа (унаследовано от *Object*)
GetType	возвращает объект *Type* для текущего экземпляра (унаследовано от *Object*)
MemberwiseClone	создает неполную копию текущего объекта *Object* (унаследовано от *Object*)
Remove	удаляет значение с указанным ключом из словаря *Dictionary<(Of <(TKey, TValue>)>)*
ToString	возвращает объект *String*, который представляет текущий объект *Object* (унаследовано от *Object*)
TryGetValue	получает значение, связанное с указанным ключом

Свойства	
Compare	Возвращает компаратор *IEqualityComparer<(Of <T>)>)*, используемый для установления равенства ключей словаря
Count	Возвращает число пар "ключ-значение", содержащихся в словаре *Dictionary<(Of <(TKey, TValue>)>)*
Item	Возвращает или задает значение, связанное с указанным ключом
Keys	Получает коллекцию, содержащую ключи из словаря *Dictionary<(Of <(TKey, TValue>)>)*
Values	Получает коллекцию, содержащую значения в объекте *Dictionary<(Of <(TKey, TValue>)>)*

Таким образом, хеш-таблица расширяет массив (*ArrayList*), реализуя индексирование элементов с помощью произвольного ключа, в отличие от индексирования порядковым значением. Поиск по уникальному ключу эффективнее производится в хеш-таблице, чем в массиве, т.к. в хеш-таблице поиск выполняется за константное время в отличие от линейного поиска в массиве. Словарь представляет собой типобезопасную хеш-таблицу с альтернативной стратегией разрешения коллизий.

4.4. Контрольные вопросы к главе 4

1. Что такое хеш-таблица, ключ хеш-таблицы, хеширование, хеш-функция? В чем заключается основная проблема преобразования ключей?

2. По каким критериям выбирается функция преобразования?

3. Как работает функция преобразования по методу деления?

4. В чем причина возникновения коллизий при хешировании? В чем заключается разрешение коллизии?

5. Как работает метод открытой адресации?

6. Как выполняется линейная, квадратичная проверка, повторное хеширование?

7. Как вычисляется коэффициент заполнения хеш-таблицы? На какую характеристику поиска он влияет?

8. Как работает метод цепочек?

ЗАКЛЮЧЕНИЕ

Учебное пособие “Структуры данных в C#. Часть II. Нелинейные динамические структуры” посвящено рассмотрению структур данных и алгоритмов, которые являются фундаментом современной методологии разработки программ.

Учебное пособие включает разделы, которые подробно описывают рекурсивные структуры данных, иерархические структуры (деревья, графы). Учебное пособие отличается методикой изложения всех разделов с точки зрения особенностей внутреннего представления структур данных различных видов в оперативной памяти компьютера, что способствует лучшему пониманию алгоритмов их обработки. Теоретический материал иллюстрируется большим количеством примеров программ, реализующих алгоритмы обработки различных структур данных. Каждый раздел содержит большое количество примеров программ обработки данных различной структуры на языке C#. Логическим завершением каждого раздела являются контрольные вопросы и упражнения для самостоятельной работы студентов.

Вопросы, имеющие практическое значение для студентов при выполнении лабораторных работ и курсового проекта, освещены в учебном пособии с необходимой для использования полнотой.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Вирт, Н. Алгоритмы и структуры данных: пер. с англ. / Н. Вирт. – Изд. 2-е, испр. – СПб.: Невский диалект, 2005. – 352 с.
2. Ахо, А. Структуры данных и алгоритмы / А.Ахо, Д. Хопкрофт, Д.Ульман; пер. с англ. – М.: Вильямс, 2016. – 400 с.
3. Кнут, Д. Искусство программирования для ЭВМ. В 3 т. Т 1 / Д. Кнут; пер. с англ. – М.: Вильямс, 2000. – 720 с.
4. Павловская, Т.А. C#. Программирование на языке высокого уровня: учебник для вузов / Т.А. Павловская. – СПб.: Питер, 2014. – 432 с.
5. Фаронов, В.В. Создание приложений с помощью C#. Руководство программиста / В.В. Фаронов. – М.: Эксмо, 2008. – 576 с.
6. Биллиг, В.А.Основы программирования на C#: учеб. пособие / В.А. Биллиг. – М.: Интернет-университет информационных технологий; БИНОМ. Лаборатория знаний, 2009. – 483 с.
7. Шилдт, Герберт. C# 3.0: руководство для начинающих / Герберт Шилдт; пер. с англ. – М.: ООО «И.Д. Вильямс», 2009. – 688 с.
8. Шилдт, Герберт. C# 4.0: Полное руководство / Герберт Шилдт; пер. с англ. – М.: Издательский дом «Вильямс», 2011. – 1056 с.
9. Гросс, Кристиан. C# 2008 и платформа NET 3.5 Framework: базовое руководство / Кристиан Гросс; пер. с англ. – М.: Издательский дом «Вильямс», 2009. – 480 с.
10. Петцольд, Чарльз. Программирование для Microsoft Windows 8 / Чарльз Петцольд; пер. с англ. – СПб.: Издательский дом «Питер», 2014. – 1008 с.
11. Стиллмен, Эндрю. Изучаем C# / Эндрю Стиллмен, Грин Дженнифер; пер. с англ. – СПб.: Издательский дом «Питер», 2017. – 816 с.
12. Симонова, Е.В. Структуры данных. Часть I. Линейные динамические структуры: учеб. пособие для вузов / Е.В. Симонова. – Самара: Изд-во СГАУ, 2006. – 82 с.
13. Симонова, Е.В. Структуры данных. Часть II. Нелинейные динамические структуры: учеб. пособие для вузов / Е.В. Симонова. – Самара: Изд-во СГАУ, 2007. – 80 с.
14. Шень, А. Программирование. Теоремы и задачи: учеб. пособие для вузов/ А.Шень. – М.: МЦНМО, 2007. – 296 с.
15. Кормен, Т. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест; пер. с англ. – М.: МЦНМО, 2000. – 960 с.