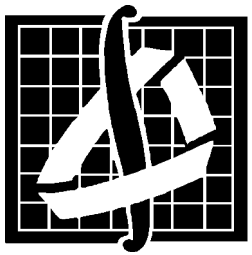
МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В. ЛОМОНОСОВА



Механико-математический факультет
Кафедра вычислительной математики

В. Д. Валединский, А. А. Корнев

Методы программирования в примерах и задачах

Москва 2000 год

ББК 32.97
Б 30
УДК 681.3

В. Д. Валединский, А. А. Корнев. Методы программирования в примерах и задачах. — М.: Изд-во механико-математического ф-та МГУ, 2000, – 152 с.

Данное пособие является обобщением опыта преподавания курса программирования для студентов 1 и 2 курсов механико-математического факультета МГУ. Здесь предпринята попытка собрать и до некоторой степени систематизировать задачи, предлагавшиеся для самостоятельного решения во время семинарских занятий, лабораторных работ, на зачетах и экзаменах. Поскольку основной целью данного пособия является обучение программированию, то помимо формулировок задач здесь даются необходимые определения, обсуждаются идеи построения алгоритмов, для некоторых задач приводятся решения или отдельные фрагменты программ. Некоторые из приведенных задач совершенно конкретны — нужно выполнить ту или иную обработку данных по определенному алгоритму, другие наоборот, сформулированы нечетко и обрисовывают, скорее, круг задач и возможных подходов к решению.

Отдавая отчет в том, что стиль программирования — неисчерпаемая тема тема для споров и дискуссий, авторы все же предлагают свои реализации как пример для подражания, надеясь, что при критическом отношении к приведенному коду читатель сможет вынести для себя много полезного из этой книги.

Рецензент: Академик РАН Н.С. Бахвалов

Б $\frac{\textbf{1702070000 – 004}}{\textbf{Ш7(03) – 99}}$ **– без объявления**

ISBN 5-87597-046-4

Оглавление

1. Простейшие алгоритмические задачи

1.1. Задачи на обработку последовательности

Общая постановка задач этого раздела выглядит следующим образом: имеется некоторое количество однотипных данных (например, чисел), и требуется вычислить некоторую характеристику этого набора (функцию от этих данных). Специфика задачи состоит в том, что мы не можем сохранить весь набор данных в памяти, поскольку нам заранее неизвестно количество элементов данных в этом наборе. Таким образом, нужно построить алгоритм, вычисляющий необходимую характеристику за один проход (просмотр) последовательности. Допускается сохранение и пересчет лишь конечного набора промежуточных значений, на основе которых в любой момент можно вычислить требуемую характеристику. Подробное изложение вопросов, связанных с возможностью построения подобных алгоритмов, см. в [1].

Типичной задачей подобного рода является обработка данных, последовательно вводимых с клавиатуры.

1.1. Составьте программу для вычисления среднего арифметического элементов числовой последовательности, вводимых с клавиатуры.

Решение. Приведем полное решение этой задачи.

```
#include <stdio.h>
int  main ()
{  double  x, res, sum=0;
   int     cnt;
     printf("  ----Вычисление среднего арифметического---- \n ");
     printf("Вводите элементы последовательности по одному \n ");
     printf("В конце последовательности введите любую букву\n ");
   for (cnt=0; scanf("%lf",&x)==1; cnt++)
      sum += x;
    res= (cnt) ? sum/cnt : 0;
   printf("The result is %lf\n",res);
   return 0; }
```

Ввод с клавиатуры весьма трудоемкая процедура, особенно для больших наборов данных, поэтому в дальнейшем предполагается, что данные (последовательность чисел неизвестной длины, возможно, пустая),

записаны в некотором файле. Требуется за один просмотр файла и без запоминания всей последовательности в массиве определить требуемую характеристику последовательности.

Решение задач следует оформить в виде функции, получающей в качестве параметра указатель на открытый файл с данными и возвращающей требуемое значение. Функция main должна считать имя файла данных с экрана, открыть файл, выполнить необходимые проверки и обратиться к этой функции, а затем вывести результат на экран.

1.2. Составьте программу для проверки является ли последовательность арифметической прогрессией.

Решение. Реализуем функцию, возвращающую следующие значения:
1 — последовательность является арифметической прогрессией,
0 — последовательность не является арифметической прогрессией,
-1 — в файле недостаточно данных.

```
#include <stdio.h>
int   progression (FILE *fin);

int  main ()
{
   char     filename [256];
   int  res;
   FILE    *fin;
   printf("Введите имя файла ->");
   scanf("%s",filename);
   fin = fopen (filename,"r");
   if (!fin) { printf("Не удается открыть файл %s\n",filename);
               return -1; }
   res = progression (fin);
       switch(res)
   {
    case 1:
 printf("Последовательность является арифметической прогрессией\n");
 break;
    case 0:
 printf("Последовательность не является арифметической прогрессией\n
 break;
    case -1: printf("В файле не достаточно данных\n");
           break;
```

```
    }
   fclose(fin);
   return 0;
}

int   progression (FILE *fin)
{
   double x,y,d;
   if ( fscanf(fin,"%lf",&y)!=1 ) return -1;
   if ( fscanf(fin,"%lf",&x)!=1 ) return -1;
   d = x-y;
   while ( fscanf(fin,"%lf",&y)==1 )
   {  if ( y-x != d ) return 0;
      x = y;
   }
   return 1;
}
```

1.3. Составьте программу для проверки является ли последовательность геометрической прогрессией.

1.4. Обозначим элементы последовательности x_k, $k = 0,1,\ldots$. Введите с клавиатуры четыре числа a,b,c,d и проверьте, удовлетворяют ли элементы последовательности рекуррентному соотношению $ax_k + bx_{k+1} + cx_{k+2} = d$.

1.5. Определите количество чисел в последовательности, которые больше предшествующего числа.

1.6. Введите с клавиатуры число x и определите, сколько раз оно встречается в последовательности.

1.7. Введите с клавиатуры число x и определите порядковый номер первого числа, равного x.

1.8. Введите с клавиатуры число x и определите порядковый номер последнего числа, равного x.

1.9. Определите все ли элементы последовательности равны между собой.

1.10. Подсчитайте количество положительных, отрицательных и нулевых чисел последовательности.

1.11. Определите является ли последовательность возрастающей, убывающей или ни той, ни другой.

1.12. Определите номер первого и последнего максимального элемента последовательности.

1.13. Определите значение минимального (или максимального) элемента последовательности.

1.14. Определите порядковый номер первого числа, равного максимуму из всех чисел последовательности.

1.15. Определите количество чисел, равных минимуму из всех чисел последовательности.

1.16. Пусть последовательность является неубывающей. Определите количество различных элементов этой последовательности.

1.17. Пусть последовательность является неубывающей. Определите количество элементов, которые появляются в этой последовательности более k раз, (значение k вводится с клавиатуры).

1.18. Вычислите среднее квадратическое отклонение D от среднего арифметического M для элементов последовательности, т.е. суммы

$$D = \frac{1}{N}\sum_{i=1}^{N}(x_i - M)^2, \quad M = \frac{1}{N}\sum_{i=1}^{N} x_i,$$

где x_i — элементы последовательности.

1.19. Определите, сколько раз во входной последовательности встречается подпоследовательность $1,2,3,\ldots,10$.

1.20. Определите длину наибольшего постоянного участка, т.е. максимальное количество подряд идущих элементов в одним и тем же значением.

1.21. Введите с клавиатуры число n и определите количество постоянных участков последовательности, имеющих длину не меньше n.

1.22. Определите сколько элементов последовательности не содержится в постоянных участках длины 2 и более.

1.23. Определите длину наибольшего возрастающего участка последовательности.

1.24. Введите с клавиатуры число n и определите количество невозрастающих участков последовательности, имеющих длину не меньше n.

1.25. Назовем локальным максимумом такой элемент x_i последовательности, для которого выполнено $x_{i-1} < x_i \geq x_{i+1}$, $i > 0$. Определите максимальное расстояние между локальными максимумами элементов последовательности.

1.26. Назовем локальным строгим экстремумом такой элемент x_i последовательности, для которого выполнено $x_{i-1} < x_i > x_{i+1}$ либо $x_{i-1} > x_i < x_{i+1}$, $i > 0$. Определите и напечатайте локальные экстремумы с соседними точками. Каждая тройка печатается с новой строки.

1.27. Определите и напечатайте все отрезки возрастания последовательности. Каждый участок печатается с новой строки.

1.28. Определите и напечатайте все отрезки монотонности последовательности с явным указанием типа монотонности. Каждый отрезок печатается с новой строки.

1.29. Последовательность чисел представляет собой коэффициенты многочлена, расположенные в порядке возрастания степеней. Введите с клавиатуры число x и вычислите значение многочлена и его производной в точке x.

1.30. Последовательность чисел представляет собой коэффициенты многочлена, расположенные в порядке убывания степеней. Введите с клавиатуры число x и вычислите значение многочлена и его производной в точке x.

Идеи реализации. Воспользуйтесь схемой Горнера вычисления многочлена и преобразуйте ее в рекуррентные соотношения, связывающее значения многочлена и его производной. Если обозначить a_k — k-й элемент последовательности коэффициентов, $P_k(x) = a_0x^k + a_1x^{k-1} + \ldots + a_k$ — значение многочлена k-й степени в точке x, $D_k(x) = ka_0x^{k-1} + (k-1)a_1x^{k-2} + \ldots + a_{k-1}$ — значение производной многочлена $P_k(x)$, то данные рекуррентные соотношения будут иметь вид

$$P_0(x) = a_0, \quad D_0(x) = 0,$$
$$D_k(x) = P_{k-1}(x) + xD_{k-1}(x), \quad P_k(x) = xP_{k-1}(x) + a_k, \quad k > 0.$$

Элементы последовательности не обязательно могут иметь числовой характер. Аналогичными однопроходными процедурами можно обрабатывать последовательность символов, т.е. текстовые файлы.

1.31. Введите с клавиатуры символ и определите сколько раз он встречается в текстовом файле.

1.32. Введите с клавиатуры строку символов и для каждого ее элемента определите сколько раз он встречается в текстовом файле.

Решение. Приведем решение задачи, когда функция `main()` считывает имя файла с экрана и вызывает вспомогательную функцию, обрабатывающую содержимое файла. По окончании работы функция `main()` распечатывает результат.

```
#include <stdio.h>
int    count (char * name, char *a, int *nmb);
int    main ()
{
   char a [256];          /* исходная строка */
   int  nmb[256];         /* количество вхождений */
   char filename [256];
   int i,n;
    printf("----Количество вхождений символов строки---- ");
    printf(" Введите имя файла ->");
    scanf("%s",filename);
    printf(" Введите строку символов ->");
      gets(a);
    if ((n=count(filename,a,nmb))==-1)
    {printf("Не удается открыть файл %s\n",filename); return -1; }
       printf("Символ   :  Количество вхождений  \n", a[i], nmb[i]);
     for(i=0;i<n;i++)
       printf("  %c     :  %d \n", a[i], nmb[i]);
   return 0;
}

int    count (char * name, char *a, int *nmb);
{
   FILE * fin;
   int i;
   int c;
   int buf[256];
```

```
      fin = fopen(name,"r");
       if (!fin)  return -1;
    for (i=0; i<256;i++) buf[i]=0;
    while ((c=fgetc(fin))!=EOF) buf[c]++;
    for (i=0; a[i]; i++) nmb[i]=buf[a[i]];
       return i;
}
```

1.33. Напишите функцию, получающую в качестве параметра имя текстового файла и подсчитывающую количество латинских букв в файле.

1.34. Напишите функцию, получающую в качестве параметра имя текстового файла и копирующую его содержимое в другой файл с заменой всех маленьких букв на большие.

1.35. Напишите функцию-фильтр, копирующую содержимое одного файла в другой файл, за исключением символов, содержащихся в заданной текстовой строке. Функция может иметь заголовок

```
void   copyfilter (FILE *fin, FILE *fout, char *badsym);
```

где `fin`, `fout` — указатели на входной и выходной файлы, `badsym` — строка символов, которые не надо пропускать на выход.

1.36. Напишите функцию-фильтр, которая при каждом обращении возвращает очередной допустимый символ из заданного тестового файла. При достижении конца файла функция возвращает значение `EOF`. Набор допустимых символов задается строкой — параметром функции. Функция может иметь заголовок

```
int   filter (FILE *fin, char *goodsym);
```

где `fin` — указатель на входной файл, `goodsym` — строка символов, которые нужно пропускать на выход.

Идеи реализации. Просмотр строки `goodsym` для каждого вновь прочитанного символа приведет к значительным затратам времени на обработку большого файла. Можно ввести в функцию массив `static int good[256]`, в который при обращении с ненулевым указателем `goodsym` записать единицы для допустимых и нули для остальных символов. Этот вызов можно рассматривать как инициализацию. Далее, обращаясь к функции с нулевым значением параметра `goodsym`, нужно проверять элементы массива `good` и выдавать ответ. Эта часть функции фактически сводится к выполнению операторов


```
   while ( (sym=fgetc(fin))!=EOF && !good[sym]);
   return sym;
```

1.2. Задачи на работу с массивами

В этом разделе собраны простейшие задачи обработки, анализа и преобразования данных, организованных в одномерный массив. Прежде чем формулировать условия задач, обсудим вопрос о том как формировать сам массив. Одно из возможных решений — заполнение массива значениями, вычисляемыми по заданной формуле или рекуррентному соотношению. В качестве примера приведем фрагмент программы, использующей для вычисления элементов массива рекуррентные соотношения чисел Фибоначчи: $x_0 = x_1 = 1$, $x_k = x_{k-1} + x_{k-2}$, $k > 1$. Будем считать, что эти числа представлены типом `long int`.

```
int  i;
long int *a;
a = (long*)malloc(n*sizeof(long int));
if (a)
{  a[0]=a[1]=1;
   for (i=2;i<n;i++)
      a[i] = a[i-1]+a[i-2];
}
```

Другим решением может быть заполнение массива значениями, читаемыми из файла. В этом случае главное — это каким либо образом определить требуемую длину массива. Можно заранее создать массив некоторой длины, а при чтении контролировать реально получающееся количество элементов. Соответствующий фрагмент программы может выглядеть, например, так (рассматриваем массив вещественных чисел и предполагаем, что файл уже открыт и определяется указателем `fin`).

```
#define  NMAX   1000
double   mas [NMAX];
int      n;
for (n=0; n<NMAX && fscanf(fin,"%lf",&mas[n])==1; n++);
```

По окончании цикла переменная `n` содержит фактическую длину массива. Наконец, другой часто применяемый прием состоит в том, что первое число, записанное в файле с данными есть длина массива. В этом

случае создание массива можно осуществить следующей последовательностью операторов.

```
double  *mas=0;
int      n,nmax=0;
if ( fscanf(fin,"%d",&nmax)==1 )
{  mas = (double*)malloc(nmax*sizeof(double);
   if (mas)
      for (n=0; n<nmax && fscanf(fin,"%lf",&mas[n])==1; n++);
}
```

Если после выполнения этих операторов окажется, что `nmax` или `mas` равны нулю, это означает, что не удалось прочитать из файла длину массива или выделить необходимую память и, следовательно, дальнейшее выполнение программы невозможно. Таким образом, в программе следует предусмотреть необходимую проверку значений `nmax` и `mas`, в результате которой либо продолжать программу, либо прерывать ее.

Решения задач, сформулированных ниже, следует оформить в виде функции, которая получает в качестве параметров имя массива и его длину (а также другие значения, если это необходимо по условию) и без использования дополнительных массивов выполняет необходимые действия. Функция `main` должна заполнить массив числами из файла, определить его фактическую длину и вызвать эту функцию.

1.37. Определите симметричны ли значения элементов массива?

Решение. Приведем полное решение для случая, когда массив читается из файла и его длина определяется первым прочитанным числом.

```
#include <stdio.h>
#include <stdlib.h>

int    symmetry (double *mas, int n);
int    main ()
{
   double  *mas=0;
   int      res,n,nmax=0;
   char     filename [256];
   FILE    *fin;
   printf("----Симметричность массива----\n Введите имя файла ->");
   scanf("%s",filename);
```



```
   fin = fopen (filename,"r");
   if (!fin) { printf("Не удается открыть файл %s\n",filename);
               return -1; }
   if ( fscanf(fin,"%d",&nmax)!=1 )
      {  printf("Ошибка ввода длины последовательности\n");
               return -1; }
   if ( nmax<=0 )
      {  printf("Ошибка ввода длины последовательности\n");
               return -1; }
  mas = (double*)malloc(nmax*sizeof(double));
   if ( !mas ) {
          printf("Не достаточно памяти для создания массива\n");
               return -1; }
   for (n=0; n<nmax && fscanf(fin,"%lf",&mas[n])==1; n++);
   res = symmetry(mas,n);
   printf("%s\n",(res) ? "Да":"Нет");
   return 0;
}
int    symmetry (double *mas, int n)
{
   int i;
   for (i=0;i<n/2;i++)
       if ( mas[i]!=mas[n-i-1] ) return 0;
   return 1;
}
```

1.38. Требуется переставить элементы массива в обратном порядке.

Решение.

```
int    invert (int *mas, int n)
{
   int i,tmp;
   for (i=0;i<n/2;i++)
    {
         tmp=mas[i];
         mas[i]=mas[n-i-1];
         mas[n-i-1]=tmp;
    }
   return 1;
}
```

1.39. Требуется циклически сдвинуть элементы массива на одну позицию вправо.

1.40. Требуется циклически сдвинуть элементы массива на k позиций вправо с затратой O(n) действий (n-длина массива).

Решение. Приведем решение с использованием функции из задачи 1.38.

```
int    move(int *mas, int n, int k)
{
   k%=n;
   invert (mas, n-k);
   invert (mas+n-k, k);
   invert (mas, n);
return 1;
}
```

Другой способ решения состоит в постановке элемента сразу на его новую позицию. При этом образуются цепочки элементов, которые требуется циклически сдвинуть. Количество таких цепочек равно наибольшему общему делителю чисел n и k.

1.41. Каждый элемент массива (кроме первого и последнего) заменить на полусумму соседних элементов.

1.42. Требуется сгруппировать положительные элементы массива в его начале, а отрицательные — в конце с сохранением их порядка.

1.43. Определите значение и индекс минимального элемента массива.

Решение. Поскольку здесь ответ составляют два числа, то примем решение получать индекс через возвращаемое значение функции, а значение минимума — через параметр-указатель. Приведем здесь только текст функции, обрабатывающей массив.

```
int   minimum (double *mas, int n, double *min)
{
   int i,m;
   for (*min=mas[0],m=0,i=1; i<n; i++)
       if ( *min<mas[i] ) { *min = mas[i]; m = i; }
   return m;
}
```


1.44. Введите с клавиатуры число x и определите индекс и значение элемента массива, ближайшего к числу x.

1.45. Введите с клавиатуры число x и определите к какому значению ближе всего x: к минимальному в массиве, максимальному или среднему арифметическому.

1.46. Введите с клавиатуры число x и определите количество элементов массива, расстояние от которых до x в два раза меньше, чем максимальное расстояние между x и элементами массива.

В следующих задачах предлагается преобразовать массив так, что он изменит свою длину. При этом предполагается, что в исходном массиве зарезервировано достаточно места для расширения массива. Функции, решающие эти задачи, должны возвращать новую длину преобразованного массива.

1.47. Введите с клавиатуры число x и удалите из массива все элементы, превосходящие x, а оставшиеся уплотните (т.е. сдвиньте к началу массива с сохранением их порядка).

1.48. Введите с клавиатуры число x и продублируйте каждый элемент массива, превосходящий x (т.е. вставьте рядом такой же элемент).

1.49. Введите с клавиатуры число x и удалите из массива каждый элемент, делящийся нацело на x, а оставшиеся уплотните к началу массива.

1.50. Требуется вставить между каждыми двумя положительными элементами исходного массива их среднее арифметическое.

1.51. Требуется заменить каждый отрицательный элемент исходного массива на три элемента, равных его абсолютной величине.

1.3. Поиск и сортировки

Процедуры поиска и сортировки традиционно относятся к классическим задачам программирования. Всюду в этом разделе для определенности мы будем рассматривать сортировку по возрастанию, т.е. упорядочивание элементов массива a[ ] так, чтобы для любого допустимого i было выполнено a[i]≤a[i+1]. При реализации программ поиска и сортировки можно следовать двум подходам: 1) частные программы,

ориентированные на конкретный тип данных массивов; 2) универсальные программы, предназначенные для сортировки любых массивов. В первом случае для сравнения элементов между собой используются стандартные операции <, <=, >, >=, ==. При втором подходе программа сортировки приобретает дополнительный параметр — указатель на функцию сравнения, которую можно определять отдельно для каждого требуемого типа данных.

1.52. Пусть элементы массива не убывают. Требуется бинарным поиском (методом деления пополам) определить принадлежит ли массиву заданный элемент x.

Решение. Построим функцию, возвращающую 1, если элемент x присутствует в массиве и 0 в противном случае. Приведем вариант решения для массива элементов конкретного типа `double`.

```
int   search (int *mas, int n, int x)
{
   int  left,right,mid;
   if ( x<mas[0] || x>mas[n-1] ) return 0;
   left = 0;  right = n-1;
   if(x==mas[0]) return 1;
   while ( right-left> 1 )
   {  mid = (left+right)/2;
      if ( x==mas[mid] ) return 1;
      if ( x>mas[mid] ) left = mid;
      else              right = mid;
   }
   return  (x==mas[right]) ? 1:0;
}
```

1.53. Измените решение предыдущей задачи так, чтобы функция `search` дополнительно определяла индекс найденного элемента в массиве либо позицию, в которую можно поставить искомый элемент, если он не присутствует в массиве.

1.54. Пусть элементы массива не убывают. Требуется вставить в этот массив новый элемент x с сохранением упорядоченности всего массива. Местоположение нового элемента определить последовательным поиском.

Решение. Пусть для определенности массив имеет тип `double`. Построенная функция будет возвращать новую длину массива.

```
int   insert (double *mas, int n, double x)
{
  int  i=0,j=0;
   for (i=0; i<n && x>mas[i]; i++);  /* ищем место для x */
   for (j=n; j>i; j--)               /* раздвигаем массив */
      mas[j]=mas[j-1];
   mas[i] = x;                       /* вставляем */
   return n+1;
}
```

1.55. Пусть элементы массива не убывают. Требуется вставить в этот массив новый элемент x с сохранением упорядоченности всего массива. Местоположение нового элемента определить бинарным поиском.

1.56. Даны два неубывающих массива. Реализуйте функцию, строящую третий неубывающий массив, который является объединением первых двух (т.е. содержит все элементы двух исходных массивов). Оформите решение в виде функции

```
int   merge (double *a, double *b, double *c, int na, int nb);
```

которая объединяет массив `a` длины `na` и массив `b` длины `nb` в массив `c` и возвращает длину полученного массива (т.е. `na+nb`).

1.57. Реализуйте функцию сортировки массива вещественных чисел по возрастанию с заголовком

```
void  sort (double *a, int n);
```

В качестве методов сортировки рассмотрите следующие алгоритмы:

1. *Метод перестановки максимального элемента*. Пусть дано некоторое число k, $(0 < k < n)$. Находим максимальный элемент среди чисел $a_0, \ldots, a_k$. Пусть этим максимумом является некоторый элемент a_j. Обмениваем значения элементов a_j и a_k. Указанную процедуру последовательно выполняем для $k = n-1, n-2, \ldots, 1$.

2. *"Пузырьковая" сортировка*. Пусть дано некоторое число k, $(0 < k < n)$. Сравниваем два соседних элемента a_{i-1} и a_i и, если оказывается, что $a_{i-1} > a_i$, то меняем их местами. Такое сравнение и возможную

перестановку последовательно выполняем для $i = 1,2,\ldots,k$. Назовем эту процедуру подъемом до k-й позиции. Последовательно выполняем подъем до k-й позиции для $k = n-1, n-2, \ldots, 1$.

Замечание. Если при очередном подъеме не будет выполнено ни одной перестановки, то массив уже является упорядоченным и процесс сортировки можно прекратить.

3. *Сортировка вставками с последовательным поиском*. Предположим, что первые k элементов массива (т.е. $a_0,\ldots,a_{k-1}$) уже упорядочены нужным образом. Тогда, выполнив упорядоченную вставку элемента a_k в этот массив (задача 1.54), мы получим упорядоченную совокупность из $k+1$ элемента. Последовательно выполняем эту процедуру для $k = 1,2,\ldots,n-1$.

4. *Сортировка вставками с бинарным поиском*. Алгоритм решения аналогичен предыдущему, но место для вставки нового элемента в упорядоченную часть массива отыскивается с помощью бинарного поиска (задача 1.55).

5. *Сортировка просеиванием*. Этот метод является модификацией пузырьковой сортировки и состоит из двух этапов — подъема и спуска. При подъеме последовательно сравниваются соседние элементы a_i и a_{i+1} ($i = 0,1,\ldots$) до тех пор, пока не будет сделана первая перестановка. Пусть эта перестановка затронула элементы a_k и a_{k+1}. Следующим этапом является спуск. Новый элемент a_k сравниваются с a_{k-1} и если $a_k < a_{k-1}$, то выполняется перестановка. Сравнение продолжается в нисходящем направлении (т.е. для a_{k-1} и a_{k-2}, a_{k-2} и a_{k-3} и т.д.) до тех пор, пока *выполняются* перестановки либо достигается начало массива. После этого возобновляется подъем с позиции $i = k+1$. Таким образом, сортировка состоит из сменяющих друг друга процессов подъема (до первой перестановки) и спуска (до первого отсутствия перестановки) до тех пор, пока при подъеме не будет затронут последний элемент массива a_{n-1} (при этом спуск также должен быть выполнен).

6. *Быстрая сортировка (quicksort)*. Пусть мы имеем неупорядоченный массив $a_0,\ldots,a_{n-1}$ и некоторое число m, которое удовлетворяет условию $\min\limits_i a_i \le m \le \max\limits_i a_i$. Перестроим массив так, чтобы при некотором $0 \le s \le n-1$ было выполнено $a_i \le m$ при $i \le s$ и $a_i \ge m$ при $i \ge s$. Последовательно сравнивая элементы массива a_j, $j = 0,1,\ldots$ с m найдем первое такое значение j, что $a_j \ge m$. Теперь последовательно сравнивая

элементы массива a_k, $k = n-1, n-2,\ldots$ (т.е. от конца к началу) с m мы либо найдем такое значение k, что $a_k \le m$, $k > j$, либо получим $k = j$. Если $j < k$, то обмениваем местами элементы a_j и a_k и повторяем описанную выше процедуру для элементов массива $a_{j+1},\ldots,a_{k-1}$. Если $k = j$, то мы получили искомое разбиение массива на две требуемых части при $s = k$. Теперь рекурсивно применим описанную выше процедуру для каждой из полученных частей массива (естественно, с соответствующим новым значением m). После завершения всех рекурсивных вызовов массив будет упорядочен.

Быстродействие алгоритма сильно зависит от удачного выбора числа m, поскольку именно это число определяет где массив будет разделен на две части. Если это разделение каждый раз происходит примерно посередине, то трудоемкость сортировки составляет $O(n\log_2 n)$ операций (n — количество элементов массива), однако, если разделение каждый раз "отщепляет" только один элемент, то трудоемкость составит $O(n^2)$ операций. Неплохие результаты получаются, когда в качестве m берется среднее арифметическое нескольких (двух – трех) элементов рассматриваемой части массива. Заметим также, что при рекурсии первым следует обрабатывать часть массива, имеющую меньшую длину так как это гарантирует, что глубина рекурсии не превысит $\log_2 n$.

7. *Пирамидальная (или турнирная) сортировка (heapsort)*. Пусть мы имеем массив a_i, $i = 0,\ldots,n-1$. Будем говорить, что i-й треугольник пирамиды выстроен, если элемент a_i массива удовлетворяет условию $a_i \ge a_{2i+1}$ и $a_i \ge a_{2i+2}$ (если одно или оба значения $2i+1$ и $2i+2$ выходят за границы массива, то соответствующее неравенство считается выполненным). Процедура выстраивания i-го треугольника состоит в проверке неравенств $a_i \ge a_{2i+1}$ и $a_i \ge a_{2i+2}$ и выполнении обмена элемента a_i с максимальным из a_{2i+1}, a_{2i+2} в том случае, когда указанные неравенства нарушены.

Шаг 1. Сборка пирамиды. Выстроим 0-й треугольник. Далее для каждого $k = 3,\ldots,n-1$ будем выстраивать s-й треугольник, где s последовательно вычисляется по рекуррентной формуле $s = [k-1]/2$ и далее $s = [s-1]/2$ до $s = 0$ ($[\,]$ — целая часть). Заметим, что если при некотором s выстраивание треугольника не приводит к перестановке, то обработку текущего значения k можно не продолжать. В результате все i-е, $i = 0,\ldots,n-1$, треугольники пирамиды будут выстроены. Заметим,

что при этом элемент a_0 окажется максимальным в массиве.

Шаг 2. Разборка пирамиды. Пусть $k = n$. Обменяем местами элементы a_0 и a_{k-1}. Таким образом, максимальный элемент занял свое место в будущем упорядоченном массиве. Будем теперь считать, что длина массива равна $k-1$, и выстроим 0-й треугольник. Если при этом произошла перестановка, то выстроим тот треугольник, вершина которого участвовала в обмене (т.е. каждый раз выстраиваем тот треугольник, куда переместился бывший элемент a_0), и т.д. до тех пор, пока при выстраивании треугольников происходят перестановки. Напомним, что элемент a_{n-1} уже занял свое место и в выстраиваниях не участвует. Далее последовательно выполняем шаг 2 для $k = n-1, n-2, \ldots, 1$. В результате на позиции $n-2, n-3, \ldots, 0$ последовательно поступают требуемые элементы.

Нетрудно видеть, что алгоритм имеет гарантированную трудоемкость $O(n \log_2 n)$ и не требует дополнительной памяти.

Алгоритмы слияния. Все алгоритмы, применяющие идею слияния учитывают, что два упорядоченных массива длины n можно объединить в один упорядоченный за n сравнений (см. задачу 1.56).

8. *Сортировка простым слиянием*. На первом шаге весь массив рассматривается как совокупность упорядоченных групп по одному элементу в каждой. Слиянием (объединением) соседних групп во вспомогательный массив мы получаем упорядоченные группы, каждая из которых содержит два элемента (кроме, может быть, последней группы, которой не нашлось парной). Далее упорядоченные группы укрупняются в исходный массив и т.д.

Данный алгоритм может быть реализован с использованием рекурсивной процедуры. Разбиваем исходный массив на две половины. Если каждая из частей является упорядоченной, то после их слияния мы получим отсортированный исходный массив. Иначе, рекурсивно применяем указанную процедуру к неупорядоченным частям. По завершении всех рекурсивных вызовов мы получим упорядоченный массив. В общем случае промежуточные проверки на упорядоченность можно опустить, считая, что упорядоченными будут только подпоследовательности, состоящие из одного элемента.

В дальнейшем предполагается, что исходный массив не помещается в оперативную память, и мы вынуждены считывать информацию по

частям из некоторого файла a.dat. При необходимости мы можем разбить исходные данные на несколько частей и записать их в файлы a1.dat, a2.dat, ..., ak.dat, либо, реально работая с одним файлом, программно поддерживать многоканальное чтение. При анализе алгоритмов такого рода особое внимание стоит уделять количеству обращений к файлу.

9. *$2k$-ленточное слияние.* Пусть необходимо упорядочить массив длины n. Будем считать, что мы можем поддерживать k различных каналов на чтение и столько же на запись. На первом шаге считаем, что входные последовательности состоят из упорядоченных серий единичной длины. Считываем по одной серии из каждого входного канала, сливаем их в одну упорядоченную серию длины k и записываем в первый выходной канал. Следующую упорядоченную серию длины k записываем во второй канал, и так далее, циклически меняя номера выходных каналов. Процедура повторяется до тех пор, пока не исчерпаются все входные последовательности. Далее входные и выходные каналы меняются местами и алгоритм повторяется. За каждый такой шаг мы получаем, что длина упорядоченных последовательностей увеличивается в k раз. Если общее число элементов равно $n = k^p$, то за $\log_k n$ шагов мы получим упорядоченный массив. Число пересылок при этом $O(n \log_k n)$. При произвольном n входные последовательности будут исчерпываться с различной скоростью и на каждом шаге мы будем реально сливать $k_i \le k$ последовательностей различной длины.

10. *Естественное слияние.* В случае прямого слияния мы не получаем выигрыша, если исходный массив был частично упорядочен. Естественным обобщением является алгоритм, когда объединяем максимальные упорядоченные серии, а не последовательности фиксированной длины.

11. *Многофазная сортировка.* Откажемся от идеи, что у нас есть k входных и столько же выходных последовательностей одновременно. Будем считать, что в каждый момент имеется только один выходной канал и работать так, что если какой-то из $2k-1$ входных массивов опустел, то он становится выходным. Данный алгоритм полезен, когда входные последовательности имеют существенно разные длины. В лучшем случае число пересылок порядка $O(n \log_{2k-1} n)$.

1.4. Разбор чисел и битовые операции

При решении задач этого раздела можно анализировать остатки от деления целых чисел, либо использовать битовые операции. Напомним, что остаток от деления находится с помощью операции `%`, а битовыми операциями в языке C являются `<<`, `>>`, `~`, `^`, `|`, `&`.

1.58. Получить массив целых чисел, представляющих собой двоичное разложение неотрицательного целого числа.

Решение.

```
void binary (unsigned int n, int *bit)
{
   unsigned int mask=1;
   int i;
     for(i=0;mask;i++)
     {  if(n&mask)bit[i]=1;
           else
              bit[i]=0;
         mask<<=1;
      }
  return;
}
```

1.59. Для целого числа получить массив целых чисел, представляющих собой его запись в k-ичной системе исчисления.

1.60. Дано целое число. Получить целое число, записанное теми же цифрами в обратном порядке.

1.61. Вычислить представление числа m/n в виде десятичной дроби (т.е. вывести цифры, составляющие начало и период этой дроби).

Идеи реализации. Можно запасти массивы необходимых размеров и реализовать алгоритм деления “столбиком”, запоминая получающиеся остатки. Период появится тогда, когда мы получим два одинаковых остатка. Можно также проанализировать делители чисел m и n, что даст возможность заранее вычислить длины начальной части и периода дроби. В этом случае массивы становятся не нужны, так как получаемые цифры частного можно сразу выводить на печать.

1.62. Требуется реализовать функцию возведения числа x в заданную целую степень N за не более чем $2\log_2 N$ умножений.

Идеи реализации. Нужно вычислять значения x^1, x^2, x^4, и т.д. и перемножить те из них, которые соответствуют единицам в двоичном представлении числа N.

1.63. Используя алгоритм Евклида, составьте функцию для нахождения наибольшего общего делителя двух целых чисел.

1.64. Требуется вывести в файл все подмножества множества $\{0,\ldots,N-1\}$.

Идеи реализации. Если ограничиться значениями $N < 32$, то можно воспользоваться следующим приемом. Если m — целое число, то набор единиц в двоичном представлении числа m соответствует некоторому подмножеству множества $\{0,\ldots,N-1\}$. Перебрав все числа $m = 0,\ldots,2^N-1$ (например, с помощью простейшего цикла), мы тем самым переберем все возможные подмножества.

1.65. Требуется вывести в файл все k-элементные подмножества множества $\{0,\ldots,N-1\}$.

Идеи реализации. Можно воспользоваться указанием к предыдущей задаче и отбирать только те значения m, которые содержат ровно k единиц в своем двоичном представлении. Другой способ — создать массив из N элементов-признаков, соответствующих элементам исходного подмножества и указывающих использован или нет данный элемент в подмножестве. Формирование подмножества выполняется рекурсивной процедурой, которая последовательно помечает один свободный элемент массива как использованный в подмножестве и обращается сама к себе. Глубину рекурсивных вызовов нужно ограничить значением k.

1.66. Требуется вычислить первые N простых чисел.

Идеи реализации. Простейший способ решения — последовательно выбирать “кандидата” на очередное простое число и проверять его простоту делением. Однако программу нужно сделать как можно более быстродействующей. Советы: не проверять на простоту числа, кратные 2 и 3; сохранять уже подсчитанные простые числа в массиве; проверять делимость очередного числа p только на простые числа, не превосходящие $\sqrt{p}$; предусмотреть управление формой вывода результата (вывод на экран каждого подсчитанного числа чрезвычайно замедляет работу программы). Программа должна вычислять миллионное простое число за разумное время (несколько минут).

1.67. Требуется вычислить разложение натурального числа на простые множители.

1.68. Составьте программу, которая для введенного натурального числа выводит его значение "словами". Например, для числа 427 выводит строку "четыреста двадцать семь".

1.69. Пусть имеется некоторое количество однозначных чисел. Построим арифметическое выражение, используя все эти числа и объединяя их операциями сложения, умножения, вычитания, деления, возведения в степень. Составьте программу, которая по заданному массиву однозначных чисел вычисляет значения всевозможных таких выражений и печатает их вместе с видом соответствующего выражения. (Порядок чисел в выражении всегда остается одним и тем же.)

1.70. Решите предыдущую задачу со следующими усложнениями:

1) можно использовать скобки для группировки операций;

2) можно изменять порядок следования чисел в выражении;

3) можно "склеивать" несколько подряд идущих чисел для образования многозначного числа.

1.5. Задачи на обработку множества точек

В следующих задачах предполагается, что в файле записано несколько пар чисел, которые можно рассматривать как координаты множества точек на плоскости или как координаты множества концов отрезков на прямой. Требуется составить программы, которые читают исходные данные и отвечают на поставленные вопросы или выдают значения параметров требуемых геометрических объектов. Следует стремиться к построению минимальных по сложности алгоритмов (сложность оценивается порядком числа операций в зависимости от числа точек или отрезков).

1.71. Множество точек определяет ломаную. Имеет ли она самопересечения?

1.72. Множество точек определяет многоугольник. Является ли он выпуклым?

1.73. Множество точек определяет многоугольник. Определите его диаметр (т.е. наибольшую диагональ). Решение должно иметь сложность $O(N)$, где N — число вершин многоугольника.

1.74. Дано множество отрезков. Покрывает ли их объединение заданный отрезок $[a,b]$?

1.75. Дано множество отрезков. Найти точку, которая принадлежит наибольшему количеству отрезков, определить это количество.

1.76. Дано множество точек. Найти центр и радиус минимального круга, который содержит все эти точки.

1.77. Дано множество отрезков. Выбрать из него и вывести связное подмножество отрезков, объединение которых дает отрезок наибольшей длины.

1.78. Дано множество точек. Из этих точек как из центров начинают одновременно и с одинаковой скоростью "расти" круги (т.е. точки являются центрами, а радиусы увеличиваются с одинаковой скоростью). Если два круга сталкиваются, то их рост прекращается. Определить радиусы всех получившихся кругов после того как их рост прекратится.

1.79. Окружность разделена на m равных дуг и внутри каждой дуги на окружности отмечено некоторое количество точек (возможно, нулевое). Требуется передвинуть эти точки на окружности так, чтобы были выполнены следующие условия:

1. каждая точка должна по возможности наименее удаляться от своей исходной позиции;

2. ни одна точка не должна покидать свою первоначальную дугу;

3. точки должны сохранять исходный порядок, а угловое расстояние между любыми двумя соседними точками не должно быть меньше заданной величины δ (для разрешимости данной задачи можно считать, что $\delta < 2\pi/mn$, где n — наибольшее количество точек, приходящихся на одну дугу).

1.80. Обобщением предыдущей задачи является задача о размещении названий на географической карте. Пусть заданы координаты нескольких точек на плоскости (населенные пункты) и с каждой точкой связан некоторый прямоугольник, задаваемый своей длиной и шириной (он ограничивает текст наименования населенного пункта). Требуется разместить каждое наименование рядом с соответствующей ему точкой так, чтобы наименования (прямоугольники) не накладывались друг на друга.

1.81. Множество точек определяет многоугольник (возможно невыпуклый, но без самопересечений). Определить находится ли заданная точка внутри или вне этого многоугольника.

1.82. Пусть плоские выпуклые многоугольники представляются следующим типом данных:

```
typedef struct {int n; double *x,*y; } Polygon;
```

где n — число вершин многоугольника, x,y — указатели на массивы координат вершин. Реализуйте функции, обеспечивающие работу с многоугольниками, как с объектами:

```
double    Perimeter (Polygon p);            /* периметр */
double    Area (Polygon p);                 /* площадь */
Polygon * Clip (Polygon a, Polygon b);      /* пересечение */
int       Equal (Polygon a, Polygon b);     /* равны ? */
int       Convex (Polygon a, Polygon b);    /* выпуклый ? */
```

Пересечение непересекающихся многоугольников пусто. Функция `Clip` создает новый многоугольник, т.е. выделяет для него память и заполняет требуемыми значениями.

1.83. Пусть в трехмерном пространстве дана плоскость с указанным плоским базисом (задан вектор нормали и трехмерные координаты начала координат и базисных векторов на плоскости). Реализуйте функцию, которая вычисляет плоские координаты проекции точки пространства на эту плоскость.

1.6. Алгоритмы работы с текстовыми строками

Простейшие функции работы с текстовыми строками входят в библиотеки всех современных компиляторов языка С (прототипы этих функций описаны в файле `string.h`). Однако самостоятельная реализация этих функций представляет собой очень полезное упражнение, особенно, если стремиться решить поставленную задачу наиболее эффективно. Напомним, что строкой в языке С называется массив элементов типа `char`, последний элемент которого имеет нулевое значение.

1.84. Реализуйте функцию, копирующую одну строку в другую.

Решение. Одно из возможных решений может выглядеть так.


```
char * strcpy (char *dst, char *src)
{
   char *s = dst;
   while ( *dst++ = *src++ );
   return s;
}
```

1.85. Реализуйте функции, выполняющие стандартные операции с текстовыми строками: strncpy, strcmp, strstr, strcat, strspn, strcspn.

1.86. Назовем словом группу символов, не содержащую внутри себя символов из заданного набора символов-разделителей. Примерами разделителей для слов обычного литературного текста могут служить символы “`.,;:!?  ()[]-+*/"`” и т.п. Реализуйте функцию, которая при каждом обращении к ней выделяет из указанного текстового файла очередное слово. Функция должна иметь прототип

```
int  GetWord (FILE *f, char *word, char *delim);
```

где `f` — указатель на входной файл, `word` — указатель на буфер для очередного слова, `delim` — указатель на сроку с символами-разделителями. Возвращаемое значение — 0, если слово прочитано, и -1 в случае конца файла или какого-либо другого отказа.

1.87. В файле, содержащем литературный текст, возможны переносы слов между строками. Модифицируйте функцию `GetWord` из предыдущей задачи так, чтобы она обнаруживала и правильно выводила перенесенные слова.

Идеи реализации. Можно считать, что слово разбито переносом на две части, если за допустимыми (буквенными) символами идет символ “-”, за которым следуют один или несколько символов перевода строки, и далее, возможно, несколько пробелов до следующего допустимого символа.

1.88. С использованием функции `GetWord` из предыдущей задачи выполните следующую обработку текстового файла:

1) подсчитайте количество слов в исходном файле;

2) подсчитайте максимальную, минимальную и среднюю длину слов;

3) подсчитайте среднее количество слов в одном предложении (предложение — это последовательность слов, оканчивающаяся одним из символов “.!?”);

4) выберите и напечатайте все слова, начинающиеся с заглавной буквы.

1.89. По заданному текстовому файлу сформируйте файл-словарь, содержащий все слова из исходного файла, записанные в алфавитном порядке по одному в строке.

2. Простейшие вычислительные алгоритмы

Реализация вычислительных алгоритмов представляет собой важный, но достаточно специфический раздел программирования. Главная особенность здесь состоит в том, что подобные алгоритмы часто являются итерационными (т.е. искомый результат вычисляется как приближение к пределу некоторой сходящейся последовательности, удовлетворяющее заданному критерию точности). Основную проблему при этом составляет теоретический и практический контроль за величиной вычислительной погрешности. Изложение принципов построения вычислительных алгоритмов, сохраняющих вычислительную погрешность на требуемом уровне, составляет предмет курса численных методов и не входит в задачи данного пособия. Поэтому в этом разделе предлагается лишь реализовать некоторые вычислительные алгоритмы по их описанию и проверить их работоспособность на простейших тестовых задачах.

2.1. Для представления вещественных чисел в памяти компьютера выделяется конечное число двоичных разрядов. Следовательно, только конечное множество вещественных чисел представляется точно, остальные числа округляются до ближайшего представимого числа. Машинной точностью называется такое максимальное вещественное число $\varepsilon > 0$, что 1 и $1+\varepsilon$ неразличимы на множестве представимых вещественных чисел, т.е. в машинной арифметике выполнено равенство $1+\varepsilon = 1$. Напишите программу, определяющую машинную точность представления вещественных чисел типов `float` и `double`.

Идеи реализации. Получить какое-нибудь приближение к машинной точности можно, например, при помощи цикла

```
for (eps=1; 1+eps>1; eps/=2);
```

После этого можно уточнить значение `eps` с помощью алгоритма деления пополам (см. задачу 2.20).

2.2. Составьте функцию для вычисления корней квадратного уравнения $ax^2+bx+c=0$ с заголовком

```
int eq_quadr(double a, double b, double c, double *x1, double *x2);
```

Здесь `a,b,c` — коэффициенты уравнения, `x1,x2` — указатели на переменные для размещения результата. Возвращаемые значения: 1 — уравнение имеет два действительных корня, 0 — один корень, -1 — корни комплексные, -2 — корней нет. В случае комплексных корней `*x1` содержит действительную часть, а `*x2` — мнимую. Функция должна корректно обрабатывать любые значения коэффициентов a,b,c. Например, случай $a=b=0$, $c\neq 0$ соответствует случаю отсутствия корней.

Идеи реализации. При использовании традиционных формул нахождения корней квадратного уравнения может возникнуть существенная потеря точности. Следует преобразовать формулы так, чтобы не вычислялась разность двух близких больших чисел (это можно сделать, умножив числитель и знаменатель одной из формул на множитель, сопряженный к числителю). Другая опасность — получить переполнение в результатах промежуточных операций. Для устранения этой опасности следует предварительно определить сравнительные порядки коэффициентов и нормировать их (сделать замену $a_1 = ka$, $b_1 = kb$, $c_1 = kc$) или масштабировать (сделать замену $y = kx$) с подходящим коэффициентом, чтобы по возможности сделать коэффициенты уравнения близкими по порядку величинами.

Протестируйте работу функции для следующих коэффициентов:

$a=1, b=-10^5, c=1$; $x_1 = 99999.999990, x_2 = 0.000010$;
$a=10^{300}, b=-3\cdot 10^{300}, c=2\cdot 10^{300}$; $x_1=1, x_2=2$;
$a=1, b=-4, c=3.9999999$; $x_1 = 1.99968377, x_2 = 2.00003162$.

2.1. Суммирование рядов и вычисление элементарных функций

Суммирование ряда — процесс весьма чувствительный к накоплению вычислительной погрешности. Чем медленнее сходится числовой ряд, тем более вероятно получить ответ, не имеющий ничего общего с истинной суммой. Следовательно, для вычисления суммы, возможно, придется преобразовать ряд к другому виду, более подходящему для машинного суммирования.

2.3. Напишите программу для вычисления частичной суммы $\sum_1^n a_k$ заданного числового ряда в прямом и обратном направлении (т.е. для $k = 1,\ldots,n$ и $k = n,\ldots,1$). Для каждого из рядов

$$\sum_k \frac{1}{\sqrt{k}}, \quad \sum_k \frac{\ln k}{k}, \quad \sum_k \frac{1}{k}, \quad \sum_k \frac{1}{k^2}, \quad \sum_k \frac{1}{k!}$$

экспериментально подберите такое n, которое дает максимальное расхождение результатов суммирования в прямом и обратном направлении. Как изменится результат, если слагаемые умножить еще на $(-1)^k$?

2.4. Напишите программу, вычисляющую следующую сумму с точностью 10^{-8}

$$S(x) = \sum_{k=1}^{\infty} \frac{1}{k(k+x)}, \quad x \geq 0$$

Идеи реализации. Определим сколько слагаемых данного ряда обеспечивает требуемую точность. Из известной оценки для остаточной суммы ряда

$$\sum_{k=n}^{\infty} \frac{1}{k(k+x)} \leq \frac{1}{(n+1)(n+1+x)} + \int_{n+1}^{\infty} \frac{1}{t(t+x)} dt$$

и из условия точности следует, что потребуется взять порядка 10^8 слагаемых. Для ускорения сходимости исходного ряда представим его в виде суммы $S(x) = (S(x) - S(1)) + S(1) = \bar{S}(x) + S(1)$, где

$$\bar{S}(x) = (1-x) \sum_{k=1}^{\infty} \frac{1}{k(k+x)(k+1)}, \quad S(1) = \sum_{k=1}^{\infty} \left(\frac{1}{k} - \frac{1}{k+1} \right) = 1$$

Легко проверить, что скорость сходимости нового ряда $\bar{S}(x)$ существенно выше и для достижения требуемой точности достаточно взять порядка 10^4 слагаемых. Попробуйте еще уменьшить полученное число.

2.5. Напишите программу для вычисления суммы

$$S(x) = \sum_1^{\infty} \frac{1}{k^2+1}$$

с точностью 10^{-8}, предварительно оценив время работы программы.

Идеи реализации. Используйте алгоритм решения предыдущей задачи и следующие соотношения

$$\sum_1^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}, \qquad \sum_1^{\infty} \frac{1}{k^4} = \frac{\pi^4}{90}$$

2.6. Напишите программу, вычисляющую для сходящихся рядов задачи сумму $\sum_1^{\infty} a_k$ с точностью 10^{-8} Обоснуйте полученные результаты.

2.7. Напишите программу, вычисляющую произведение

$$S = \prod_{k=2}^{\infty} \left(1 - \frac{1}{k^2} \right)$$

Сравните полученный результат с точным ответом $S = 0.5$.

2.8. Напишите программу, вычисляющую произведение

$$S = \prod_{k=1}^{\infty} \frac{4k^2}{(2k-1)(2k+1)}$$

Сравните полученный результат с точным ответом $S = 0.5\pi$.

2.9. Реализуйте функции

```
double  sin_e (double x, double eps);
double  cos_e (double x, double eps);
double  exp_e (double x, double eps);
double  ln_e (double x, double eps);
```

которые вычисляют значения указанных элементарных функций в точке `x` с точностью `eps` суммированием рядов Тейлора. Критерием достижения точности считается $|a_k| < \varepsilon$, где a_k — последний прибавленный к сумме член ряда Тейлора.

Идеи реализации. Хотя ряды Тейлора для функций sin, cos, exp сходятся для любого x, их непосредственное суммирование для больших x приводит к неверным результатом из-за накопления вычислительной погрешности. Опишем приемы снижения погрешности суммирования.

1. Для функций $\sin x$ и $\cos x$ используем периодичность и формулы приведения и сведем требуемое значение x к значению $x' \in [0, \pi/2]$.

(При совместной реализации этих двух функций формулы приведения позволяют уменьшить этот отрезок до $[0,\pi/4]$). Вычисление очередных слагаемых рядов Тейлора следует проводить по рекуррентным соотношениям

$$a_1 = x,\ a_{k+1} = -a_k \tfrac{x^2}{2k(2k+1)} \text{ для } \sin x,$$
$$a_1 = 1,\ a_{k+1} = -a_k \tfrac{x^2}{2k(2k-1)} \text{ для } \cos x.$$

2. Для функции e^x представим аргумент $x>0$ в виде суммы целой и дробной части $x=[x]+\{x\}$ и будем вычислять $e^x = e^{[x]}\cdot e^{\{x\}}$. Для вычисления $e^{[x]}$ запасем константу $2.718281828459045235 \approx e$ и применим алгоритм быстрого возведения в целую степень (задача 1.62). Для $e^{\{x\}}$ можно суммировать ряд Тейлора (с учетом рекуррентного соотношения для слагаемых: $a_{k+1} = a_k \frac{\{x\}}{k}$). При $x<0$ используем равенство $e^{-x} = 1/e^x$.

3. Для натурального логарифма ряд Тейлора имеет вид

$$\ln(1+y) = y - \frac{y^2}{2} + \frac{y^3}{3} - \frac{y^4}{4}\ldots$$

и сходится для $|y|<1$. Таким образом, следует выразить $\ln x$ через $\ln(1+y)$ с возможно ме́ньшим y. Для этого запасем константу $1.6487212707001282 \approx \sqrt{e}$ и подберем число n так, чтобы представить $x>1$ в виде $x = (\sqrt{e})^n\cdot(1+y)$, $|y|<1$. Тогда искомое значение вычисляется как $\ln x = n/2 + \ln(1+y)$. Логарифм в правой части этого равенства вычисляется по ряду Тейлора. Для $0<x<1$ используем равенство $\ln 1/x = -\ln x$.

2.10. Для функции $\operatorname{tg} x$ имеет место представление в виде цепной дроби

$$\operatorname{tg} x = \cfrac{1}{\frac{1}{x} - \cfrac{1}{\frac{3}{x} - \cfrac{1}{\frac{5}{x} - \cfrac{1}{\frac{7}{x} - \frac{1}{\ldots}}}}}$$

Реализуйте функцию, вычисляющую $\operatorname{tg} x$ по этому представлению, и проверьте насколько быстро сходится процесс вычислений (т.е. сколько членов дроби надо взять для получения результата с заданной точностью при различных значениях x).

2.2. Численное интегрирование

Для приближенного вычисления определенных интегралов обычно используется следующий прием. Область интегрирования разбивается на несколько отрезков и на каждом отрезке значение интеграла заменяется на сумму значений функции в некоторых точках отрезка с некоторыми весовыми коэффициентами. Таким образом, мы получаем соответствие

$$I(a,b,f) = \int_a^b f(x)\,dx \sim \sum_{i=0}^{n} c_i f(x_i),$$

где точки x_i и коэффициенты c_i определяются выбранной формулой численного интегрирования на каждом из частичных отрезков. Приведем несколько общеупотребительных формул численного интегрирования для частичного отрезка $[x_k, x_{k+1}]$ длины $h = x_{k+1} - x_k$:

формула левых прямоугольников
$I(x_k, x_{k+1}, f) \approx h f(x_k)$;
формула центральных прямоугольников
$I(x_k, x_{k+1}, f) \approx h f(\frac{x_k+x_{k+1}}{2})$;
формула трапеций
$I(x_k, x_{k+1}, f) \approx h(\frac{1}{2} f(x_k) + \frac{1}{2} f(x_{k+1}))$;
формула Симпсона
$I(x_k, x_{k+1}, f) \approx h(\frac{1}{6} f(x_k) + \frac{2}{3} f(\frac{x_k+x_{k+1}}{2}) + \frac{1}{6} f(x_{k+1}))$;
формула Гаусса с двумя узлами
$I(x_k, x_{k+1}, f) \approx h(\frac{1}{2} f(x_-) + \frac{1}{2} f(x_+))$, где $x_\pm = \frac{x_k+x_{k+1}}{2} \pm \frac{h}{2\sqrt{3}}$;

2.11. Реализуйте каждый из описанных выше методов интегрирования в виде функции

```
double  Integral (double a, double b, double (*f)(double), int n);
```

где `f` — указатель на подинтегральную функцию, `n` — число разбиений отрезка интегрирования $[a,b]$ на равные частичные отрезки. Возьмите несколько функций, интегрируемых в элементарных функциях, и сравните приближенные значения интегралов, полученные при разных n, с

точными значениями. Например

$$\int_0^{100\pi} \cos 1000x\,dx = 0, \quad \int_0^{100} \exp^{-1000x} dx = 10^{-3}, \quad \int_{-1}^{1} \frac{dx}{\sqrt{1-x^2}} = \pi$$

2.12. Реализуйте функцию

```
double  Integral_e (double a, double b,
                    double (*f)(double), double eps);
```

для вычисления значения интеграла с заданной точностью методом автоматического выбора шага интегрирования ε по какой-либо приближенной формуле.

Идеи реализации. Обозначим через $I_h(x,f)$ приближенное значение интеграла от функции $f(x)$, вычисленное по данной приближенной формуле на частичном отрезке $[x,x+h]$. Пусть мы имеем некоторое значение шага h и уже вычисленное значение интеграла $I(x)$ на отрезке $[a,x]$, $a \le x < b$. Вычислим $I_1 = I_h(x,f)$, $I_2 = I_{h/2}(x,f) + I_{h/2}(x+h/2,f)$. Если $|I_1 - I_2| \le \varepsilon h$, то считаем, что требуемая точность на шаге достигнута и полагаем $I(x+h) = I(x) + I_2$. Если $|I_1 - I_2| > \varepsilon h$, то уменьшим шаг h в два раза ($h = h/2$) и повторим вычисление I_1 и I_2, начиная с точки x (при необходимости будем уменьшать шаг и далее, пока не добьемся выполнения неравенства $|I_1 - I_2| \le \varepsilon h$). Следующий шаг от точки $x+h$ будем выполнять с полученным значением h. Если же мы получим неравенство $|I_1 - I_2| \le \delta h$, где $\delta \ll \varepsilon$, то выбранный шаг h обеспечивает "слишком высокую" точность и с целью сокращения вычислительных затрат для вычисления интеграла по следующему частичному отрезку этот шаг можно увеличить в два раза ($h = 2h$). На практике величину δ нужно выбирать в пределах от 0.1ε до 0.01ε.

2.3. Работа с матрицами, решение систем линейных уравнений

Работа с матрицами имеет свою специфику в каждом алгоритмическом языке. В первую очередь, эта специфика связана с необходимостью передавать матрицу в качестве параметра в различные процедуры и функции. В языке С обычно используются два подхода.

1. Элементы матрицы $a_{i,j}$, $i = 0,\ldots,n-1$, $j = 0,\ldots,m-1$ "укладываются" по строкам в одномерный массив длины nm. Если обозначить

элементы этого массива через b_k, то очевидно, имеет место соответствие $k = mi + j$ и $i = [k/m]$, $j = k \bmod m$. Параметром, который определяет местоположение элементов матрицы, здесь является указатель на начало массива b.

2. Для каждой строки матрицы выделим отдельный массив длины m, а указатели на начала этих массивов-строк соберем в дополнительном массиве `a` длины n. Таким образом, `a[i]` является указателем на i-ю строку матрицы a, а `a[i][j]` — элементом $a_{i,j}$. Местоположение элементов матрицы определяет указатель на начало массива `a`. Отметим, что с точки зрения эффективности вычислений этот метод несколько хуже предыдущего, поскольку требует двух обращений к памяти для извлечения элемента матрицы, а также использует дополнительный массив для хранения указателей на строки.

2.13. Постройте функции заполнения квадратной матрицы значениями $a_{i,j} = \dfrac{1}{1+i+j}$, $i,j = 0,\ldots,n-1$ (это так называемая матрица Гильберта) на основе каждого из двух описанных подходов.

Решение.

```
void  Hilbert_1 (double *a, int n)
{
#define  a_(i,j)  a[(i)*n+(j)]
   int  i,j;
   for (i=0;i<n;i++)
   for (j=0;j<n;j++)
      a_(i,j) = 1.0/(double)(i+j+1);
#undef   a_
}
void  Hilbert_2 (double **a, int n)
{
   int  i,j;
   for (i=0;i<n;i++)
   for (j=0;j<n;j++)
      a[i][j] = 1.0/(double)(i+j+1);
}
```

2.14. Реализуйте функции для выполнения операций суммирования и умножения двух прямоугольных матриц, умножения матрицы на число, умножения матрицы на вектор.

2.15. Реализуйте функцию выделения памяти для матрицы размера $n \times m$ и функцию освобождения этой памяти при хранении матрицы по способу 2. Функции должны иметь прототипы

```
double ** CreateMatrix (int n, int m);
void      FreeMatrix (double **a, int n);
```

Функция `CreateMatrix` возвращает указатель на массив указателей на строки матрицы, либо NULL в случае отказа в выделении памяти. Заметим, что отказ может возникнуть уже после выделения памяти для нескольких строк матрицы. Функция `CreateMatrix` должна корректно обработать эту ситуацию и освободить уже захваченную для строк память.

Методом Гаусса обычно называется процедура приведения матрицы к треугольному виду путем выполнения линейных комбинаций ее строк. Подробное изложение различных вариантов метода Гаусса можно найти в [3]. Здесь мы схематически изложим идею одного из них — метода с выбором главного элемента по столбцу.

Пусть $\{a_{i,j}\}$ — исходная матрица и $\mathbf{a}_i = \{a_{i,0}, a_{i,1}, \ldots, a_{i,n-1}\}$ обозначает i-ю строку матрицы. Для $k = 0, 1, \ldots, n-2$ последовательно выполняются следующие действия.

1. Определяется такое s, что $|a_{s,k}| = \max\limits_{k \le i \le n-1} |a_{i,k}|$.

2. Строки $\mathbf{a}_k$ и $\mathbf{a}_s$ обмениваются местами.

3. Для каждого $i = k+1, \ldots, n-1$ выполняется модификация i-й строки: $\mathbf{a}_i = \mathbf{a}_i - \frac{a_{i,k}}{a_{k,k}} \mathbf{a}_k$. В результате элементы $a_{i,k}$ становятся равными нулю.

Естественно, что при выполнении п.3. не нужно суммировать нулевые элементы матрицы (т.е. $a_{i,j}$ при $j < k$).

2.16. Реализуйте функцию вычисления ранга прямоугольной вещественной матрицы. Параметрами функции должны быть: матрица, ее размерности. Возвращаемое значение — искомый ранг. Проведите тестирование функции для матриц

$$A_1 = \begin{pmatrix} 5 & 1 & 1 & 2 \\ 5 & 1 & 3 & 4 \\ 10 & 2 & 4 & 6 \\ 15 & 3 & 5 & 8 \end{pmatrix} \qquad A_2 = \begin{pmatrix} 1 & 0.99 & 0.999 \\ 1 & 1 & 0.99 \\ 1 & 1 & 1 \end{pmatrix}$$

$\mathrm{rank}A_1 = 3 \qquad \mathrm{rank}A_2 = 3$

Идеи реализации. Применяя метод Гаусса мы получим нули ниже главной диагонали $a_{i,i}$ матрицы. При отсутствии ошибок округления ранг матрицы равен количеству ненулевых элементов на этой диагонали. Однако при практической реализации за счет вычислительной погрешности нулевые элементы диагонали могут не оказаться таковыми, а иметь некоторое малое значение. Следует ввести порог, ниже которого считать элементы матрицы нулями. С другой стороны, “почти вырожденные” матрицы могут иметь точные ненулевые диагональные элементы ниже установленного порога, что даст для них неверный результат. Аккуратный выбор порога представляет собой непростую задачу. Для указанных матриц можно попробовать взять порог от 10^{-12} до 10^{-15}.

2.17. Реализуйте функцию вычисления определителя квадратной вещественной матрицы. Параметрами функции должны быть матрица и ее размерность. Возвращаемое значение — искомый определитель. Протестируйте работу этой функции на следующих матрицах:

$$A = \begin{pmatrix} 5 & -4 & 1 & & & & 0 \\ -4 & 6 & -4 & 1 & & & \\ 1 & -4 & 6 & -4 & 1 & & \\ & . & . & . & . & . & \\ & & 1 & -4 & 6 & -4 & 1 \\ & & & 1 & -4 & 6 & -4 \\ 0 & & & & 1 & -4 & 5 \end{pmatrix} \quad \begin{matrix} \text{порядка } n \\ \det A = (n+1)^2, \end{matrix}$$

$$A = \begin{pmatrix} x+y & x & x & x & x \\ x & x+y & x & x & x \\ x & x & x+y & x & x \\ . & . & . & . & . \\ x & x & x & x+y & x \\ x & x & x & x & x+y \end{pmatrix} \quad \begin{matrix} \text{порядка } n \\ \det A = y^{n-1}(y+nx), \end{matrix}$$

Идеи реализации.

Не стоит решать данную задачу, используя явную формулу для подсчета определителя. Даже для матриц 20×20 потребуется слишком большое время, но даже если дождаться ответа, то полученное число вряд ли будет близким к правильному из-за потери точности.

Приведем исходную матрицу к верхнему треугольному виду, применяя алгоритм метода Гаусса. Перестановка строк матрицы может изменить знак определителя, умножение строки на число умножает определитель на то же число. При приведении матрицы к треугольному виду следует отслеживать множитель изменения определителя и знак, появившиеся в результате каждой перестановки и линейной комбинации. Для полученной треугольной матрицы определитель вычисляется перемножением диагональных элементов, а потом корректируется в соответствии с накопленным множителем.

2.18. Реализуйте функцию решения системы линейных уравнений методом Гаусса. Параметрами функции должны быть: матрица, ее размерность, вектор правой части системы, вектор решения. Возвращаемое значение: 0 — система решена, -1 — матрица системы вырождена. Протестируйте работу функции на системе с матрицей Гильберта.

Идеи реализации. С компонентами вектора правой части системы проводятся те же преобразования, которые выполняются при приведении матрицы к треугольному виду. После получения эквивалентной системы с треугольной матрицей решения вычисляется “снизу-вверх” обратной подстановкой. Для контроля вырожденности матрицы следует использовать пороговое значение для нулевых элементов (см. замечание к задаче 2.16).

2.19. Реализуйте функцию вычисления обратной матрицы для данной квадратной вещественной матрицы. Параметрами функции должны быть: исходная матрица, обратная матрица, их размерность. Возвращаемое значение: 0 — обратная матрица построена, -1 — исходная матрица вырождена.

Идеи реализации. Перед выполнением приведения к треугольному виду в области, отведенной для обратной матрицы, размещается единичная матрица. Далее выполняется приведение к треугольному виду. Затем аналогичными преобразованиями треугольная матрица приводится к диагональному виду и делением на диагональные элементы — к единичной матрице. Все проделанные преобразования повторяются и с бывшей единичной матрицей. В результате она преобразуется в обратную к исходной.

2.4. Решение нелинейных уравнений и систем

Здесь рассматриваются простейшие методы решения уравнений вида $f(x) = 0$ (если f — вектор функций векторного аргумента, мы получаем систему уравнений).

2.20. Реализуйте функцию для решения уравнения $f(x) = 0$ с точностью ε методом деления пополам со следующим заголовком

```
int root (double *x, double a, double b,
          double (*f)(double), double eps);
```

Здесь `x` — указатель на полученный корень; `a,b` — границы отрезка, содержащего корень; `f` — указатель на функцию, задающую уравнение; `eps` — точность. Возвращаемое значение: 0, если точность достигнута, -1, если решить не удалось.

Идеи реализации. Метод деления пополам состоит в нахождении середины c отрезка $[a,b]$ и выборе в качестве следующего отрезка, содержащего корень, либо $[a,c]$, либо $[c,b]$ в зависимости от знаков функции f на его концах до тех пор, пока не получим отрезок длины меньше ε. Однако, в реальном вычислительном процессе при малой величине ε может оказаться, что при вычислении по формуле $c = (a+b)/2$ точное значение c округлится до ближайшего представимого числа (a или b). В результате процедура вычисления корня зациклится. Программа должна предусматривать возможность подобной ситуации а также анализировать некорректные входные данные ($f(a)$ и $f(b)$ имеют один знак).

2.21. Реализуйте функцию для решения уравнения $f(x) = 0$ с точностью ε методом Ньютона со следующим заголовком

```
int root (double *x, double (*f)(double),
          double (*df)(double), double eps);
```

Здесь `x` — указатель на начальное приближение и полученный корень; `f` — указатель на функцию, задающую уравнение; `df` — указатель на функцию, вычисляющую производную функции `f`; `eps` — точность. Возвращаемое значение — 0, если точность достигнута, -1, если решить не удалось.

Идеи реализации. Метод Ньютона (метод касательных) состоит в вычислении последовательных приближений к корню по формуле

$$x_0 = x, \quad x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

В качестве предварительного критерия окончания вычислений можно взять неравенство $|x_k - x_{k+1}| < \varepsilon$. Однако, данное неравенство не гарантирует требуемой точности приближения x_{k+1}. Поэтому при выполнении указанного неравенства желательно сделать дополнительную проверку знаков функции f на краях отрезка $[x_{k+1} - \varepsilon, x_{k+1} + \varepsilon]$. Если эта проверка даст одинаковые знаки на концах отрезка, то следует продолжить вычисление приближений, взяв меньшее ε. Если точность не достигается за значительное число итераций (например, 200), то вычисления разумно прекратить и вернуть значение -1.

2.22. Реализуйте функцию для решения уравнения $f(x) = 0$ с точностью ε методом секущих со следующим заголовком

```
int root (double *x1, double *x2, double (*f)(double), double eps);
```

Здесь `x1` — указатель на первое начальное приближение; `x2` — указатель на второе начальное приближение и полученный корень; `f` — указатель на функцию, задающую уравнение; `eps` — точность. Возвращаемое значение — 0, если точность достигнута, -1, если решить не удалось.

Идеи реализации. Метод секущих совпадает с методом Ньютона за исключением того, что вычисление точной производной заменяется разностным соотношением, для которого и требуются две точки. Вычисления ведутся по формуле

$$x_1, x_2 \text{ даны}, \quad x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}.$$

2.23. Проведите тестирование методов деления пополам, Ньютона, секущих на решении уравнений

$$\sin x - a = 0, \qquad e^x - a = 0, \qquad \log_2 x - a = 0, \qquad x^3 + ax = 0$$

при различных значениях a. Сравните число итераций этих методов при одном и том же значении точности.

2.5. Интерполяция и приближение функций

Задача приближения функций обычно ставится следующим образом: заданы набор аргументов x_i и значения функции $y_i = f(x_i)$, $i = 0, .., n-1$.

Требуется составить функцию, вычисляющую приближенное значение функции в некоторой точке x.

2.24. Реализуйте функцию вычисления интерполяционного многочлена Лагранжа с заголовком

```
double Lagrange (double x, double *xi, double *yi, int n);
```

Здесь `x` — точка, в которой вычисляется значение функции, `xi,yi` — массивы значений аргументов и функции, `n` — количество элементов в массивах `xi,yi`, возвращаемое значение — результат вычисления многочлена.

Идеи реализации. Многочлен Лагранжа имеет следующий вид:

$$L_n(x) = \sum_{i=0}^{n-1} y_i \prod_{\substack{j=0 \\ j \neq i}}^{n-1} \frac{x - x_j}{x_i - x_j}$$

При больших n стоит уделить внимание алгоритму по которому происходят вычисления по данной формуле.

2.25. По заданным массивам значений x_i, y_i, $i = 0, \ldots, n-1$ вычислите приближенное значение функции в заданной точке x с помощью

1) кусочно-линейной интерполяции,

2) кусочно-квадратичной интерполяции.

Идеи реализации. Сначала определяется отрезок $[x_k, x_{k+1}]$, которому принадлежит заданная точка x. Затем на данном отрезке функция приближается многочленом первой степени (по значениям $x_k, x_{k+1}, y_k, y_{k+1}$) либо многочленом второй степени (по значениям $x_k, x_{k+1}, x_{k+2}, y_k, y_{k+1}, y_{k+2}$ или $x_{k-1}, x_k, x_{k+1}, y_{k-2}, y_k, y_{k+1}$).

2.26. По заданным массивам значений аргумента x_i, функции y_i и производной функции $dy_i = f'(y_i)$, $i = 0, \ldots, n-1$ вычислите приближенное значение функции в заданной точке x с помощью эрмитовой интерполяции.

Идеи реализации. Сначала определяется отрезок $[x_k, x_{k+1}]$, которому принадлежит заданная точка x. Затем на данном отрезке функция приближается многочленом третьей степени так, чтобы на краях отрезка значения многочлена и его производной совпадали со значениями функции и ее производной. Для этого удобно представить искомый многочлен в виде $a(x-x_k)^3 + b(x-x_k)^2 + c(x-x_k) + d$, что приводит с системе

уравнений для определения неизвестных коэффициентов a,b,c,d:

$$\begin{cases} d &= y_k; \\ c &= dy_k; \\ ah_k^3+bh_k^2+ch_k+d &= y_{k+1}; \\ 3ah_k^2+2bh_k+c &= dy_{k+1}, \end{cases}$$

где $h_k = x_{k+1}-x_k$.

2.27. По заданным массивам значений аргумента x_i и функции y_i, $i=0,\ldots,n-1$, вычислите коэффициенты многочлена наилучшего среднеквадратичного приближения заданной степени k для функции $y=f(x)$.

Идеи реализации. Искомый многочлен $P_k(x)=a_kx^k+\ldots+a_0$ реализует минимум выражения

$$\Phi(P_k)=\sum_{i=0}^{n-1}(y_i-P_k(x_i))^2.$$

Приравнивая нулю производные $\frac{\partial\Phi(P_k)}{\partial a_s}$, получаем систему линейных уравнений для определения неизвестных коэффициентов a_s:

$$\frac{\partial\Phi(P_k)}{\partial a_s}=0, \quad s=0,\ldots,k.$$

Например, для приближения линейной функцией $y=ax+b$ получается система

$$\begin{cases} \sum\limits_{i=0}^{n-1}(y_i-ax_i-b)x_i=0 \\ \sum\limits_{i=0}^{n-1}(y_i-ax_i-b)=0, \end{cases}$$

или после проведения простейших преобразований

$$\begin{cases} \alpha_{11}a+\alpha_{12}b=\beta_1, & \alpha_{11}=\sum\limits_{i=0}^{n-1}x_i^2, & \alpha_{12}=\sum\limits_{i=0}^{n-1}x_i, & \beta_1=\sum\limits_{i=0}^{n-1}x_iy_i, \\ \alpha_{21}a+\alpha_{22}b=\beta_2, & \alpha_{21}=\sum\limits_{i=0}^{n-1}x_i, & \alpha_{2}2=n, & \beta_2=\sum\limits_{i=0}^{n-1}y_i. \end{cases}$$

Теперь значения a и b можно легко определить, выписав решение этой системы в явном виде.

3. Базовые структуры данных

В данном разделе рассматриваются реализации классических структур данных — стека, дека, очереди, списка, дерева. Каждая из этих структур позволяет хранить некоторое количество элементов данных, добавлять новые элементы, извлекать и удалять существующие элементы, а также предоставлять непосредственный доступ к значениям отдельных элементов. Эти структуры различаются между собой по правилам доступа к элементам и взаимными логическими связями между отдельными элементами.

В задачах, приведенных в этом разделе, предлагается построить структуры данных для хранения элементов некоторого абстрактного типа `Type`. Можно считать, что для программы на языке C этот тип определен где-то в программе при помощи оператора `typedef`. Например, для элементов, являющихся целыми числами, мы можем записать

```
typedef  int  Type;
```

В этом случае программная реализация каждой структуры данных должна представлять собой отдельный файл, в котором содержатся объявления некоторого набора глобальных статических объектов (для реализации внутреннего представления данных) и определения функций (для реализации требуемых действий с данными), а также заголовочный файл с описаниями используемых типов данных и прототипами функций.

При использовании языка C++ реализация выливается в построение некоторого класса (или иерархии классов) с соответствующими скрытыми членами и общедоступными методами. Для обеспечения возможности использовать реализацию для различных типов данных, можно воспользоваться параметризацией построенных классов с помощью конструкции `template<class Type>`.

Практическая реализация каждой структуры данных должна сопровождаться тестирующей программой, которая вызывала бы все функции реализации и проверяла их работу в различных ситуациях.

3.1. Стек, дек, очередь

Стек. Так называется структура данных, в которой в каждый момент времени доступным является только элемент, добавленный в стек

последним. Этот элемент называется вершиной стека. Именно этот элемент можно затем извлечь (или удалить) из стека, после чего становится доступным предыдущий добавленный элемент. Эта дисциплина доступа часто обозначается аббревиатурой LIFO (Last In, First Out), т.е. последний добавленный элемент извлекается из стека первым. Непосредственный доступ к значениям хранящихся данных разрешен только для текущей вершины.

3.1. Построить реализацию стека элементов абстрактного типа Type.

Идеи реализации. Для реализации достаточно выделить участок памяти для размещения необходимого количества элементов, размещать поступающие элементы последовательно один за другим рядом друг с другом и хранить указатель на элемент, являющийся текущей вершиной. При добавлении (извлечении) элементов указатель текущей вершины будет перемещаться к следующему свободному (предыдущему занятому) элементу выделенного участка памяти.

Решение. В качестве примера возможного стиля программирования приведем здесь две реализации стека (для определенности тип `Type` есть `int`).

Реализация на языке C

```
/*---   Файл  stack.h  ------------------------------------------*/
typedef int  Type;
int     St_Init (int maxsize);  /* создать стек на maxsize эл-тов */
void    St_Term (void);         /* закончить работу               */
int     St_Push (Type val);     /* добавить элемент  val          */
int     St_Del  ();             /* удалить вершину                */
int     St_Pop  (Type *dst);    /* удалить вершину и поместить ее
                                   значение по указателю dst     */
Type  * St_Top  (void);         /* указатель на вершину           */
int     St_Size (void);         /* текущее кол-во элементов       */
int     St_Room (void);         /* кол-во свободных элементов     */
/*---   конец файла stack.h   -----------------------------------*/
```

Функции `St_Init`, `St_Push`, `St_Del`, `St_Pop` возвращают 0 при успешном выполнении требуемой операции и -1 в противном случае.

```
/*---   Файл  stack.c  ------------------------------------------*/
#include <stdlib.h>
```



```
#include "stack.h"
static  Type    *mem;  /* начало области элементов стека */
static  Type    *top;  /* указатель на вершину стека */
static  int     size;  /* текущее количество элементов */

int     St_Init (int maxsize)
{  size = 0;
   if ( top=mem=malloc(sizeof(TYPE)*maxsize) )
      top+=maxsize;
   return  (mem) ? 0 : -1;
}
void    St_Term (void)
{  free (mem);
   size = 0;
   return;
}
int     St_Push (Type val)
  { return (top!=mem) ? *(--top)=val, ++size, 0 : -1; }
int     St_Pop (Type *dst)
  { return (size) ? *dst=*(top++), --size, 0 : -1; }
int     St_Del ()
  { return (size) ? ++top, --size, 0 : -1; }
Type  * St_Top  (void)
  { return top; }
int     St_Size (void)
  { return size; }
int     St_Room (void)
  { return top-mem; }
/*---   конец файла stack.c   -----------------------------------*/
```

Реализация на языке C++.

```
//---   Файл  stack.h  ----------------------------------------
#define     DEFAULT_SIZE   10

template <class Type>
class  Stack
{  private:
      Type   *mem, *top;
      int       size;
```

```
   public:

// Конструктор и деструктор
   Stack ( int maxsize = DEFAULT_SIZE )
   {  mem = new Type [maxsize];
      top = mem + maxsize; size = 0;
   }
   ~Stack () { delete [] mem; }

// Методы работы со стеком
   int  Push (Type value)
   {  return (top!=mem) ? *(--top)=value, ++size, 0 : -1; }
   int  Pop (Type &dst)
   {  return (size) ? dst=*(top++), --size, 0 : -1; }
   int  Del () { return (size) ? ++top, --size, 0 : -1; }
   Type & Top () { return *top; }
   int  Empty() { return size==0; }
   int  Room () { return top-mem; }
   int  Success () { return (mem) ? 1 : 0; }
};
//---   конец файла stack.h   ---------------------------------
```

Замечание. Функция `Success()` добавлена для того, чтобы мы могли обнаружить отказ в выделении памяти под стек не привлекая механизма исключений языка C++.

3.2. Напишите программу для распечатки полного содержимого стека в наглядной форме. Составьте тестирующую программу, которая бы позволяла в режиме диалога вызывать все функции стека и выводила на экран состояние стека после каждой операции. Проверьте работоспособность реализации для различных типов `Type`.

3.3. Стек строк. Назовем строкой массив символов, заканчивающийся символом конца строки (этот символ обычно имеет код 0x0A и обозначается `\n`). Реализуйте стек таких строк.

Идеи реализации. Выделим достаточно большой блок памяти и будем последовательно укладывать строки одну за другой в этом блоке. Для добавления достаточно поддерживать указатель на начало свободной области в блоке и сравнивать длину этой области с длиной добавляемой строки. Для удаления можно отыскивать в занятой части блока конец

предыдущей строки и передвигать указатель свободной позиции на следующий за этим концом символ.

3.4. Реализация стека строк из предыдущей задачи требует последовательного поиска конца предыдущей строки при удалении элемента. Модифицируйте реализацию, помещая вслед за каждой добавленной строкой ее длину. Это позволит сразу вычислить значение адреса начала последней добавленной строки по адресу начала свободной области в стеке.

3.5. Файловый стек строк. Назовем строкой массив символов, заканчивающийся символом конца строки (обозначение `\n`, код 0x0A). Реализуйте стек таких строк при условии, что строки хранятся в файле, причем начало файла соответствует дну стека, а вершиной стека является последняя строка в файле. При добавлении строк к файлу его размер должен увеличиваться, а при удалении строк — уменьшаться (следует использовать функции fread, fwrite, и т.п.). Реализуйте следующие функции

```
int     OpenFileStack (char *filename);
int     PushString (char *str);
int     PopString (char *str);
int     CloseFileStack (void);
```

Каждая из этих функций возвращает 0 в случае успешного выполнения требуемой операции и -1, если выполнение операции по каким-либо причинам невозможно.

Модифицируйте реализацию так, чтобы уменьшение длины файла при удалении строк происходило не каждый раз, а только при накоплении достаточно большой суммарной длины удаленных строк, например, 1024 байт). Проверьте как повлияет такая модификация на время работы со стеком.

3.6. Файловый стек записей произвольной длины. Требуется реализовать стек, аналогичный стеку из предыдущей задачи, но элементом данных теперь будет являться некоторая байтовая запись произвольной длины. Пусть файловый стек хранит эти записи в следующем формате:

```
<запись 1>
<длина записи 1>
<запись 2>
<длина записи 2>
```

```
    ........
    ........
    <запись k>
    <длина записи k>
```

где <длина записи n> есть четырехбайтовое поле, содержащее значение числа типа unsigned long — длины соответствующей записи. Последняя запись в файле (т.е. <запись k>) соответствует вершине стека.

Реализуйте функции со следующими прототипами

```
int     OpenFileStack (char *filename);
int     PushRecord (void *record, size_t length);
          // функция помещает в стек length байтов,
          // размещенных начиная с адреса record
int     PopRecord  (void *record, size_t *length);
          // функция берет вершину стека и помещает ее
          // в области памяти по адресу record,
          // значение длины записи помещается по адресу length
size_t  TopLength  (void);
          // функция возвращает длину записи
          // на вершине стека
int     CloseFileStack (void);
```

Функции, возвращающие int, возвращают 0 в случае успешного выполнения и -1 — при отказе.

3.7. Постройте реализацию файлового стека строк на основе объединения идей задач XX.XX, XX.XX. Строки частично хранятся в памяти, частично — на диске. В памяти выделяется два блока — первый и второй. Строки накапливаются сначала в первом блоке, а когда он заполнится — во втором. При заполнении второго блока первый блок записывается в файл (по стековому принципу) и блоки обмениваются своими номерами: второй (занятый) блок становится первым, а первый (теперь свободный) становится вторым. При удалении процедура обращается: пока существуют строки в блоках, работа идет в памяти, а когда оба блока становятся пустыми, в первый считываются данные из файлового стека блоков.

Дек. Эту структуру можно рассматривать как обобщение стека путем разрешения добавлять и удалять элементы “с обоих концов” набора данных. При этом мы получаем два текущих элемента, к которым разрешен

доступ и которые называются “начало” и “конец” дека.

Дисциплина доступа к началу и концу дека аналогична дисциплине работы с вершиной стека, т.е. разрешено чтение и изменение значений начала и конца дека, добавление и извлечение элементов из начала или конца дека.

3.8. Постройте реализацию дека элементов абстрактного типа Type.

Идеи реализации. Разместим элементы в выделенном участке памяти вплотную друг к другу и будем хранить два указателя — на начало и конец дека. При добавлении или удалении элементов эти указатели будут перемещаться на позицию следующего или предыдущего элементов. При перемещении указателей возможна ситуация, когда новая позиция указателя уже не попадает в выделенную для хранения элементов область памяти. В этом случае новая позиция “перебрасывается” с позиции за концом области в начало или с позиции перед началом — в конец. Массив оказывается как бы замкнутым в кольцо. Если принять, что начало и конец выделенной области памяти задаются указателями

```
Type *begmem, *endmem;
```

то процедуры определения следующего и предыдущего элементов могут выглядеть, например, так

```
Type * Next (Type *pos)
{ return (pos==endmem) ? begmem : pos+1; }
Type * Prev (Type *pos)
{ return (pos==begmem) ? endmem : pos-1; }
```

Реализуйте дек на языке C со следующим интерфейсом:

```
/*---   Файл  dequeue.h  ----------------------------------------*/
typedef ....  Type;
int     Dq_Init (int maxsize);   /* создать дек                 */
void    Dq_Term (void);          /* закончить работу            */
int     Dq_PushHead (Type val);  /* добавить элемент  val       */
int     Dq_PushTail (Type val);  /*    в начало или конец дека  */
int     Dq_PopHead  (Type *dst); /* взять начало или конец дека */
int     Dq_PopTail  (Type *dst); /* и поместить по указателю dst */
Type  * Dq_Head (void);          /* указатель на начало         */
Type  * Dq_Tail (void);          /*     или конец дека          */
int     Dq_Size (void);          /* текущее кол-во элементов    */
int     Dq_Room (void);          /* кол-во свободных элементов  */
/*---   конец файла dequeue.h   ---------------------------------*/
```

Реализуйте дек на языке C++ в виде следующего класса:

```
//---  Файл  dequeue.h  ----------------------------------
template <class Type>
class   Dequeue
{  private:
      Type  *begmem, *endmem;
      Type  *head, *tail;
      int    size;
      Type * Next (Type *pos);
      Type * Prev (Type *pos);
   public:
      Dequeue (int maxsize);
     ~Dequeue ();
      int    Success  ();
      int    PushHead (Type val);
      int    PushTail (Type val);
      int    PopHead  (Type &dst);
      int    PopTail  (Type &dst);
      Type & Head ();
      Type & Tail ();
      int    Size ();
      int    Room ();
};
//---  конец файла dequeue.h  ----------------------------
```

По поводу функции `Success()` см. замечание к реализации стека.

3.9. Выпуклой оболочкой множества точек называется наименьший выпуклый многоугольник с вершинами к некоторых точках этого множества и содержащий внутри себя все остальные точки множества. Требуется построить выпуклую оболочку заданного множества точек плоскости (т.е. указать точки, являющиеся ее вершинами).

Идеи реализации. Для хранения подмножества точек, составляющих вершины выпуклой оболочки, выделим дек. Точки, лежащие в начале и конце дека будут составлять текущее ребро выпуклой оболочки. Сначала в множестве точек находятся три точки, являющиеся вершинами некоторого невырожденного треугольника. Далее последовательно проверяется в какой полуплоскости относительно линии, содержащей ребро уже построенной выпуклой оболочки, лежит каждая следующая точка

множества. Для этого последовательно рассматривается текущее ребро, которое меняется перекладыванием элементов дека, например, из начала в конец. Таким образом можно выяснить находится точка внутри или вне уже построенной части выпуклой оболочки и при необходимости включить новую точку в выпуклую оболочку, т.е. удалить некоторые стороны и добавить новые две стороны с общей вершиной в этой точке.

Очередь. Дисциплина работы с очередью часто обозначается сокращением FIFO (First In, First Out), т.е. элемент, первым добавленный в очередь (в ее конец), будет также первым взят из (начала) очереди. Как нетрудно видеть, очередь является частным случаем дека, если запретить удаление и доступ к элементу, находящемуся в конце, а также добавление элемента в начало. Доступ к значению хранимых данных разрешается только для элемента, расположенного в начале очереди.

3.10. Реализуйте очередь элементов абстрактного типа `Type` на языке C со следующим интерфейсом:

```
/*---   Файл  queue.h  ----------------------------------------*/
typedef ....  Type;
int     Qu_Init (int maxsize);   /* создать очередь             */
void    Qu_Term (void);          /* закончить работу            */
int     Qu_Add  (Type val);      /* добавить элемент в очередь  */
int     Qu_Take (Type *dst);     /* взять элемент из очереди    */
Type  * Qu_Head (void);          /* указатель на начало очереди */
int     Qu_Size (void);          /* текущее кол-во элементов    */
int     Qu_Room (void);          /* кол-во свободных элементов  */
/*---   конец файла queue.h  ----------------------------------*/
```

3.11. Реализуйте очередь на языке C++ в виде следующего класса:

```
//---   Файл  queue.h  ------------------------------------
template <class Type>
class   Queue
{  private:
      Type  *begmem, *endmem;
      Type  *head, *tail;
      int    size;
      Type * Next (Type *pos);
   public:
```

```
      Queue (int maxsize);
     ~Queue ();
      int    Success  ();
      int    Add  (Type val);
      int    Take (Type &dst);
      Type & Head ();
      int    Size ();
      int    Room ();
};
//---  конец файла queue.h  --------------------------------
```

3.2. Списки

Список можно представить как некоторую структуру данных, где элементы связаны в линейную цепочку и совместно с каждым элементом данных хранится адрес места размещения соседнего с ним элемента. Различают однонаправленный список (хранится адрес только следующего элемента) и двунаправленный список (хранятся адреса следующего и предыдущего элементов). В каждый момент времени в списке определен один текущий элемент с которым и разрешается работать (также может быть разрешена работа с непосредственными соседями этого элемента). Положение текущего элемента определяется так называемым указателем списка. Указатель списка можно перемещать на соседние элементы по направлению ссылок и тем самым получить доступ к любому элементу списка. Добавление и удаление элементов происходит только в окрестности текущего положения указателя списка, при этом новые добавляемые элементы “раздвигают” список. Можно принять различные соглашения по поводу того, какой элемент станет текущим в результате добавления или удаления. Мы будем считать, что при добавлении текущим остается старый текущий элемент, а при удалении новым текущим будет следующий элемент (например тот, который был “правым соседом” бывшего текущего элемента).

3.12. Постройте реализацию двунаправленного списка элементов абстрактного типа `Type` на языке C.

Идеи реализации. Для описания элемента списка будем использовать следующий тип данных:

```
typedef struct L {
```



```
        Type  value;
        struct L *next, *prev;
}  ListItem;
```

Здесь `next` и `prev` являются соответственно указателями на следующий и предыдущий элементы по отношению к данному. Введем дополнительный элемент `base` и свяжем весь список в кольцо через этот элемент. Таким образом, начальный и конечный элементы списка будут ссылаться на адрес элемента `base`.

Текущую позицию в списке будем задавать указателем на текущий элемент:

```
ListItem  *current;
```

Для работы со списком примем следующий интерфейс:

```
/*---   Файл  list2.h  ----------------------------------------*/
typedef struct L {
        Type  value;
        struct L *next, *prev;
}  ListItem;
int     L2_Init (void);          /* создать (пустой) список      */
void    L2_Term (void);          /* закончить работу             */
int     L2_AddBefore (Type val); /* добавить элемент до текущего */
int     L2_AddAfter (Type val);  /* добавить элемент за текущим  */
int     L2_DelCurrent (void);    /* удалить текущий элемент      */
int     L2_ToNext (void);       /* перейти к следующему элементу */
int     L2_ToPrev (void);      /* перейти к предыдущему элементу */
int     L2_ToHead (void);        /* перейти к началу списка      */
int     L2_ToTail (void);        /* перейти к концу списка       */
int     L2_AtHead (void);        /* позиция в начале списка ?    */
int     L2_AtTail (void);        /* позиция в конце списка ?     */
Type  * L2_Current (void);       /* доступ к текущему элементу   */
int     L2_NumElem (void);       /* кол-во элементов в списке    */
/*---   конец файла stack.h   ---------------------------------*/
```

Функция `L2_NumElem` возвращает количество элементов в списке, все остальные функции, возвращающие `int`, возвращают 0 в случае успеха и -1 в случае отказа.

Решение. Приведем в качестве примера реализацию нескольких функций.

```
/*---   Файл  list2.c  ----------------------------------------*/
#include <stdlib.h>
#include "list2.h"
static ListItem  base;
static ListItem *current;
static int   num_elem;

int     L2_Init (void)
{
   current = &base;
   base.next = base.prev = &base;
   num_elem = 0;
   return 0;
}
void    L2_Term (void)
{
   for (L2_ToHead(); !L2_NumElem(); L2_DelCurrent());
}
int     L2_AddBefore (Type val)
{
   ListItem *pos;
   pos = (ListItem*)malloc(sizeof(ListItem));
   if (!pos) return -1;
   pos->prev = current->prev;
   pos->next = current;
   current->prev->next = pos;
   current->prev = pos;
   pos->value = val;
   num_elem++;
   return 0;
}
int     L2_DelCurrent (void)
{
   ListItem *pos;
   if (L2_Empty()) return -1;
   current->prev->next = current->next;
   current->next->prev = current->prev;
   pos = current;
   current = current->next;
   num_elem--;
```



```
   free (pos);
   return 0;
}
int     L2_ToNext (void)
{
   if (L2_AtTail()) return -1;
   current = current->next;
   return 0;
}
int     L2_AtHead (void)
{
   return  (current==base.next);
}
Type  * L2_Current  (void)
{
   return  current;
}
int     L2_NumElem (void)
{
   return  num_elem;
}
```

Остальные функции реализуйте самостоятельно.

3.13. Добавьте к реализации списка из предыдущей задачи функцию последовательного поиска элемента с заданным значением поля `value` (при успехе текущая позиция `current` устанавливается на найденный элемент, при неудаче — не изменяется) и функцию, применяющую заданную процедуру к каждому элементу списка (текущая позиция не изменяется). Заголовки этих функций могут иметь вид

```
int    L2_Search (Type val);
void   L2_Process (void (*action)(Type));
```

Функция поиска должна возвращать 0, если элемент найден, и -1 при отсутствии элемента в списке.

3.14. Постройте реализацию двунаправленного списка элементов абстрактного типа `Type` на языке C++.

3.15. Постройте реализацию двунаправленного списка элементов абстрактного типа `Type` без использования элемента `base`.

Идеи реализации. Свяжем весь список в кольцо (т.е. свяжем ссылками первый и последний элементы). Для запоминания позиций начального и конечного элементов введем два указателя: `ListItem *first, *last`. Для пустого списка значения этих указателей равны нулю. В процедурах добавления и удаления теперь появятся дополнительные проверки, необходимые для корректной работы, когда текущая позиция находится в начале или конце списка.

3.16. Постройте реализации однонаправленного списка элементов абстрактного типа `Type` на языках C и C++ с интерфейсом, аналогичным двунаправленному списку с естественными изменениями (недоступны перемещение и доступ к элементам в одном из направлений).

Идеи реализации. Однонаправленный список можно легко получить соответствующим упрощением реализации двунаправленного списка. Однако теперь недостаточно хранить позицию только текущего элемента (существенно усложняется процедура удаления). Наряду с указателем текущей позиции здесь будет удобно хранить указатель и на предыдущий элемент (назовем этот указатель `before`).

Решение. В структуре `ListItem` оставим только ссылку `next`. Приведем здесь в качестве иллюстрации процедуры удаления текущего элемента и перемещения по списку.

```
int     L1_DelCurrent (void)
{
   ListItem *pos;
   if (L1_Empty()) return -1;
   before->next = current->next;
   pos = current;
   current = current->next;
   num_elem--;
   free (pos);
   return 0;
}
int     L1_ToNext (void)
{
   if (L1_AtTail()) return -1;
   before = current;
   current = current->next;
   return 0;
}
```



```
int     L1_ToHead (void)
{
   if (L1_Empty()) return -1;
   before = &base;
   current = base.next;
   return 0;
}
```

3.17. Добавьте к реализациям списков из задач XX.XX процедуры упорядочивания списка по некоторому критерию. Заголовок соответствующей функции может иметь вид

```
void  Sort (int (*compare)(Type a,Type b));
```

где `compare` — указатель на функцию сравнения значений двух элементов списка. Естественно, при упорядочивании нужно ограничиться только перестройкой взаимных ссылок, не перемещая в памяти сами элементы списка. Реализуйте на списках следующие алгоритмы сортировки (см. раздел 1.3):

1) сортировка выбором максимального элемента;

2) пузырьковая сортировка;

3) сортировка просеиванием;

4) вставка в упорядоченную часть списка с последовательным поиском;

5) слияние подсписков (алгоритм Неймана);

6) быстрая сортировка (алгоритм quicksort) для двунаправленного списка.

Какие из этих алгоритмов могут быть эффективно реализованы на однонаправленных списках?

3.18. Массив списков. Постройте совместную реализацию нескольких списков на языке C. Каждая функция работы со списком при этом получит дополнительный параметр — номер списка. Процедура инициализации получает в качестве параметра начальное количество списков. Прототипы функций могут иметь следующий вид (ср. с задачей XX.XX)

```
int     L_Init (int k);                /* создать k списков  */
int     L_AddAfter (int k, Type val); /* добавить в k-й списков */
```

и т.д.

3.19. Матрица, в которой число ненулевых элементов значительно меньше общего количества элементов, называется разреженной. Сохраняя в элементе списка значения a_{ij} и j для каждого i, мы можем представить такую матрицу как набор списков элементов матрицы по строкам. На основе решения предыдущей задачи реализуйте на языке C процедуры работы с разреженной матрицей со следующим интерфейсом:

```
int     CreateMatr (int m, int n);    /* создать m*n матрицу */
int     Put (double x, int i, int j); /* записать элемент x(i,j) */
double  Get (int i, int j);           /* прочитать элемент x(i,j) */
```

Обратите внимание на то, что нулевые элементы не должны храниться в матрице, т.е. при записи нулевого значения в существующий элемент списка, он должен удаляться из памяти.

3.20. Добавьте к реализации разреженной матрицы из предыдущей задачи функции, выполняющие линейные комбинации и перестановки строк (и столбцов). Используя построенные функции напишите программу решения системы линейных уравнений с разреженной матрицей методом Гаусса.

3.21. Реализуйте разреженную матрицу на языке C++ со следующим интерфейсом:

```
class sparse_matrix
{
 private:
   .....
 public:
   sparse_matrix (int m, int n);
  ~sparse_matrix ();
   double & x(int i, int j);    /* элемент матрицы */
}
```

3.22. Добавьте в класс `sparse_matrix` из предыдущей задачи переопределения арифметических операций с матрицами (сложение, вычитание, умножение).

3.23. На основе реализации разреженной матрицы постройте электронную таблицу. Смысл задачи состоит в автоматической модификации значений некоторых элементов матрицы при изменении значений других элементов. Простейшим примером может служить числовая матрица, в

которой последний столбец содержит суммы элементов соответствующих строк, а последняя строка содержит суммы элементов из соответствующих столбцов. При изменении элементов матрицы эти суммы соответствующим образом модифицируются. Для повышения универсальности реализации можно продумать способы предоставления пользователю возможности менять алгоритмы пересчета элементов матрицы.

3.3. Деревья

Прежде чем формулировать задачи, касающиеся деревьев, напомним ряд необходимых понятий.

Дерево — это структура вполне соответствующая обычному математическому понятию дерева, как связного планарного графа, не содержащего циклов и имеющего одну выделенную вершину — корень дерева. Связь между отдельными вершинами дерева обычно описывается в терминах "родитель—потомок". Каждая вершина дерева, кроме корня, связана с одной и только одной родительской вершиной и с некоторым (возможно, нулевым) числом вершин-потомков. При этом каждая вершина является родительской для своих потомков. Корень не имеет родительской вершины.

Будем говорить, что корень составляет нулевой уровень дерева. Непосредственные потомки корня составляют первый уровень дерева. Непосредственные потомки вершин k-го уровня образуют $k+1$-й уровень дерева.

Если каждая вершина дерева имеет не более двух потомков, то это дерево называется бинарным, в противном случае — произвольным (иногда используется термин “сильно ветвящееся дерево”). Для бинарного дерева можно естественным образом ввести понятия левого и правого поддеревьев.

Вершина дерева называется концевой, если она не имеет потомков. Длиной ветви называется количество вершин в связном участке дерева от корня до концевой вершины. Длиной дерева называется длина максимальной ветви в этом дереве. Дерево называется сбалансированным, если в любой его вершине длины левого и правого поддеревьев отличаются не более, чем на единицу. Дерево называется идеально сбалансированным, если длины двух любых его ветвей отличаются не более, чем на единицу.

Пусть данные, хранящиеся в вершинах дерева, обладают отношением порядка (т.е. их можно сравнивать между собой на “больше–меньше”). Бинарное дерево называется упорядоченным (или деревом поиска), если для любой вершины все элементы левого поддерева меньше либо равны элемента, расположенного в этой вершине, а все элемента правого поддерева строго больше элемента из этой вершины.

С деревьями обычно связаны следующие задачи.

- Обход дерева — требуется выполнить некоторую заданную операцию для каждой вершины дерева. Для бинарных деревьев существует три принципиально различных способа обхода — сверху-вниз (сначала обрабатывается текущая вершина, а затем ее левое и правое поддеревья), снизу-вверх (сначала обрабатываются поддеревья, а затем текущая вершина) и слева-направо (сначала обрабатывается левое поддерево, затем вершина, затем правое поддерево). Процедуры обходов можно обобщить на случай произвольных деревьев.
- Формирование дерева по указанным правилам. Важным частным случаем здесь является работа с обычным или сбалансированным деревом поиска с возможностью добавления, поиска и удаления заданного элемента.

Для реализации бинарных деревьев удобно использовать структуру

```
typedef struct _Tree {
        Type val;
        struct _Tree *left, *right;
} Tree;
```

Если в алгоритме необходимо явное перемещение по направлению к корню, то в эту структуру можно добавить ссылку на родительскую вершину:

```
typedef struct _Tree {
        Type val;
        struct _Tree *left, *right, *up;
} Tree;
```

В рекурсивных процедурах обработки дерева подобное движение обычно реализуется автоматически за счет возврата из последовательности

рекурсивных вызовов, при использовании таких алгоритмов указатель `up` не нужен.

Вершины произвольных деревьев также можно реализовывать с помощью структуры с двумя ссылками.

```
typedef struct _Tree {
        Type val;
        struct _Tree *down, *next;
} Tree;
```

В этом случае смысл указателей несколько меняется: все непосредственные потомки одной вершины связываются в однонаправленный список по указателю `next`, а указатель `down` указывает на начало списка потомков данной вершины.

Отсутствие следующей вершины в списке потомков обычно обозначается нулевым значением соответствующего указателя (`next` или `down`).

3.24. Постройте три упомянутые выше процедуры обхода для подсчета суммы элементов, предполагая, что в вершинах бинарного дерева хранятся целые числа (`Type` есть `int`).

Решение. Пусть корень указанного дерева определяется указателем `root`. Тогда процедуры могут иметь следующий вид.

```
Type  Up_Down (Tree *root)
{
   Type  sum = 0;
   if (!root) return 0;
   sum = root->val;
   sum += Up_Down(root->left);
   sum += Up_Down(root->right);
   return sum;
}
Type  Down_Up (Tree *root)
{
   Type  sum = 0;
   if (!root) return 0;
   sum += Up_Down(root->left);
   sum += Up_Down(root->right);
   sum += root->val;
   return sum;
}
```

```
Type  Left_Right (Tree *root)
{
   Type  sum = 0;
   if (!root) return 0;
   sum += Up_Down(root->left);
   sum += root->val;
   sum += Up_Down(root->right);
   return sum;
}
```

3.25. Постройте аналоги процедур обхода сверху-вниз и снизу-вверх для произвольных деревьев.

3.26. Пусть произвольное дерево построено по следующим правилам: каждая ветвь дерева, начиная с первого уровня, соответствует некоторому слову (текстовой строке); вершины k-го уровня дерева соответствуют k-й букве в записи слова; значением вершины является пара

```
struct { char sym; char flag; },
```

где `sym` — очередная буква слова, а `flag=1`, если существует слово, оканчивающееся на этой букве, и `flag=0`, если слово продолжается дальше. Например, ветвь, задающая слово “столовая”, имеет `flag=1` для первой буквы “о” и букв “л” и “я”; каждый горизонтальный список потомков одной вершины упорядочен по значению поля `sym` (по алфавиту). Требуется построить процедуры добавления, удаления и поиска слова в таким дереве.

3.27. Постройте процедуры обхода для получения следующей информации о деревьях.

1. Определите длину бинарного (или произвольного) дерева (т.е. длину максимальной ветви).

2. Определите количество концевых элементов бинарного (произвольного) дерева

3. Определите количество ветвей, имеющих максимальную длину.

4. Определите количество элементов, лежащих на уровне k.

5. Найдите максимальный элемент среди всех элементов k-го уровня.

6. Проверьте является ли бинарное дерево сбалансированным.

7. Подсчитайте число несбалансированных вершин в бинарном дереве.

8. Подсчитайте показатель сбалансированности для бинарного дерева (т.е. максимальную разницу между длинами правого и левого поддеревьев для каждой вершины).

9. Распечатайте значения всех вершин k-го уровня дерева.

10. Выведите наглядное представление бинарного дерева целых чисел в текстовой файл.

11. Определите длину связного пути между двумя вершинами неупорядоченного дерева, хранящими указанные значения.

12. Пусть дано числовое бинарное дерево поиска. Найдите поддерево, для которого сумма элементов совпадает с указанным числом.

3.28. Пусть концевые вершины бинарного дерева содержат числа, а всем остальным вершинам дополнительно сопоставлены арифметические операции (в начальный момент числовые значения во всех остальных вершинах не определены). Назовем вычислением по дереву процедуру, записывающую в родительскую вершину значение, полученное в результате выполнения указанной арифметической операции со значениями в корнях его поддеревьев. Разработайте форму представления такого дерева и реализуйте процедуру вычисления по дереву.

3.29. Постройте рекурсивную и нерекурсивную процедуру поиска элемента в бинарном дереве поиска. Функция поиска должна возвращать указатель на найденный элемент или `NULL`, если такого элемента в дереве нет.

3.30. Постройте рекурсивную процедуру добавления элемента в бинарное дерево поиска.

Идеи реализации. Начиная с корня, определяем, в какое поддерево должен попасть данный элемент, и рекурсивно вызываем процедуру добавления для требуемого поддерева. Возвращаемое значение функции — указатель на корень дерева после добавления элемента или `NULL`, если не удалось добавить.

Решение.

```
Tree  * T_Add  (Type x, Tree *pos)
{
   if (!pos)
   {  if ( !(pos=GetPlace()) ) return NULL;
      pos->value = x;
      pos->right = pos->left = NULL;
```

```
   }
   else
   {  if ( x>pos->value )
      {  pos->right = T_Add (x,pos->right);
         if ( !pos->right )  return NULL;
      }
      else
      {  pos->left  = T_Add (x,pos->left);
         if ( !pos->left )  return NULL;
      }
   }
   return pos;
}
```

3.31. Реализуйте рекурсивную процедуру удаления элемента из бинарного дерева поиска.

Идеи реализации. Если удаляемая вершина имеет не более одного потомка, то удаление сводится к перестановке ссылки от родительской вершины на соответствующего потомка (возможно пустого). Если удаляемая вершина имеет два потомка, то сначала в левом поддереве ищется вершина с максимальным значением поля `val` (обозначим эту вершину через M), это значение записывается в поле `val`, удаляемой вершины и далее из дерева удаляется вершина M (докажите, что вершина M имеет не более одного потомка).

Решение. Возвращаемое значение функции — указатель на корень дерева после удаления элемента.

```
Tree * T_Del (Type x, Tree *root)
{
   Tree *pos;
  /* пустое дерево */
   if ( !root )  return 0;
  /* непустое дерево */
   if ( x==root->val )
   {  if ( root->left==0 )
      {  pos = root->right;
         FreePlace(root);
         return pos;
      }
```



```
      if ( root->right==0 )
      {  pos = root->left;
         FreePlace(root);
         return pos;
      }
     /* есть оба потомка --- поиск элемента для подмены */
      for ( pos=root->left; pos->right; pos=pos->right) ;
      root->val = pos->val;
      root->left = T_DelElement(pos->val,root->left);
   }
   else
   {  if ( x<root->value )
         root->left = T_DelElement(x,root->left);
      else
         root->right = T_DelElement(x,root->right);
   }
   return  root;
}
```

3.32. Постройте нерекурсивные процедуры добавления и удаления элементов в бинарном дереве поиска.

3.33. Постройте процедуры добавления и удаления элементов в бинарном дереве поиска при условии, что связи между вершинами задаются тремя указателями — `left`, `right`, `up`.

3.34. В рассмотренной выше процедуре добавления в дерево поиска элементы с одинаковыми значениями могут оказаться в дереве далеко друг от друга. Измените процедуру добавления так, чтобы элементы с одинаковыми значениями всегда оказывались рядом друг с другом (т.е. образовывали в ветви линейную цепочку).

Трудоемкость процедур поиска, добавления и удаления элемента в произвольном бинарном дереве поиска оценивается по порядку величиной, равной длине максимальной ветви дерева. Если дерево содержит N элементов, то в наихудшем вырожденном случае эта величина равна N, а в наилучшем случае — $\log_2 N$. Оказывается, что можно организовать работу с деревом поиска (т.е. добавление и удаление элементов) так, чтобы оно всегда оставалось сбалансированным, при этом длина максимальной ветви не будет превосходить величины $1.5\log_2 N$. В данном случае

нам придется ограничиться деревьями, все элементы которых различны (действительно, дерево, содержащее только одинаковые элементы, не может быть одновременно упорядоченным и сбалансированным в смысле приведенных выше определений).

3.35. Докажите, что длина сбалансированного дерева с N элементами не превосходит $1.5\log_2 N$.

Для реализации сбалансированных деревьев нам придется модифицировать структуру его вершины. Назовем балансом вершины разность между значениями длин его правого и левого поддеревьев. Будем хранить значение баланса каждой вершины вместе с остальной информацией данной вершины, т.е. будем использовать, например, следующее определение типа вершины:

```
typedef struct _Tree {
        Type val;
        int  balance;
        struct _Tree *down, *next;
} Tree;
```

Для сбалансированного дерева значения поля `balance` в каждой вершине могут быть $-1, 0, 1$. Пусть мы имеем сбалансированное дерево поиска. Поставим следующую задачу: организовать добавление и удаление элементов в это дерево так, чтобы сохранить сбалансированность дерева. Заметим, что если при добавлении элемента в некоторое поддерево длина этого поддерева не возросла, то баланс в вершине, родительской по отношению к этому поддереву, не изменился и, следовательно, балансировка не нужна. Если же длина поддерева возросла, то, возможно, придется перестраивать дерево для восстановления утраченного баланса. Алгоритмы балансировки деревьев при добавлении и удалении элементов описаны в [2], и мы их здесь не будем приводить.

3.36. Реализуйте функцию, которая проверяет является ли данное бинарное дерево а) идеально сбалансированным; б) просто сбалансированным.

3.37. Реализуйте функцию, которая обходит бинарное дерево (не обязательно сбалансированное) и записывает в поле `balance` значение баланса каждой вершины.

3.38. Реализуйте процедуры добавления и удаления элементов в сбалансированном бинарном дереве поиска элементов абстрактного типа `Type`.

При организации поиска на внешних носителях, т.е. когда все данные не помещаются в оперативную память, выгодно использовать так называемые B-деревья, позволяющие находить данные в больших файлах за малое количество обращений к диску.

B-деревом порядка m называется дерево, обладающее следующими свойствами:

1. Каждая вершина дерева содержит не более $2m$ элементов данных.
2. Каждая вершина, кроме корня и концевых элементов, имеет не менее m элементов данных.
3. Корень, если он не концевой элемент, имеет не менее 2 потомков.
4. Вершина с k элементами данных имеет ровно $k+1$ потомков.
5. Все концевые элементы расположены на одном уровне.

Предположим, что мы заранее можем зафиксировать значение m (например, $m = 100$). Тогда вершины B-деревьев порядка m можно реализовать с помощью структуры с двумя массивами

```
#define m 100
typedef struct _BT {
   Type    val[2*m];         // массив элементов
   struct _BT *down[2*m+1]; // указатели на корни потомков
   int k;                    // текущее количество элементов
} BTree;
```

Удобно считать, что для каждого элемента `val[i]` определены “левое” поддерево (указатель `down[i]`) и “правое” поддерево (указатель `down[i+1]`). Для концевых вершин эти указатели `down` считаются равными нулю. B-дерево называется деревом поиска, если данные в каждой вершине упорядочены, например, по возрастанию `val[i]<val[i+1]` и для каждого элемента `val[i]` все элементы левого поддерева меньше его, а все элементы правого поддерева больше его.

Несложно показать, что если в B-дереве хранится N элементов данных и в одной вершине мы можем хранить до $k = 2m$ элементов, то концевые элементы будут расположены на уровне $l \approx \log_{k/2} N$. При $k = 200$, $N \sim 10^6$ имеем $l \sim 3$.

3.39. Реализуйте функцию поиска элемента данных в B-дереве с заголовком

```
Type  *  BTreeSearch (Type x, BTree *root);
```

Идеи реализации. Так как по условию построения B-дерева массив `val` в каждой вершине является упорядоченным, то мы можем методом деления пополам либо найти искомый элемент x, либо определить позицию, где x можно вставить в этот массив. Для такой позиции j имеем `val[j-1]<x<val[j]` (либо `x<val[0]`, либо `val[k-1]<x`). В этом случае мы рекурсивно переадресуем поиск на поддерево, определяемое указателем `down[j]` (соответственно `down[0]`, или `down[k]`). Если очередное поддерево пусто, то элемент отсутствует.

Опишем схему добавления элемента в B-дерево поиска. Сначала отыскивается концевая вершина, в которую попадает добавляемый элемент. Если в массиве `val` этой вершины есть свободное место (т.е. $k < 2m$), то элемент добавляется в массив так, чтобы сохранить упорядоченность. В противном случает требуется перестройка узла и, возможно, всего дерева.

Первый способ перестройки заключается в выталкивании элемента `val[m]` в родительскую вершину и расщеплении исходной вершины на две части: с элементами `val[i]` для $0 \le i < m$ и элементами для $m+1 \le i < 2m$. Массив данных и массив ссылок на поддеревья в родительской вершине корректируются соответствующим образом. Если при этом происходит переполнение родительской вершины, то расщепляем аналогичным образом, и т.д. (возможно, вплоть до корня). При расщеплении корня создается новый корень.

Второй способ заключается в попытке перемещения части "лишних"данных в соседнюю правую (либо левую) вершину. При этом необходимо модифицировать родительскую вершину. Если соседние вершины заполнены, то производится расщепление первым способом.

Третий способ заключается в объединении данных двух соседних вершин и их расщеплении в три новых вершины. При этом необходимо модифицировать родительскую вершину. Если у расщепляемой вершины нет соседних, то это — корень. Корень расщепляется на два узла.

Алгоритм удаления можно получить обращением описанной процедуры. Если при удалении элемента из массива `val` количество данных в

нем станет меньше m, то необходима перестройка, которая может быть выполнена перемещением данных из соседних вершин или слиянием двух соседних вершин в одну новую. Более подробно алгоритмы работы с B-деревьями описаны в [2].

3.40. Реализуйте процедуры добавления и удаления элементов некоторого типа `Type` на базе B-дерева.

3.41. Разработайте формат файла для сохранения B-дерева на диске и реализуйте процедуры добавления, удаления и поиска элементов некоторого типа `Type` на базе B-дерева, при условии, что все дерево не помещается в оперативную память.

Идеи реализации. Пронумеруем все вершины дерева последовательными натуральными числами. В начале файла должна содержаться таблица, которая устанавливает соответствие между номером вершины и смещением от начала файла, определяющим местоположение этой вершины. Поскольку количество вершин в дереве может меняться, то следует предусмотреть способы для удлинения этой таблицы. За этой таблицей размещаются вершины B-дерева. Прочитав данную таблицу в оперативную память, мы получаем возможность считывания любой вершины по ее номеру.

Для хранения вершины B-дерева возьмем структуру следующего вида

```
typedef struct _BT {
   Type val[2*m];     // массив элементов
   int  down[2*m+1]; // массив номеров вершин след. уровня
   int  k;           // текущее количество элементов
   int  number;      // номер данной вершины
} BTree;
```

Пусть мы имеем возможность разместить в памяти N вершин. Создадим список (массив), в котором будем хранить номер вершины, указатель на начало ее размещения в памяти, смещение в файле для данной вершины, количество обращений к элементам данной вершины. Если нужная нам вершина уже расположена в памяти, то указанная информация обеспечивает возможность доступа к ее элементам (при этом следует корректировать поле, относящееся к количеству обращений), а также запись вершины из памяти обратно в файл. Если вершины в памяти нет, то она

считывается с диска и информация о ней записывается в свободную ячейку этого списка (массива). Если список уже содержит N значений, то мы должны освободить один из его элементов, переписав одну из вершин обратно на диск. При этом можно воспользоваться одной из следующих стратегий:

1) освобождается та вершина, которая имела наименьшее количество обращений (освобождается наименее используемая вершина);

2) освобождается та вершина, которая имела наибольшее количество обращений (возможно, элементы этой вершины уже обработаны и далее не потребуются);

3) подсчитывается общее число m обращений ко всем элементам B-дерева; при достижении числом m некоторого порогового значения, количество обращений к каждой отдельной вершине обнуляется; освобождается та вершина, которая в данный момент имеет наименьшее количество обращений (т.е. реже всего используется).

3.4. Графы

Граф — это пара (V,E), где V — множество вершин, а E — множество ребер (т.е. пар вершин) или дуг (т.е. ориентированных пар вершин). Каждой вершине и каждому ребру могут быть дополнительно сопоставлены некоторые наборы данных. В этом случае говорят о нагруженном графе. Реализация графов в программировании может быть выполнена различными способами. Наиболее часто применяется явное перечисление множеств V и E в виде таблиц, либо ссылочные конструкции по типу деревьев. Еще один способ представления графа — матрица, где элемент с индексами i,j соответствует нагрузке ребра между i-й и j-о вершинами графа.

Наиболее характерные задачи — это анализ и нахождение путей (т.е. связных последовательностей ребер) в графе.

3.42. Пусть вершины графа идентифицируются целыми числами от 0 до $N-1$ и конфигурация графа задана в текстовом файле в виде строк:

```
<номер вершины>  <список номеров смежных вершин>
```

Разработайте внутренне программное представление графа и реализуйте функции, формирующие граф и сохраняющие граф в файле в таком же виде.

3.43. Рассматриваются графы с нагруженными ребрами. Пусть вершины графа идентифицируются целыми числами от 0 до $N-1$, нагрузка ребра есть вещественное число и конфигурация графа задана в текстовом файле в виде строк:

```
<номер вершины> <номер вершины> <нагрузка ребра>
```

Например, треугольник со сторонами 4, 4.2, 5.1 может быть представлен следующими строками:

```
0 1  4
1 2  4.2
2 0  5.1
```

Разработайте внутренне программное представление графа и реализуйте функции, формирующие граф и сохраняющие граф в файле в таком же виде.

3.44. Назовем циклом замкнутую связную последовательность ребер (дуг) графа. Пусть вершины графа идентифицируются целыми числами от 0 до $N-1$. Постройте функцию, которая выводит номера вершин, принадлежащих каждому циклу графа.

3.45. Назовем путем между двумя вершинами замкнутую связную последовательность ребер, начинающуюся в одной вершине и заканчивающуюся в другой. Постройте функцию, которая выводит номера вершин, принадлежащих каждому пути между двумя указанными вершинами графа.

3.46. Пусть каждому ребру графа приписан некоторый вес (нагрузка ребра — вещественное число). Назовем весом пути сумму весов всех ребер, составляющих этот путь. Постройте функцию, которая определяет путь с минимальным весом между двумя указанными вершинами графа.

3.47. Разработайте форму представления плоского лабиринта в виде планарного графа и напишите программу, которая из заданной точки внутри лабиринта отыскивает путь к выходу (или говорит, что выход невозможен). Дополните программу графическим выводом результатов поиска.

3.48. Пусть даны два (ненагруженных) графа. Постройте функцию, которая определяет совпадают ли эти два графа. Два графа совпадают,

если они становятся тождественными после некоторой перенумерации вершин.

3.49. Назовем приведением графа следующую процедуру. Если вершина a соединена ровно двумя вершинами b и c, которые не были ранее связаны ребром bc, то она удаляется из графа, а ребра ab и ac заменяются на ребро bc. Указанная операция проводится то тех пор, пока в графе не останется вершин, которые можно было бы удалить. Реализуйте функцию, выполняющую приведение графа.

3.50. Назовем два графа подобными, если они совпадают после приведения (см. задачи 3.48, 3.49) Реализуйте функцию, выясняющую подобны ли два графа.

4. Контейнеры и множества

Задачи данного раздела связаны с построением программных реализаций, предназначенных для хранения элементов данных без учета их взаимной связи. Фактически речь идет о реализации множества элементов заданного типа с общепринятыми операциями добавления элемента в множество, удаления элемента и ответа на вопрос о принадлежности данного элемента конкретному множеству.

К множествам примыкают так называемые контейнеры. Мы будем называть контейнером программную реализацию, предназначенную для размещения в памяти разнообразных наборов данных с дополнительной возможностью последующего освобождения места, занимаемого этими наборами.

Языки C и C++ фактически имеют встроенную реализацию контейнеров, основанную на функциях malloc и free (или операторах new и delete). Однако, в ряде случаев использование дополнительной информации о типе размещаемых данных позволяет более эффективно организовать использование памяти. Кроме того, задача построения эффективного контейнерного способа работы с памятью интересна сама по себе.

Простейшую реализацию множества однородных элементов можно построить на базе массива. Основной проблемой при этом является необходимость заранее определить необходимую длину массива.

4.1. Постройте реализацию множества элементов абстрактного типа `Type` на базе ограниченного массива при условии, что заранее задается

максимальное количество элементов множества и элементы нельзя упорядочить. Для множества, содержащего N элементов сложность поиска должна составлять в среднем $O(N)$ операций, а сложность добавления и удаления элементов (без учета поиска) — $O(1)$ операций.

Идеи реализации. Для реализации выделим участок памяти под максимально допустимое количество элементов и будем размещать поступающие элементы последовательно, добавляя очередной элемент в конец цепочки. Для удаления элемента из середины цепочки просто запишем на его место элемент, стоящий на последнем месте в занятой части массива.

4.2. Постройте реализацию множества элементов абстрактного типа `Type` на базе ограниченного массива при условии, что заранее задается максимальное количество элементов множества и элементы можно упорядочить. Для множества, содержащего N элементов, сложность поиска должна составлять в среднем $log_2 N$ операций, а сложность добавления и удаления элементов (без учета поиска) — $O(N)$ операций.

Идеи реализации. Считая, что элементы массива можно сравнивать на “больше—меньше”, будем поддерживать упорядоченный массив. Поиск элементов осуществляется методом деления пополам, а добавление и удаление — вставкой и удалением в упорядоченном массиве, что потребует перемещения части элементов массива.

4.1. Динамический массив

Ограничение на начальную длину массива можно снять, если реализовать динамический массив, в котором количество элементов может увеличиваться или уменьшаться в процессе работы. Идея реализации подобного массива состоит в следующем. Выделим блок некоторой длины и разместим элементы массива в этом блоке. Если одного блока недостаточно для размещения всех элементов массива, то выделим еще один блок, и.т.д. При добавлении элемента в массив этот элемент размещается в свободном участке последнего блока или в новом блоке, если последний блок не имеет свободного места. Таким образом, для доступа к элементу массива с конкретным индексом k необходимо определить блок, в котором находится этот элемент, и порядковый номер данного элемента внутри данного блока.

4.3. Реализуйте динамический массив элементов абстрактного типа `Type` на основе однонаправленного списка блоков следующего типа:

```
#define BLK_LEN 100
typedef struct _Block {
        struct _Block *next;
        Type   elem [BLK_LEN];
} Block;
```

При увеличении количества элементов в массиве новый блок включается в этот список, а при уменьшении количества элементов последний блок исключается из списка.

4.4. Добавьте к предыдущей реализации функцию сортировки динамического массива (рассмотрите применение всех базовых методов сортировки) и функцию поиска заданного элемента в упорядоченном динамическом массиве методом деления пополам. Протестируйте реализацию, чтобы определить во сколько раз в среднем увеличивается время работы по сравнению с применением тех же алгоритмов для обычных массивов.

4.2. Битовая реализация множества

Рассмотрим один важный частный случай. Требуется реализовать работу с произвольными подмножествами множества целых чисел, принимающих значения в диапазоне от 0 до некоторого $p-1$. Выделим массив из p битов и сопоставим целому числу k элемент этого битового массива с индексом k. Теперь, если k-й бит нашего массива равен 1, то число k принадлежит множеству, если k-й бит есть 0, то число k не принадлежит множеству. Таким образом, работа с множеством свелась к проверке или установке соответствующих битов. В стандартном языке C нет битового типа данных, поэтому нам придется организовывать работу с битами самим по аналогии с динамическим массивом. Итак, возьмем массив четырехбайтовых целых чисел (отдельные ячейки), которые будут соответствовать блокам в динамическом массиве. В каждом таком числе размещается 32 бита. Теперь для данного конкретного числа (элемента множества) нам достаточно вычислить номер ячейки и номер бита внутри данной ячейки. Эти вычисления легко выполняются делением на 32 с остатком. Заметим, что деление на 32 (степень двойки) более эффективно

выполнять сдвиговыми операциями, а вычисление остатка — побитовым логическим умножением на константу $0x1F$.

Приведем одну из возможных реализаций на языке C++.

```
typedef  unsigned long  Bitcell_t;    // четырехбайтовая ячейка

//  Bitcell  --- битовое множество на основе одного числа
//               типа unsigned long (четыре байта).
class Bitcell
{
 private:
   Bitcell_t  c;
 public:
   Bitcell () { c = 0; }
  ~Bitcell () {}
   void  Put (int x) { c |= (1<<x); }
   void  Del (int x) { c &=~(1<<x); }
   int   Exists (int x) { return (int)(c&~(1<<x)); }
   void  Clear () { c = 0; }
   int   Empty () { return (int)(!c); }
};

//  Bitset   --- битовое множество на основе массива элементов
//               типа Bitcell.
class Bitset
{
 private:
   Bitcell  *c;
   int       m,mx;
          //  проверка допустимого диапазона чисел
   int    Valid (int x) { return (x>=0) && (x<mx); }
          //  вычисление номера ячейки
   int    cell_n(x) { return (x)>>5; }
          //  вычисление номера бита в ячейке
   int    cell_b(x) { return (x)&0x1F }
 public:
   Bitset (int n) { mx=n; m=cell_n(n)+1; c=new Bitcell[n]; }
  ~Bitset () { delete [] c; }
   int   OK() { return (c) ? 1 : 0; }
   void  Put (int x) { if (Valid(x)) c[cell_n(x)].Put(cell_b(x)); }
```

```
   void  Del (int x) { if (Valid(x)) c[cell_n(x)].Del(cell_b(x)); }
   int   Exists (int x)
         { return (Valid(x)) ? c[cell_n(x)].Exists(cell_b(x)) : 0; }
   void  Clear () { for(int i=0;i<m;i++) c[i].Clear(); }
};
```

4.5. Добавьте к предыдущей реализации функции-члены для выполнения следующих действий:

а) взять (и удалить из множества) один какой-нибудь присутствующий там элемент;

б) для каждого элемента, присутствующего в множестве, выполнить указанную операцию, задаваемую указателем на функцию `void (*action)(int);`

4.6. Реализуйте битовое множество на базе массива двухбайтовых ячеек.

4.7. Добавьте к реализации битового множества на языке C++ операции или функции, выполняющие объединение, пересечение, вычитание, дополнение множеств и т.п.

4.8. Составьте программу вычисления простых чисел на основе битовой реализации множества и алгоритма решета Эратосфена. Напомним, что этот алгоритм состоит в создании множества, содержащего все натуральные числа до некоторого максимального значения и последующем удалении чисел, делящихся на 2, 3, 5, и т.д. (т.е. на уже определенные ранее простые числа). Проведите тестирование соотношения (время работы)/(количество простых чисел) при использовании различных объемов оперативной памяти для хранения множества.

4.3. Хеш-реализации множеств

Для реализации множества произвольных элементов можно использовать списки или деревья, поскольку эти структуры позволяют легко построить процедуры поиска требуемого элемента. Однако, в ряде случаев можно ускорить поиск элемента, если применить хеширование. Суть хеширования состоит в том, что строится массив из p однотипных множеств и вводится некоторая функция `int Hash (Type x)` (так называемая хеш-функция), возвращающая значения от 0 до $p-1$. Теперь при поступлении элемента x первым делом вычисляется значение

$k = Hash(x)$ и работа с данным элементом переадресуется множеству с индексом k. Если хеш-функция подобрана удачно, то мощность каждого k-го подмножества будет примерно в p раз меньше мощности всего множества, с которым мы в данный момент работаем. Таким образом, мы будем тратить в среднем в p раз меньше времени на работу с множеством (при условии, что хеш-функция вычисляется достаточно быстро).

4.9. Реализуйте хеш-множество элементов некоторого типа `Type` на базе массива списков.

Идеи реализации. Пусть хеш-значение для данного типа `Type` вычисляется функцией

```
int Hash (Type x);
```

возвращающей значения от 0 до $p-1$.

В рамках языка C++ решение является прямолинейным — строим класс `List1<Type>` — однонаправленный списков элементов типа `Type` (см. Главу 3) с дополнительной процедурой последовательного поиска требуемого элемента. Далее просто берем массив таких классов.

Для языка C решение сводится к реализации массива списков (см. задачу XX.XX)

В некоторых случаях достаточно эффективным оказывается следующий подход, называемый методом проб (см. [Вирт]). Выделим массив из p элементов типа `Type` (назовем его для определенности `A`) и будем считать, что нулевое значение соответствует отсутствию элемента в множестве (это достаточно естественно, когда `Type` представляет собой некоторый указатель). В начале работы все значения `A[i]` равны нулю. Для заданного элемента множества `x` вычислим значение хеш-функции `k=h(x)` и разместим `x` в элементе массива `A[k]`. Если элемент `A[k]` уже хранит некоторое ненулевое значение, то ищем ближайший свободный элемент массива справа от `A[k]` (т.е. среди `A[k+1]`, `A[k+2]` ...) и в нем размещаем наш элемент `x`. При достижении правой границы `A[p-1]` — циклически переходим к началу массива. Поиск заданного элемента `x` производится аналогично — сначала проверяется соответствующий хеш-функции элемент `A[k]`, а затем при необходимости — элементы `A[k+1]`, `A[k+2]` ... до тех пор, пока либо не будет найден `x`, либо пустое место (в этом случае `x` не содержится в множестве). Если длина массива

p превосходит количество элементов в рабочем множестве, то подобная процедура поиска будет с большой вероятностью завершаться после небольшого количества таких проверок (проб).

4.10. Реализуйте функции добавления и поиска элементов в хеш-множестве по методу проб.

4.11. Разработайте процедуру удаления заданного элемента из хеш-множества, реализованного по методу проб. Учтите, что элементы с одним хеш-значением не обязательно будут расположены рядом друг с другом, но между ними не должно быть пустых ячеек массива.

4.12. Реализуйте множество слов (текстовых строк) по методу проб. Сами слова размещаются в памяти с помощью функций `malloc`, а указатели на них — в хеш-множество. Хеш-функцию постройте на основе идей задачи XX.XX.

4.13. Пусть p — количество значений хеш-функции (длина базового массива), n — некоторое усредненное количество элементов множества, с которым нам приходится работать. Проведите тестирование метода проб, подсчитав среднее количество проб при поиске, добавлении, удалении элемента в зависимости от заполненности базового массива (т.е. от отношения p/n). Рассмотрите различные значения p/n $(0.3, 0.5, 0.7, 0.9)$.

Эффективность работы данных алгоритмов существенно зависит от удачного выбора хеш-функции. Рассмотрим один из возможных способов построения такой функции для множества слов.

4.14. Рассмотрим некоторое множество слов и будем вычислять хеш-функцию по следующей формуле

$$h(x) = \sum_i \alpha_i x_i \pmod p,$$

где x_i — коды букв данного слова x, α_i — некоторые заранее заданные весовые коэффициенты, p — число, определяющее диапазон значений хеш-функции. При фиксированном p, попробуйте экспериментально подобрать набор коэффициентов α_i так, чтобы хеш-функция наиболее равномерно рассеивала слова по хеш-подмножествам для исходного множества слов. На данную задачу можно посмотреть как на проблему минимизации дисперсии $D(\xi(\alpha))$ последовательности ξ_i (ее значение

- количество слов в i-м хеш-подмножестве) по набору коэффициентов α и применить стандартный алгоритм нахождения минимума функции многих переменных.

Протестируйте работу программы на следующих источниках множества слов:

1. большой текст на русском языке;
2. большой текст на английском языке;
3. текст программы на языке C;

4.4. Контейнеры

Как уже говорилось выше, контейнером называется программная реализация, предназначенная для выделения памяти под размещение некоторого набора данных с возможностью последующего освобождения этой памяти. Возможность повышения эффективности алгоритмов управления памятью по сравнению со стандартными системными средствами опирается на отказ от универсальности процедур выделения памяти и учет конкретных особенностей решаемых задач. Важным частным случаем является размещение элементов данных заранее известного фиксированного размера.

4.15. Реализуйте контейнер для элементов заданного типа `Type` на базе одного заранее выделенного блока памяти.

Идеи реализации. Введем следующий тип данных “ячейка контейнера”:

```
typedef union U {
        Type  element;
        union U * next;
} Cell;
```

Зададимся некоторым ограничением M на максимальное количество элементов контейнера и создадим массив из M подобных ячеек. В начале работы все ячейки считаются свободными и связываются в однонаправленный список с помощью поля `next`. При запросе на выделение памяти берется первая ячейка из списка свободных, и ее поле `element` используется для хранения значения размещаемого элемента. При освобождении ячейки, она опять включается в начало списка свободных. Реализуйте описанные идеи в виде функций

```
int    InitContainer (int max_elem);
Type * AllocMem ();
void   FreeMem  (Type *x);
void   FreeContainer ();
```

Последняя функция в этом списке освобождает контейнер и возвращает весь выделенный блок в свободную системную память. В реализации следует предусмотреть отказ по недостатку памяти в выделенном блоке.

4.16. Что произойдет, если реализациях предыдущей задачи повторно освободить ранее освобожденный элемент? Устраните возникающие при этом проблемы, добавив к предыдущей реализации битовое множество для идентификации занятых и свободных ячеек в блоке контейнера.

4.17. Реализуйте контейнер для элементов заданного типа `Type` на базе одного заранее выделенного блока памяти с использованием только битового множества свободных ячеек.

Идеи реализации. Для хранения элементов данных выделяется отдельный блок. Дополнительно создается битовое множество, в котором единицы соответствуют занятым, а нули — свободным ячейкам блока. Свободные блоки для добавления данных в множество отыскиваются просмотром битового множества.

4.18. Замените в реализациях списков и деревьев задач главы 3 выделение места под элементы из свободной памяти (функции `malloc` и `free`) на работу с контейнерами, описанными в предыдущих задачах.

4.19. Реализуйте контейнер для размещения в памяти элементов заданного фиксированного размера на основе динамического массива блоков.

Идеи реализации. Для хранения элементов с помощью функции `malloc` выделяется блок заранее оговоренного размера. В каждом блоке помещается фиксированное количество ячеек. В начале блока дополнительно содержится битовое множество, показывающее занятые и свободные места для элементов (каждый бит соответствует одному элементу). Свободное место в блоке определяется просмотром этого битового множества. При полной занятости данного блока выделяется новый блок, при освобождении всех элементов в данном блоке этот блок возвращается в свободную память с помощью функции `free`. При добавлении элемента нужно искать блок со свободными полями. При освобождении

элемента нужно искать блок, соответствующий освобождаемому адресу. Простейший подход к реализации этих процедур поиска — связать блоки в однонаправленный список и последовательно просматривать этот список, пока не будет найден блок, включающий искомый адрес (удаление элемента) или имеющий свободное поле (добавление элемента). Другой более эффективный подход состоит в размещении адресов блоков в узлах сбалансированного бинарного дерева (упорядоченность с соответствии со значениями адресов блоков) вместе с двухбитовым полем, показывающем есть ли в левом и правом поддеревьях блоки со свободными полями.

4.20. Реализуйте контейнер текстовых строк без операции удаления. Контейнер может иметь следующий интерфейс:

```
int    InitStrContainer ();
char * PutString (char *str);
void   FreeStrContainer ();
```

Функция `PutString` возвращает указатель на место размещения добавленной строки `str`, либо 0 при отказе.

Идеи реализации. Для хранения строк с помощью функции `malloc` выделяется блок заранее оговоренного размера. Строки размещаются в блоке последовательно одна за другой (поддерживается указатель на начало свободной области в блоке). Если очередная строка не помещается в остатке блока, то выделяется новый блок, и строка размещается в нем целиком (т.е. не режется между блоками). Отдельные блоки связываются в однонаправленный список, необходимый чтобы вернуть все блоки в свободную память в конце работы.

4.21. Реализуйте контейнер байтовых записей произвольной длины.

Идеи реализации. Выделяется один большой блок памяти. Непосредственно перед каждой записью располагается служебная область, содержащая следующие данные:

а) длину этой записи в байтах;

б) адрес следующей записи;

в) адрес предыдущей записи,

т.е. все записи фактически образуют двусвязный список. Место для добавления новой записи в первую очередь ищется после последней размещенной записи. При отсутствии там достаточного места просматривается

список и ищется первый подходящий свободный промежуток, где можно разместить требуемую запись. При добавлении и удалении соответственно корректируются адреса следующих и предыдущих записей в списке.

4.5. Моделирование файловой системы

Работа файловой системы в любой операционной системе во многом напоминает работу контейнеров — решаются задачи размещения и удаления файлов. Однако, здесь появляется дополнительное условие — созданный файл может изменять размер в процессе своего существования. Файловые системы обычно отводят для хранения каждого файла целое число отдельных блоков некоторой фиксированной длины. Поэтому одной из основных функций файловой системы является определение множества блоков, принадлежащих указанному файлу, а изменение размера файла сводится к добавлению или удалению блока в этом множестве.

Не касаясь проблем, связанных с реальными дисковыми операциями чтения и записи, в этом разделе мы рассмотрим контейнеры, моделирующие работу файловой системы, т.е. позволяющие изменять размер хранимого набора данных и обеспечивающие в некотором смысле доступ к информации, хранящейся в данном наборе. Будем считать, что в нашем распоряжении есть достаточно большой участок оперативной памяти, который мы по аналогии будем называть “диском”, а отдельный набор данных будем называть “файлом”. Задача состоит в построении модели файловой системы на базе такого “виртуального диска”. Все файловые системы предполагают разбиение диска на несколько отдельных служебных областей, название и конкретная структура которых могут существенно отличаться для различных операционных и файловых систем. Однако в любом случае эти области содержат описание общей структуры диска, информацию, необходимую для доступа к файлам, и собственно блоки файлов. Всюду далее мы будем считать, что блоки всех файлов и пустые блоки расположены в едином массиве, т.е. набор блоков отдельного файла определяется набором индексов в этом массиве. С каждым файлом связывается его имя, которое и служит для пользователя идентификатором этого файла.

Все приведенные ниже задачи в простейшем варианте предполагают моделирование основных функций файловой системы:


```
int  CreateFile (char *filename); // создать файл
int  DelFile    (char *filename); // удалить файл
int  AddBlock   (char *filename); // добавить блок в конец файла
int  FreeBlock  (char *filename); // удалить последний блок файла
int  CatFile    (char *filename); // распечатать номера блоков,
                                  //   входящих в файл
int  RenameFile (char *oldname, char *newname);  // переименовать
int  ListDir  ();   // напечатать список имен существующих файлов
```

Возвращаемое значение этих функций — код успешности выполнения указанной операции.

Дополнительно можно предложить реализацию функций

```
unsigned long  WriteFile (char *filename, unsigned long offset,
                          void *src, unsigned long size);
unsigned long  ReadFile  (char *filename, unsigned long offset,
                          void *dst, unsigned long size);
```

выполняющих запись или чтение `size` байтов с позиции файла, определяемой смещением `offset` байтов от его начала. Эти функции возвращают количество реально записанных или прочитанных байтов.

4.22. Реализуйте модельную файловую систему на базе списков файловых блоков и битового множества свободных блоков.

Идеи реализации. Выделим на диске четыре области: Form — описание формата диска, Bit — битовое множество свободных блоков, Dir — каталог файлов, Data — массив блоков файлов.

Содержимое области Form может быть представлено следующими значениями:

```
unsigned long DiskSise;    // общий размер диска в байтах
unsigned long FormSize, BitSize; // размеры его частей.
unsigned long DirSise, DataSise;
unsigned long NDataBlocks;  // Количество блоков в области Data
unsigned long BlockLength   // Длина одного блока в байтах
```

Область Bit служит для идентификации свободных и занятых блоков в области Data. Она представляет собой битовое множество.

Область Dir является каталогом файлов и содержит массив из `NDataBlocks` структур `DirEntry`:

```
typedef struct {
        char Filename[256];  // имя файла
        unsigned int Lenth;  // длина в байтах
        unsigned int FirstBlock;  // номер первого блока
} DirEntry;
```

Область Data представляет собой массив из `NDataBlocks` структур следующего вида:

```
typedef struct {
        char     data [BlockLength];
        unsigned int next;
} DataBlock;
```

Блоки одного файла завязаны в однонаправленный список по полю `next`, т.е. в переменной `next` хранится номер следующего блока файла (для последнего блока — -1).

4.23. Измените реализацию файловой системы из предыдущей задачи, заменив битовое множество свободных блоков на список свободных блоков.

Идеи реализации. Свяжем все свободные блоки в однонаправленный список. Таким образом, мы получим "файл", состоящий из свободных блоков. Сопоставим этому файлу отдельную запись `DirEntry`, которую поместим первой в области Dir. Захват и освобождение блоков эффективно выполняется в начале этого "файла" (см. задачу XX.XX)

4.24. Измените реализацию файловой системы из задачи XX.XX(4.22), выделив ссылки между блоками файлов в отдельную область диска.

Идеи реализации. Выделим на диске четыре области: Form — описание формата диска, Fat — ссылки между блоками файлов, Dir — каталог файлов, Data — массив блоков файлов.

Содержимое области Form может быть представлено следующими значениями:

```
unsigned long DiskSise;    // общий размер диска в байтах
unsigned long FormSize, FatSize; // размеры его частей.
unsigned long DirSise, DataSise;
unsigned long NDataBlocks;  // Количество блоков в области Data
unsigned long BlockLength   // Длина одного блока в байтах
```


Область Fat представляет собой массив чисел типа `unsigned long` — ссылок между блоками файлов, т.е. если за `k`-м блоком файла следует `m`-й блок, то значение `Fat[k]` равно `m`. Если `k` — последний блок файла, то `Fat[k]` равно -1, если `k`-й блок свободен, то `Fat[k]` равно 0. Поиск свободного блока при выполнении операции добавления нового блока к файлу выполняется последовательным просмотром области Fat.

Область Dir имеет тот же вид, что и в задаче XX.XX(4.22).

Область Data теперь является просто массивом блоков длины `BlockLength` каждый.

4.25. Модифицируйте реализацию предыдущей задачи, включив все свободные блоки в отдельный "файл" (как в задаче XX.XX(4.23)).

Файловые системы, основывающиеся на поддержке списков блоков файла, могут привести к значительной вычислительной работе, если необходимо получить доступ к последним блокам файла (требуется последовательно просмотреть весь список блоков файла от начала до требуемого блока). В реальных файловых системах это может привести к большому количеству дисковых операций чтения, что снижает эффективность работы.

Рассмотрим другой способ задания множества блоков файла. Будем считать, что область Dir состоит из двух частей. В первой части для каждого файла хранится его имя и дескриптор — некоторое уникальное целое число, называемое индекс файла (или inode). Вторая часть содержит массив структур следующего вида:

```
typedef struct {
        int  type;     // тип файла
        unsigned long Length;        // длина файла в байтах
        unsigned long DirectRef[10]; // номера первых
                                     // десяти блоков файла
        unsigned long IndirectRef[3]; // косвенные ссылки
                                      // на остальные блоки файла
} INode;
```

Информация структуры `INode` позволяет напрямую обращаться к первым десяти блокам файла. В `IndirectRef[0]` хранится номер блока с номерами следующих блоков данных, т.е. в его первых 4-х байтах хранится номер 11-го блока, в следующих — 12-го и т.д. В `IndirectRef[1]` хранится номер блока с номерами блоков в которых хранятся номера

блоков данных. С помощью `IndirectRef[2]` поддерживается третий уровень косвенных ссылок. Таким образом, `IndirectRef[k]` можно рассматривать как корень сильно ветвящегося дерева глубины `k+1`, в концевых вершинах которого расположены номера блоков файла.

Одними из основных процедур при реализации файловой системы являются захват свободного блока при увеличении длины файла и освобождение занятого блока при сокращении длины файла. Для построения этих процедур предложим следующие идеи. Для хранения номеров всех свободных блоков выделим один специальный блок (обозначим его `FBS` — free block stack) и будем брать и добавлять номера в этом блоке по стековому принципу. Если номера всех свободных блоков не помещаются в `FBS`, то дно этого стека (первый номер в блоке) есть номер блока, который содержит следующие номера свободных блоков. При этом первый номер в этом следующем блоке есть номер блока, содержащего следующие номера, либо 0, если больше свободных блоков нет. Таким образом, блоки с номерами свободных блоков образуют список.

При запросе на выделение блока, его номер берется из `FBS`. Если `FBS` содержит только один номер, то блок с этим номером переписывается в `FBS` (т.е. заполняет `FBS` номерами свободных блоков) и после этого номер свободного блока можно опять взять из `FBS`. Если при освобождении блока с номером k оказывается, что `FBS` заполнен до конца, то содержимое `FBS` переписывается в блок с номером k, первый элемент в стеке `FBS` полагается равным k, и количество элементов в `FBS` полагается равным 1.

4.26. Реализуйте модельную файловую систему на основе описанных выше подходов.

Идеи реализации. Выделим на диске следующие области: Superblock — описание формата диска, NameDir — каталог имен файлов, INode — массив структур типа `INode`, Data — массив файловых и служебных блоков.

Область Superblock должна содержать данные о размерах всех служебных областей, количестве блоков, размере блока, текущем и максимально возможном количестве файлов, а также номер выделенного блока `FSB`.

Область NameDir представляет собой массив структур следующего вида

```
typedef struct {
```



```
        char filename[256];  // имя файла
        unsigned int inode;  // индекс файла
}  NameDirEntry;
```

Область INode представляет собой массив структур типа `INode`, при этом значение поля `type`, равное 1, означает, что данная структура относится к существующему файлу, а значение 0 означает, что данная структура свободна и может быть использована для создания нового файла.

Отметим, что раздельное хранение имени файла и ссылок на его блоки позволяет иметь несколько имен для одного и того же файла, так как различные структуры `NameDirEntry` могут иметь одно и то же значение поля `inode`.

4.27. Измените реализацию файловой системы из предыдущей задачи так, чтобы каталог имен NameDir не являлся выделенной областью диска, а был специальным файлом. При этом его `INode` может быть размещена первой в области INode.

4.28. Модифицируйте реализации модельных файловых систем так, чтобы при создании каждому файлу можно было установить права доступа на чтение и запись: ReadOnly — файл можно только читать, WriteOnly — в файл можно только писать, NoDelete — файл нельзя удалить. Эти права доступа должны проверяться файловой системой при выполнении файловых операций внешним пользователем. Предусмотрите функции смены прав доступа к файлу.

5. Словари, базы данных

Основой алгоритмов решения задач этого раздела является прямой доступ к записям, хранящимся в (двоичном) файле в соответствии с правилами некоторого формата. Таким образом, основной задачей реализации является поддержание целостности структуры файла (т.е. формата) и эффективное определение позиции требуемой записи в файле. Достаточно простым примером подобных задач являются компьютерные словари, в которых происходит поиск и выдача словарной статьи по введенному ключевому слову. Более сложный пример дают базы данных, связывающие файлы с разнообразными записями в комплексную структуру, обеспечивающую работу с данными на основе динамически

формируемых запросов. Мы начнем с обсуждения нескольких простых задач.

5.1. Толковый (двуязычный) словарь

Рассмотрим вспомогательную задачу.

5.1. Реализуйте множество слов (текстовых строк) с возможностью быстрого поиска, добавления, удаления слов. Продумайте вопросы оптимизации алгоритмов и представлений при одном или нескольких из следующих дополнительных предположений:

а) известна достоверная оценка максимального количества слов в множестве, реальный объем множества близок к этой оценке;

б) количество слов, с которыми придется работать, заранее неизвестно;

в) в процессе работы не требуется удалять слова, они только накапливаются;

г) скорость поиска гораздо важнее занимаемой памяти;

д) требуется минимизировать память, возможно, за счет скорости работы.

Идеи реализации. Во-первых, нужно решить как размещать в памяти сами слова. для этого можно использовать функцию `malloc` или некоторую контейнерную реализацию (см. §4). Во-вторых, указатели на размещенные слова нужно поместить в некоторую структуру данных (список, дерево, хеш-множество и т.п.). В качестве возможных реализаций можно предложить:

а) упорядоченный массив указателей с бинарным поиском и вставкой;

б) упорядоченный динамический массив указателей с бинарным поиском и вставкой;

в) дерево поиска;

г) сбалансированное дерево поиска;

д) хеш-множество по методу списков;

е) хеш-множество по методу проб;

ж) дерево символов (см. задачу XX.XX).

Для каждой из реализаций оцените (теоретически и экспериментально) трудоемкость операций поиска, добавления, удаления. Оцените процент накладных расходов памяти относительно полезной загрузки памяти (т.е. суммарного количества байт-символов во всех словах).

Теперь перейдем к обсуждению требований к реализации словаря.

Во-первых, следует разработать и зафиксировать формат словарных файлов с тем, чтобы программа могла работать с разными словарями при условии, что словарные файлы подчинены одному и тому же формату.

Во-вторых, нужно принять решение о том, какие данные в процессе работы должны размещаться в оперативной памяти, а какие будут по мере необходимости читаться с диска.

В третьих, как организовать работу для обеспечения наибольшей эффективности при заданных ограничениях на память.

В четвертых, как должен выглядеть пользовательский интерфейс программы, кем и как должны формироваться или модифицироваться словарные файлы.

Словарные файлы. Каждый элемент толкового (или двуязычного) словаря состоит из двух частей — ключевого слова и словарной статьи. Формат словарного файла должен обеспечивать программе удобный доступ и к словам (для поиска), и к статьям (для их выдачи пользователю). Примем следующие решения.

Словарный файл имеет следующий формат:

```
<заголовок>
<блок слов>
<блок статей>
<блок слов>
<блок статей>
.............
.............
```

Пусть заголовок имеет длину 64 байта и содержит данные о количественных характеристиках словаря:

смещение от начала	длина байт	содержимое
0	8	идентификатор версии словаря
8	4	количество блоков слов/статей
12	4	длина одного блока слов в байтах
16	4	смещение начала первого блока слов
20	4	общее количество слов/статей
24	4	максимальная длина словарной статьи
28	4	максимальное количество слов в одном блоке

```
32          4       количество удаленных слов
36          4       количество удаленных статей
40         24       резерв (заполнение нулями)
```

Замечание. Добавление новых слов и статей легко производить в конце словарного файла. Удаление слов уже не так просто, поэтому в процессе работы целесообразно удаляемые слова только помечать, а чистку и перестройку словарного файла выполнять отдельной программой. В связи с этим мы добавили в заголовок соответствующие поля.

Для блока слов примем следующий формат:

```
<заголовок>
<словарная ссылка>
<словарная ссылка>
 . . . . . .
```

где заголовок имеет длину 16 байт и содержит

```
4 байта  -    признак блока слов (символы "blk#");
4 байта  -    абсолютное смещение от начала файла для
              следующего блока слов (в последнем
              блоке записывается нулевое смещение);
4 байта  -    количество слов в данном блоке;
4 байта  -    резерв (заполнение нулями),
```

словарная ссылка состоит из четырехбайтового поля задающего смещение словарной статьи данного слова от начала файла (для удаленного слова это 0), за которым идет само словарное слово с завершающим нулем.

Блок статей состоит из последовательности словарных статей. При этом каждая словарная статья начинается с четырехбайтового поля — длины данной словарной статьи, далее идет словарная статья с завершающим нулем. Первая строка статьи есть словарное слово, последующие строки — толкование или перевод. Для удаленной статьи первый байт словарного слова есть 0.

Таким образом, считав блок слов в память, мы имеем возможность считать и соответствующие словарные статьи, поскольку нам известны их смещения относительно начала словарного файла.

Внутренее представление словаря. Прежде всего нам понадобится структура


```
typedef struct {
        char *word;           /* указатель на слово в памяти */
        unsigned long offset; /* смещение словарной статьи в файле *
} DictNode;
```

Теперь, читая блоки слов из файла, мы можем заполнить подобные структуры и организовать из в виде упорядоченного массива, дерева и т.д. для удобного и быстрого поиска. Найдя нужное слово, далее по смещению `offset` можно прочитать из файла словарную статью.

Экономия памяти. Нетрудно подсчитать, что для большого словаря (100 тыс слов) хранение всех слов в памяти может потребовать 1–2 мегабайта. Если такой памяти в нашем распоряжении нет, придется разбивать доступ к словам на несколько этапов. Одно из возможных решений: считаем, что слова в словарном файле упорядочены по алфавиту; размещаем в памяти множество, содержащее первые слова из каждого блока слов вместе со смещениями этих блоков; по этой информации определяем блок, в котором находится искомое слово и считываем этот блок слов в память; далее отыскиваем требуемое слово и его словарную статью. При таком способе доступа для каждой словарной статьи требуется два обращения к диску (в варианте с загрузкой всех слов — только одно) и жесткое требование упорядоченности слов в словарном файле, однако, налицо существенная экономия памяти. Нетрудно заметить, что описанный способ весьма напоминает реализацию B-дерева. Действительно, B-деревья представляют собой весьма удобный и гибкий инструмент для работы с подобными словарями.

Изменение словарного файла. При добавлении, удалении слов и при изменении словарных статей возникает ряд проблем, связанных с необходимостью поддерживать соответствие между данными, хранящимися в памяти, и содержимым словарного файла. В частности, при удалении слова и статьи мы не можем свободно использовать освободившиеся части файла и, тем более, не можем сдвигать записи в файле на другие позиции поскольку это изменит смещения статей. Возможное решение — при редактировании словаря лишь помечать некоторые записи как удаленные, а чистку и упорядочивание словарного файла поручить другим программам. Подобные служебные программы могут выполнять следующие функции:

а) формирование словарного файла на основе текстового файла, размеченного некоторым удобным для человека образом; например, так:

```
word:  cat
article:  кошка
word:  dog
article:  собака
word:  man
article:  человек, мужчина
 и т.п.
```

б) формирование размеченного текстового файла (в смысле предыдущего пункта) на основе имеющегося словарного файла;

в) объединение двух словарных файлов в один новый файл с учетом упорядоченности слов по алфавиту;

г) чистка словарного файла, т.е. перестройка таким образом, чтобы в нем отсутствовали удаленные записи и “дыры”.

Замечания по программированию. Для чтения и записи данных в указанное место двоичного файла нужно использовать функции позиционирования (fseek) и чтения/записи группы байтов (fread, fwrite).

5.2. Модельная база данных

Базой данных (БД) принято называть некоторый набор информации, содержащий, кроме собственно данных, внутренние логические связи между единицами информации. В зависимости от используемой модели представления логических связей обычно различают следующие основные типы баз данных: иерархическая, сетевая и реляционная модель баз данных. Приведем основные идеи, заложенные в данные модели. Иерархическая модель наиболее хорошо представляется в виде дерева: единицы информации хранятся в вершинах, а связи в ветвях. При этом каждый элемент базы связан с некоторым количеством элементов следующего уровня иерархии и одним элементом предыдущего уровня. Данную модель удобно использовать, если информация естественно представляется в виде отношения один-ко-многим.

Для ускорения работы можно организовать дополнительные связи внутри данного дерева. При этом мы получим новую модель, которую обычно называют сетевой. Такая модель, бесспорно, представляет более широкие возможности для организации эффективной работы с БД, однако сильно усложняет процедуру формирования эффективных запросов, т.к. при этом необходимо четко представлять топологию отношений. Так

как каждый элемент базы в сетевой модели сам ссылается на некоторое количество элементов и произвольное количество элементов может ссылаться на него, то данную модель удобно использовать, если информация естественно представляется в виде отношения многие-ко-многим.

Появление реляционной модели связывают с именем Э. Кодда (E. Codd), опубликовавшего в 1970 году статью, в которой он сформулировал и доказал основные принципы построения данной модели на базе реляционной алгебры. Набор отношений (relation) между фиксированными типами элементов БД удобно представлять в виде таблицы. Строкой такой таблицы является некоторое упорядоченное множество элементов, каждый из которых представляет собой неделимую единицу информации. В этом случае говорят, что отношения в ДБ нормализованны (выделяется 5 уровней нормализованности). Считается, что логические отношения в базе данных установлены между элементами одной строки. Набор различных строк и составляет БД.

5.2.1. Модель базы данных “курс”

Рассмотрим иерархическую модель базы данных на основе информации об учащихся I-V курсов высшего учебного заведения. Пусть для каждого курса в базе хранятся

```
номера учебных групп     // целое число
```

Для каждой группы известны

```
Фамилия Имя Отчество     // текстовая строка
```

всех студентов данной группы. Про каждого студента известны

```
пол                      // один из символов м/ж
дата рождения            // текстовая строка вида ДД.ММ.ГГ
адрес                    // текстовая строка
средний балл             // вещественное число
```

Требуется реализовать систему управления базой данных, позволяющую обновлять и редактировать указанную информацию, а также выдавать справки в виде таблиц или списков на основании различных запросов.

Будем для простоты предполагать, что вся база может быть загружена в память. Поэтому в этом разделе мы не будем рассматривать форматы

файлов с данными базы, считая, что ее можно загрузить из обычного текстового файла с минимальной разметкой.

Внутреннее представление. Основой внутреннего представления будет набор структур:

Однонаправленный список по студентам по каждому курсу по каждой группе. Элементами этого списка являются структуры

```
typedef struct _Student{
        char   *name;
        char    sex;
        char    birth[9]
        char   *addres;
        double  rating;
        struct _Student *next;
}  Student;
```

Однонаправленный список по группам по каждому курсу. Элементами этого списка являются структуры

```
typedef struct _Group{
        int     number;
        struct _Group *next;
        Student * first;
}  Course;
```

Однонаправленный список по курсам. Элементами этого списка являются структуры

```
typedef struct _Course{
        int     number;
        struct _Course * next;
        Group * first;
}  Course;
```

В результате мы имеем дерево, в котором корневой уровень составляет список курсов, следующий уровень — списки групп, последний уровень — списки студентов.

Инициализирующая процедура должна читать файл с исходной информацией и настроить базу данных.

Запросы. Для запросов к базе данных будем использовать специальный язык. Опишем ключевые слова и конструкции этого языка.

Понятия.

Шаблон — текстовая строка, в которой допустимы символы “?” (любой печатный символ) и “$*$” (любая последовательность печатных символов).

Диапазон — запись числового отрезка или интервала в привычном математическом виде, например, $[1,5]$, $(-3.6,12.23)$, и т.п. символ “$*$” здесь обозначает “бесконечность”, т.е. запись $(-*,*)$ означает всю числовую ось.

Число — запись конкретного целого или вещественного числа.

Ключевые слова.

select — начало задания критериев выборки;

endselect — конец задания выборки, осуществить выборку из базы;

reselect — начало задания дополнительных критериев выборки из уже выбранных данных;

Параметры выборки задаются следующим образом:

name=*Шаблон* — выбрать фамилии, имена, отчества, подходящие под указанный шаблон;

group=*Число* или group=*Диапазон* — выбрать номер группы, равный данному числу или попадающий в данный диапазон;

Выборка остальных полей задается аналогично с ключевыми словами sex, birth, addres, rating и использованием шаблонов для текстовых полей и чисел или диапазонов для числовых полей. Если для какого-либо поля критерии не заданы, то при выборке это поле не анализируется.

Приведем несколько примеров запросов.

Выбрать всех студентов второго курса со средним баллом выше 4.5.

```
select
group=[200,299]
rating=[4.5,*)
endselect
```

Из полученной выборки дополнительно выбрать мужчин 1980 года рождения, фамилии которых начинаются на “И”.

```
reselect
sex="м"
name="И*"
birth="*.80"
endselect
```

5.2. Добавьте к языку запросов возможность управления формой выдачи результатов:

format — начало задания формата выдачи выборки;

endformat — конец задания формата выдачи;

print — вывести выборку в указанном формате;

Формат выдачи определяется указанием полей (тех же ключевых слов , что и при задании выборки), которые нужно вывести на печать, через запятую в том порядке, в котором это нужно. Например, по запросу

```
format
name,rating,addres
endformat
print
```

должен получиться список студентов из сделанной выборки, содержащий фамилии, средние баллы и адреса.

5.3. Добавьте к языку запросов возможность объединения нескольких значений в правой части критериев запроса (логическая операция “или”). Например,

```
group=201,[207,208],412
```

означает выборку из групп 201,207,208,412

Очевидно, что запрос типа “Найти всех студентов 201 группы, у которых средний балл выше 4.5” не потребует много времени, однако, поиск информации типа “Найти всех студентов, родившихся 31 декабря” потребует обхода всего дерева.

Для ускорения работы можно организовать дополнительные связи внутри данного дерева. Например, создав бинарное дерево поиска по фамилиям студентов со всего курса, L2-список указателей на упорядоченные результаты по графе средний балл для студентов со всего курса и т.д. Подобные модификации превратят нашу базу в сетевую модель.

5.2.2. Модель базы данных “студент”

Рассмотрим сетевую модель базы данных на основе информации об учащихся высшего учебного заведения. Пусть для каждого студента в нашем распоряжении имеются следующие данные:


```
Фамилия Имя Отчество     // текстовая строка
номер учебной группы     // целое число
пол                      // один из символов м/ж
дата рождения            // текстовая строка вида ДД.ММ.ГГ
адрес                    // текстовая строка
средний балл             // вещественное число
```

Требуется реализовать базу данных, позволяющую обновлять и редактировать указанную информацию, а также выдавать справки в виде таблиц или списков на основании различных запросов.

Будем для простоты предполагать, что вся база может быть загружена в память. Поэтому в этом разделе мы не будем рассматривать форматы файлов с данными базы, считая, что ее можно загрузить из обычного текстового файла с минимальной разметкой.

Внутреннее представление. Заметим, что вся рассматриваемая информация о студенте является сугубо индивидуальной за исключением номера группы и пола. Одинаковый номер группы и одинаковый пол имеют много студентов. Но в данном случае эти данные занимают мало места, поэтому мы можем пойти на повторное указание этой информации у каждого студента, а не группировать каким либо образом студентов одной группы или одного пола. Таким образом, основой внутреннего представления будет структура

```
typedef struct {
        char   *name;
        int     group;
        char    sex;
        char    birth[9]
        char   *addres;
        double  rating;
}  Info;
```

Наша главная задача — уменьшить время обработки запросов. Поскольку наиболее частыми, повидимому, бывают запросы по конкретному студенту или по студентам конкретной группы (курса), то поступим так. Свяжем структуры типа `Info` в бинарное дерево поиска по фамилиям студентов и дополнительно включим в хеш-множество по методу многих списков, где каждый список соответствует одной группе. Тогда поиск студента с данной фамилией удобно производить по дереву, а обработку группы — через список хеш-множества. Мы приходим к следующей структуре

```
typedef struct _Student {
        Info    data;
        struct _Student *left,*right;  /* указатели дерева */
        struct _Student *next,*prev;   /* указатели списка */
}  Student;
```

Заметим, что найдя данные о студенте по дереву, мы также легко получаем данные о всей его группе за счет двунаправленного списка.

Инициализирующая процедура должна читать файл с исходной информацией и включать прочитанные данные в базу с правильной расстановкой указателей дерева и списков хеш-множества.

Запросы. Для запросов к базе данных можно использовать рассмотренный в предыдущем разделе язык запросов.

5.2.3. Модель базы данных “расписание”

Учебное расписание представляет собой прямоугольную таблицу, в которой столбцы соответствуют учебным группам, а строки — дню и времени проведения занятий в рамках одной недели. В каждую клетку этой таблицы заносится наименование учебной дисциплины, фамилия преподавателя и номер аудитории. Требуется разработать программу, позволяющую редактировать расписание и выдавать различные справки о занятости того или иного преподавателя, аудитории, расписании отдельно взятой группы и пр.

Внутренее представление данных. Данные, используемые в базе, представим в виде следующих структур: *список преподавателей*, *список аудиторий*, *список учебных дисциплин*, *массив номеров групп*, *массив времени занятий*, *матрица расписания*. Списки и массивы используются для хранения информации об указанных объектах, а матрица — для связывания этой информации в единое целое (что и составляет расписание). Мы выбрали массивы для хранения номеров групп и времени начала занятий поскольку эти данные обычно не изменяются или изменяются очень редко, в то время как добавление и замена преподавателей или учебных дисциплин — это обычное явление. Минимальный набор данных, записанных в элементах этих структур может быть представлен следующими типами данных:

```
typedef struct _tli {
        char *name;
```



```
        struct _tli *next;
} TeacherListItem;
typedef struct _rli {
        int   number;
        struct _rli *next;
} RoomListItem;
typedef struct _sli {
        char *subject;
        struct _sli *next;
} SubjectListItem;
typedef {
        int    number;
} GroupItem;
typedef {
        int    day, hour, min;
} TimeItem;
```

Таким образом, мы рассматриваем однонаправленные списки. При желании в эти элементы можно добавить дополнительную информацию, например, должность преподавателя, количество мест в аудитории и т.д. Матрицу расписания можно оформить как матрицу элементов типа

```
typedef struct  {
        TeacherListItem *teacher;
        RoomListItem    *room;
        SubjectListItem *subject;
} MatrixItem;
```

при этом элемент этой матрицы с индексами i, j будет соответствовать i-й группе в массиве групп и j-му времени начала занятий в массиве времен.

Инициализация базы. В качестве начальных данных для заполнения базы можно взять текстовый файл, в котором последовательно перечислены необходимые данные о преподавателях, дисциплинах, аудиториях, группах и временах. На основании этой информации заполняются соответствующие списки и массивы. Далее могут идти фрагменты расписания, на основании которых заполняются некоторые элементы матрицы расписания. При инициализации расписания следует предусмотреть проверку корректности вводимых данных (существуют ли указанные имена,

названия, номера в уже заполненных списках преподавателей дисциплин, групп и т.д.)

Работа с базой. Основными функциями базы являются редактирование расписания и получения справок. Эти действия целесообразно производить в интерактивном режиме. Например, можно реализовать простейшее меню с такими пунктами:

- добавить (преподавателя, дисциплину, группу, аудиторию, время);
- удалить (преподавателя, дисциплину, группу, аудиторию, время);
- редактировать (преподавателя, дисциплину, группу, аудиторию, время);
- редактировать клетку расписания;
- вывести расписание группы;
- вывести расписание преподавателя;
- вывести занятость аудитории;
- вывести аудитории, занятые в данное время;
- вывести свободные аудитории на заданное время;
- вывести состав преподавателей данной дисциплины;

и т.п.

5.4. Некоторые учебные занятия проводятся один раз в две недели. Модифицируйте этот проект, чтобы поддерживать в расписании и такие занятия.

5.5. При указанном представлении данных составление расписания отдельного преподавателя приводит к просмотру всех элементов матрицы. Организуйте дополнительную связь всех элементов матрицы, относящихся к одному преподавателю, в однонаправленный список, указатель на начало которого хранится в элементе списка преподавателей. Это можно сделать, например, так

```
struct _matrix;
typedef struct _tli {
        char *name;
        struct _tli *next;
        struct _matrix  *start;
} TeacherListItem;
typedef struct _matrix {
        TeacherListItem *teacher;
        RoomListItem    *room;
```



```
        SubjectListItem *subject;
        struct _matrix  *next;
} MatrixItem;
```

Рассмотрим еще один подход к построению базы данных “расписание”. Учебное расписание в стандартном виде представляет собой прямоугольную таблицу, в каждую клетку этой таблицы заносится наименование учебной дисциплины, фамилия преподавателя и номер аудитории. Однако, в явном виде матрица расписания не является нормализованной. Представим ее в виде `Time-table` таблицы со следующими столбцами: `[room, time, name, discipline, group]`. Так как каждому столбцу таблицы присвоено уникальное имя, то столбцы как единое целое можно переставлять. Информация при этом разрушаться не будет. Строки такой таблицы принято называть записями, а столбцы полями. Чтобы можно было идентифицировать записи БД необходимо чтобы любые две записи отличались хотя бы по одному полю. То есть вся запись как единое целое должна быть уникальна. Таким полем может быть дополнительное поле number, содержащее порядковый номер записи внутри данной БД.

Представление данных, формат dbf-файла. Таблицу с данными будем хранить в dbf-формате, работу с которым поддерживают большинство субд. Содержимое dbf-файла состоит из заголовочной записи, содержащей заголовок и описание структуры полей в записях, и данных:

```
|Заголовок|Описание полей|ЗаписьЗаписьЗапись...
   |Заголовочная запись|
```

Структура заголовочной записи:

Смещение	Длина в байтах	Примечание
0	1	Номер версии
1	3	Дата последнего изменения
4	4	Количество записей в файле
8	2	Смещение, с которого начинаются записи
10	2	Длина записи в байтах
12	20	Резервные байты
32	32*N	По 32 байта на описание каждого поля записи
32+32*N+1	1	Признак конца заголовка (Odh)

Первый байт записи предназначен для идентификации dbf-файла. Если значение байта равно 02h, 03h - то файл без memo-полей, если первый байт имеет значение 83h(f5h), то dbf-файлу ставится в соответствие dbt(fpt)-файл со значениями начала memo-полей. В противном случае файл не рассматривается как файл dbf, и работа с ним запрещается.

Следующее поле длиной 3 байта содержит в двоичном коде дату последней корректировки файла. Начиная с байта со смещением 4, в поле длиной 4 байта хранится значение количества записей dbf-файла, в том числе и помеченных для удаления, которое представлено в виде числа без знака длиной 32 бита. Используется традиционный для процессоров семейства Intel способ представления числа – младший байт всегда хранится в ячейке с младшим адресом. Следующее поле используется для хранения 2-байтового числа без знака, указывающего длину заголовочной записи в байтах. Эта информация необходима, поскольку в dbf-файле может содержаться описание различного количества полей (см. ниже). В 2-байтовое поле со смещением 10 заносится длина записи. Значение хранимого в данном поле числа всегда на единицу больше суммы длин всех полей, поскольку в начале каждой записи имеется еще один байт для индикации удаления записи. Остальные 20 байтов, начиная со смещения 12 зарезервированы для внутреннего использования. После заголовка в заголовочной записи размещено описание полей. Для описания каждого поля базы данных отведено по 32 байта. Формат описания представлен в таблице:

Смещение	Длина в байтах	Примечание
0	11	Имя поля (строка ascii)
11	1	Тип поля в ascii-кодах (C, N, L, D, M)
12	4	Смещение в памяти до данного поля в первой записи от адреса начала записей
16	1	Длина поля в байтах
17	1	Число знаков после десятичной точки в байтах
18	2	Зарезервированы для многопользовательских систем
20	1	ID для рабочей области
21	2	Зарезервированы для многопользовательских систем
23	1	Используется командой SET FIELDS
24	8	Резервные

Первые 11 байтов каждого описания содержат имя поля в виде ascii-строки. Цепочка символов замыкается нулевым байтом. Если имя поля содержит менее 11 символов, оставшиеся байты заполняются нулевыми значениями (00h). Следующий байт содержит тип поля в ascii-кодах. Далее описаны допустимые значения полей каждого типа:

Обозначение	Тип поля	Допустимые значения
C	Символьный	ascii-символы
N	Числовой	- 0,1,...,9
L	Логический	TtFf
D	Дата	ггммдд
M	memo-поле	Идентификатор записи в dbt-файле

Вслед за типом поля указывается его смещение в памяти (4 байта, начиная с 12-го). Для первого поля это 1, для второго (1+длина первого поля) и т.д.

Длина поля хранится в байте со смещением 16. Таким образом, длина поля не может превышать 256 символов. В действительности такая длина допускается только для полей символьного типа. Для числовых полей указывается количество десятичных знаков, включая десятичную точку. Длина поля памяти всегда равна 10 байтам, поскольку в этом поле хранится номер блока dbt-файла со значением соответствующего поля памяти. Фиксированную длину имеют поля логического типа (1 байт) и поля типа "дата"(8 байтов).

Следующий байт указывает количество знаков после десятичной точки для полей числового типа или имеет значение 00h для полей других типов. Количество знаков после точки всегда меньше длины поля.

Остальные 14 байтов предназначены для внутренних целей. Итак, в заголовочной записи dbf-файла описание каждого поля занимает 32 байта. Конец описания полей помечается кодом 0dh.

После заголовочной записи идут записи данных. Значение длины записи указывается в заголовке dbf-файла. Оно равняется сумме длин полей плюс единица, так как, в начале каждой записи содержится байт, предназначенный для маркирования удаления записи. Запись хранится в виде последовательности ascii-кодов без разделяющих символов, благодаря чему импорт/экспорт данных осуществляется достаточно просто. Данные в отдельных символьных полях представляются в виде последовательности ascii-кодов. Если последовательность короче длины поля, оставшиеся

байты заполняются пробелами (код 20h). Целые числа представляются в десятичной системе счисления и хранятся в символьном виде (ascii). Допустимое количество знаков числа указано в описании поля. Для чисел, имеющих дробную часть, задается количество знаков после десятичной точки, причем сама точка также учитывается при определении длины поля. Если количество знаков в числе меньше длины поля, то ведущие позиции заполняются пробелами (например, "999.99"). Значения логических полей представляются символами "f"или "t". Дата записывается в соответствующие поля как последовательность символов ascii в формате "ггммдд". В memo-поле длиной 10 байтов хранится номер соответствующего блока в dbt-файле. Начальные байты при необходимости заполняются пробелами. Если значение поля памяти состоит только из пробелов, значит для него не существует соответствующей текстовой записи в dbf-файле. Таким образом, все поля, независимо от их типа, могут обрабатываться как представленные в кодах ascii тексты. Конец действительной области данных помечается кодом 1ah (eof). Следует учитывать, что положением данной отметки управляет не операционная система. Помеченные для удаления записи остаются в файле. Можно логически удалить все записи файла. При большом количестве логически удаленных записей доступ к действительным записям при последовательном просмотре базы данных будет происходить медленно, поскольку приходится просматривать все записи как помеченные, так и не помеченные для удаления). Удаленные записи размещаются после метки конца файла и могут быть восстановлены соответствующими программными средствами.

При работе с dbf-файлами удобно считать "Заголовок"и "Описание полей"в следующие структуры:

```
typedef struct{
 char dbf_id;                // (0x) 03-без  memo-полей,
                                    83- с  memo-полями (в .dbt);
 char last_update[3];        // дата; две цифры - год,
                                       две - месяц, две - день;
 int last_rec;               // количество записей в базе;
 unsigned short data_offset; // смещение, с которого
                                          начинаются записи;
 unsigned short rec_size;    // размер каждой записи = сумма полей
                                             + 1 байт;
```



```
 char filler[20];         //  пустые (дополняют структуру
                                                  до 32 байт);
} DBF_HEAD;

typedef struct{
   char field_name[11]; // название поля -- 10 символов;
   char field_type;     // тип поля: C(0x43),
                        // D (0x44), L(0x4C), M (0x4D), N(0x4E);
   char dymmy[4];      // Смещение
   union  {
   unsigned short  char_len; // когда тип поля не N; длина
   struct        // когда тип N
   {  char len;   // длина поля;
      char dec;   // число десятичных знаков;
   } num_size;

         } len_info;
  char filler[14];       // дополняет до  32 байт;
}FIELD_REC;
```

Следует отметить, что предложенное описание структур будет правильным при sizeof(int)=4, sizeof(short int)=2. Иначе размер структур будет отличаться от 32 байт.

5.6. Написать программу, распечатывающую содержимое dbf-файла в текстовый файл в виде таблицы.

5.7. Выбрать из dbf-файла записи, удовлетворяющие по заданному полю некоторому шаблону (см. задачу стр. 104 XX.XX) и создать из них новый dbf-файл.

Несложно понять, что реализация запроса: Найти все записи, удовлетворяющие по полю `name` (фамилия преподавателя) строке "Ефремов потребует просмотра содержимого данного поля во всех записях. Если же поддерживать упорядоченность по данному полю, переставив строки таблицы, то аналогичная проблема возникает с другим полем. Поэтому для осуществления быстрого поиска по некоторому полю предусматривается создание так-называемого индексного файла. В этом файле (его имя совпадает с именем поля) хранится упорядоченная информация для всех записей по содержимому интересующего нас поля и ссылка, позволяющая определить какой именно записи принадлежит данное

поле. Обычно данная информация представлена в виде B-дерева (либо B*-дерева). Допускается создание мультииндексного файла, в котором хранится информация для быстрого поиска по двум и более полям.

5.8. Написать программу, создающую индексные файлы для базы данных, хранящейся в dbf-формате. С ее помощью модифицировать решение предыдущей задачи.

Инициализация базы.

5.9. Написать программу заполнения базы данных (создания dbf-файла) из текстового файла, содержащего в виде таблицы необходимые данные о преподавателях, дисциплинах, аудиториях, группах и временах. Формат txt-файла фиксируется автором программы. Для упрощения работы в файле возможно наличие *заголовочной строки*, описывающей дальнейший формат таблицы.

5.10. Написать программу, позволяющую создавать новую базу данных и работать с имеющейся в интерактивном режиме. Реализовать стандартные запросы языка типа miniSQL: CREATE, DROP, DELETE, UPDATE, а так же операторы SAVE и OPEN.

```
CREATE имя_базы_данных
    ( имя_первого_столбца тип_первого_столбца размер_типа,
    имя_второго_столбца тип_вторго_столбца  размер_типа,
    ..................
    имя_последнего_столбца тип_последнего_столбца размер_типа);

DROP имя_столбца FROM имя_базы_данных;


INSET INTO имя_базы_данных ( имя_столбца, ... , имя_столбца)
      VALUES          ( значение, ... , значение);

DELETE FROM имя_базы_данных WHERE имя_столбца OPERATOR значение
  [AND|OR имя_столбца OPERATOR значение];

UPDATE имя_базы_данных SET имя_столбца = значение
   [ имя_столбца=значение ], WHERE имя_столбца OPERATOR значение
 [AND|OR имя_столбца OPERATOR значение]

OPEN имя_базы_данных
SAVE имя_базы_данных [AS новое_имя ]
```


Где `OPERATOR` может принимать значения `<, >, =, <=, >=, <>`. При этом допустимо использование шаблонов `*` и `?`.

Работа с базой.

5.11. Для запросов к базе данных реализовать оператор `SELECT`.

```
SELECT  имя_столбца  [имя_столбца] FROM имя_базы_данных
[ WHERE имя_столбца OPERATOR значение
     [ AND|OR имя_столбца OPERATOR значение ] ]
 [ ORDE BY имя_столбца [DESC] ]

SELECT name, discipline, group FROM Time-table
  WHERE name = 'Efremoff' AND group = 2*  OR group=3*
    ORDE BY group DESC
```

Ключ `DESC` означает сортировку по убыванию.

В результате работы оператора `select` создается новая база данных с именем `Current` и ее содержимое отображается на экране.

5.3. Модельная справочная система

Требуется разработать и реализовать модельную гипертекстовую систему для выдачи справочной информации. В каждый момент времени при работе программы на экране отображается текущая информационная статья и список ключевых фраз, из которого пользователь может выбирать для просмотра последующие статьи.

Решение этой задачи разбивается на три составные части: программа подготовки рабочих файлов, программа поиска требуемых информационных записей, интерфейсная часть.

Поскольку реализация интерфейсной части сильно зависит от возможностей и квалификации программиста, мы будем считать, что выбор нужной ключевой фразы производится из некоторого меню, исходными данными для которого является список ключевых фраз и сопутствующая информация, необходимая для поиска. В остальном мы не будем касаться этой части проекта.

Рабочими данными для системы должен являться специально подготовленный файл, формат которого обеспечивает эффективный доступ к требуемой информации. Рассмотрим возможный формат рабочего файла с данными. Файл содержит заголовок и последовательность записей.

Формат заголовка:

```
смещение    длина   содержимое
от начала   байт
   0         8   идентификатор рабочего файла
   8         4   длина заголовка
  12         4   количество записей
  16         4   максимальная длина записи
  20         4   максимальное количество ключевых ссылок в записи
  24         8   начало стартовой записи
  32         4   длина  стартовой записи
  36        28   резерв
```

Формат отдельной записи:

```
смещение    длина   содержимое
от начала   байт
записи
   0         4        признак начала записи
   4         4        длина данной записи
   8         4        количество ссылок в записи  (n)
  12        8n        список смещений статей
 ...       ...        список ключевых фраз
 ...         4        признак начала статьи
 ...       ...        информационная статья
```

Признаки начала записи и статьи введены для обеспечения дополнительного контроля при чтении записи во время работы (т.е. действительно ли по указанному смещению находится запись или статья). В качестве таких признаков можно взять некоторый набор символов, которые вряд ли встретятся в другом месте файла.

Прочитав такую запись в память, мы имеем все необходимое для доступа к последующим статьям, на которые указывают ключевые фразы (а именно, смещения, определяющие местоположение соответствующих записей).

Следующий этап — подготовка рабочего файла. Рабочий файл практически невозможно создать вручную, нужна специальная программа для его подготовки. Исходные данные для такой программы могут быть сформированы в виде текстового файла с дополнительной разметкой. Например, в справочной системе по работе с некоторым программным продуктом, подобный файл может содержать последовательность записей типа


```
keyword:
   начало работы
   выполнение задания
reference:
   верхнее меню
   сохранение результата
   окно редактирования
   завершение работы
text:
   Для начала работы нажмите кнопку START в верхнем меню.
Выполнив задание, сохраните результат в файле и
закройте окно редактирования. После этого вы можете
закончить работу.
```

Здесь слово `keyword` обозначает начало списка ключевых фраз для данной статьи, слово `reference` — начало списка ключевых фраз для последующих статей, `text` — начало текста статьи.

Сложность подготовки состоит в том, что пока мы не сформируем целиком рабочий файл, мы не сможем определить истинные смещения информационных статей. Другая проблема — проверка корректности исходных данных — может оказаться, что некоторые статьи не имеют ссылок на себя, а некоторые ключевые фразы не имеют статей. Программа подготовки должна обнаружить подобные некорректные ситуации и выдать протокол ошибок, по которому можно подправить исходный текстовый файл с данными. При построении рабочего файла можно сначала не заполнять поля смещений статей, а вести таблицу, в которую записывать смещения статей, уже занявших свое место в файле, и фиксировать в какие места файла нужно проставить смещения тех или иных статей. После завершения этого предварительного форматирования файла, можно заполнить поля смещений на основе построенной таблицы.

Процедура поиска и выбора статей не создает особых проблем. На основании информации, прочитанной из заголовка файла, считывается и выводится стартовая статья. Далее запись для каждой очередной статьи содержит всю необходимую информацию для формирования меню ключевых фраз и доступа к последующим статьям.

5.12. Реализуйте описанный проект. Дополнительно реализуйте возможность возвращения к предыдущим статьям (стек). Добавьте в меню переход на стартовую статью и завершение работы.

5.13. Подберите наполнение для данной системы. Интересными приложениями в этом направлении являются тестирующие программы (вопрос и набор ответов), игровые программы (блуждание в лабиринте). В этих вариантах посещение каждой статьи и выбор правильного ответа дает пользователю определенный набор баллов. При достижении пользователем некоторого уровня, программа может перейти на другой набор статей.

5.4. Гипертекстовая HTML-система

Требуется реализовать программу просмотра гипертекстовых документов, записанных с использованием некоторого подмножества языка HTML (HyperText Markup Language). Этот язык представляет собой совокупность достаточно простых команд (тегов), которые вставляются в исходный текстовый файл и позволяют управлять формой вывода этого документа на экране дисплея и осуществлять переходы на различные части этого файла или к другим файлам. Сами команды языка на экране не отображаются. Таким образом, текст набранный в произвольном текстовом редакторе и сохраненный как обычный ascii-файл, становится гипертекстовым документом после добавления в него ряда команд языка HTML. Основная цель данного проекта — возможность включать в текст ссылки на другие аналогичные документы.

Для задания в тексте ссылки на другой файл или на некоторый фрагмент текущего файла используется пара команд <A> </A> (Anchor - Якорь):

```
<A HREF = "Informations.txt"> Подробная информация</A>
```

Фраза `Подробная информация` при выводе на экран выделяется каким-либо образом и, если пользователь выбирает данную ссылку, то начнет отображаться файл `Informations.txt`, расположенный в текущей директории.

Можно осуществлять переход не к началу документа, а к некоторой его части. Если в файле "document.txt"содержится строка

```
<A NAME="label1">Пункт первый.</A>
```

начиная с которой следует отображать текст, то соответствующая ссылка на данную строку из произвольного файла может выглядеть так:


```
<A HREF = "document.txt#label1"> Переход к пункту первому </
```

Для перехода внутри данного файла ссылка на метку используется без указания имени файла. Так команда

```
<A HREF="#label2">Начало сообщений</A>"
```

означает переход к строке данного файла, помеченной тегом

```
<A NAME="label2"> Сообщения.</A>
```

5.14. Реализуйте программу для просмотра гипертекстовых файлов с описанной системой команд.

Идеи реализации. В простейшем варианте можно ограничиться выводом на экран нескольких строк файла, начиная с выбранной метки. Для вывода последующих строк можно предусмотреть отдельную команду. Напомним, что сами теги гипертекстовой разметки выводить на экран не нужно. Способ выделения ссылок при выводе текста зависит от типа и режима работы монитора. Можно выделять ссылки цветом шрифта, подчеркиванием и т.п. Выбор следующей ссылки можно реализовать через некоторое меню, либо непосредственным перемещением курсора (или указателя мыши) на требуемую ссылку.

5.5. Задачи со словами

В задачах данного раздела считается, что имеется словарный файл, содержащий лексиграфически упорядоченное множество различных слов. Слова хранятся по одному в строке с первой позиции. При решении конкретной задачи, возможно, потребуется построить новый словарный файл с наиболее подходящей структурой (см., например, задачу 5.1), обеспечивающей эффективное выполнение программы.

5.15. Написать программу поиска слов в словарном файле, удовлетворяющих заданному шаблону. Шаблон – текстовая строка, в которой допустимы символы “?” (любой печатный символ), “$*$” (любая последовательность печатных символов), “[a-z,1,2,3]” (диапазон символов). Например: *а???ф* - все слова у которых встречаются буквы ’а’ и ’ф’ через три символа.

5.16. Написать программу проверки правописания слов в тексте с возможностью корректировки ошибок в диалоговом режиме. Если слово

в словарном файле отсутствует, то для него находятся все "близкие слова" – отличающиеся на одно исправление (вставку, удаление, замену одной буквы).

5.17. Написать программу, создающую матрицу кроссворда. Выбирая слова из словаря, записать их с пересечениями, так, что каждое слово имеет не менее двух пересечений и вся матрица помещается в некоторый прямоугольник $n \times m$. Можно дополнительно потребовать построение симметричного кроссворда.

5.18. Пусть имеется прямоугольная матрица размерности $m \times n$ в каждой ячейке которой хранится некоторая буква. Найти все слова, записанные в данной матрице. Слова могут начинаться с произвольной позиции и читаться по правилу вправо, вверх, вниз.

5.19. Написать игру "Слово по букве". Играющий и программа дописывают в конец текущей последовательности букв по букве, так чтобы она оставалась началом некоторого слова. Проигрывает тот, кто пишет последнюю букву слова. В качестве усложнения можно разрешить дописывать букву как в конец, так и в начало имеющегося слова.

5.20. Написать программу "Все слова". С экрана вводится некоторое множество букв (строка). Требуется напечатать все слова, которые можно составить из имеющихся букв.

Идеи реализации. Из n букв можно составить $n + n(n-1) + ... + n!$ различных "слов". Если каждое проверять на наличие его в словаре, то время работы программы при больших n будет слишком велико.

По данному словарю предлагается построить новый словарь в котором каждое слово составляется из упорядоченных по алфавиту букв соответствующего слова исходного словаря и ссылки на оригинальное слово в исходном словаре. Новый словарь в памяти хранится в виде массива списков. Внутри каждого списка слова упорядочены так, что каждое предыдущее слово содержит все буквы последующего.

5.21. Написать программу, которая находит в словаре все тройки слов: `Слово1`, `Слово2`, `Слово3` таких, что `Слово2` является концом `Слова1` и началом `Слова3`.

5.22. Пусть имеется некоторое множество слов различной длины, обладающих одинаковым окончанием. Найти это окончание, при условии, что в каждом слове известны все первые буквы.

5.23. Найти всевозможные слова, обладающих данным окончанием.

5.24. Реализовать игру "Квадрат 5*5". В начале игры на средней линии квадрата 5*5 написано некоторое слово. Программа и человек по очереди пишут букву так, чтобы при прочтении по правилам 1-4 получилось новое слово. За написанное слово, играющий получает количество очков равное количеству букв нового слова.

1. Начинать читать можно с любого места, но двигаться только вправо, вверх, вниз. 2. Одну и туже букву нельзя читать дважды. 3. Нельзя использовать как часть нового слова уже выбранные до этого слова. 4. За один ход учитывается только одно слово.

В качестве усложнения можно разрешить читать с любого места в любом направлении (в том числе и по диагонали), т.е. по цепочке которую может обойти шахматный король.

Далее будем считать, что слова словарного файла разделены на слоги и в них поставлены ударения: Прог-рам-ми'-ро-ва-ни-е. Рассмотрим задачу создания стихотворного текста. Сформулируем, что же именно мы хотим получить. Назовем контрольным рядом некоторую последовательность конечной длины двух допустимых символов **c**, **C**. Например:

cccCcccCcccC

Будем говорить, что набор слов удовлетворяет данному контрольному ряду, если каждому ударному слогу фразы соответствует символ **C**. Символы **c**, **C** соответствуют как ударным, так и безударным слогам.

5.25. Написать функцию, формирующую по данному контрольному ряду фразу из слов имеющегося словаря .

Будем говорить, что две строки рифмуются простой рифмой, если их заключительные слоги совпадают, например:

```
......  без-мя-теж-ный
......  неж-ный.
```

5.26. Написать функцию, формирующую по данному контрольному ряду и заданному последнему слогу, фразу из слов имеющегося словаря.

Строка: "При-ро-дой здесь нам суж-де-но,"контрольный ряд:

cCcCcCcC(yj)

Хотелось бы получить строку: "В Ев-ро-пу про-ру-бить ок-но."

Рассмотрим способ задания контрольных рядов. Фиксируем следующие элементарные группы.

1. Двудольная.

cC ямб, **Cc** хорей.

2. Трехдольная.

Ccc дактиль, **cCc** амфибрахий, **ccC** анапест.

3. Четырехдольная.

Cccc *ccCc* хорейные, *cCcc* *cccC* ямбические

Аналогично строятся пять пятидольных и шесть шестидольных групп.

Главным структурным признаком "стихотворных"строк является повторение (обычно 2-6 раз) элементарной группы. При этом некоторое количество первых и последних символов **c**, **C** контрольного ряда может быть отброшено. При чтении стиха в этом месте естественно возникает необходимое количество тактов паузы.

Будем говорить, что стихотворные строки образуют стихотворный текст, если они рифмуются между собой.

5.27. Написать программу, формирующую стихотворный текст из слов словаря по данному набору контрольных рядов.

```
                        Ямб:
  cCcCcCcCc(ой)                    cCcCcCcCcCcCc(мо)
  cCcCcCcC(ой)                     cCcCcCcCcCcCc(ма)
  cCcCcCcCc(ой)                    cCcCcCcCcCcC(ан)
  cCcCcCcC(ой)                     cCcCcCcCcCcC(ан)
Для берегов отчизны дальной    Среди семи холмов, где так неодолимо
Ты покидала край чужой;        Прекрасен на заре воздушный облик Рима
.......................          .......................
                    А. Пушкин                        А. Фет

                        Хорей:
  CcCcCcCcCc(та)                   CcCcCcCc(ны)
  CcCcCcCcCc(ча)                   CcCcCcC(на)
Финиша летящая минута,           Сквозь волнистые туманы
```



```
Молодость легка и горяча -           Пробирается луна
.........................            .........................
                    Б.Корнилов                          А. Пушкин

Дактиль:                              Амфибрахий:
   CccCccCccCc(вы)                    cCccCccCccC(ли)
   CccCccCccCc(вы)                    cCccCccCccC(ли)
Были и лето и осень дождливы;     В песчаных степях аравийской земли
Были потоплены пажити, нивы ...   Три гордые пальмы высоко росли
.........................            .........................
                 В. Жуковский                          М. Лермонтов

Анапест:
   ccCccCccCccC
   ccCccCccCc
Зачинается песня от древних затей
От веселых пиров и обедов
.........................
```

Более связный текст получается если при подборе слов учитывать согласование частей речи.

6. Задачи на преобразование файлов

В данном разделе содержатся примеры, которые составляют достаточно широкий класс задач обработки данных и являются составной частью многих программ или даже самостоятельными программами в реальных вычислительных системах. Речь идет о преобразовании информации, записанной в файле, из одной формы представления в другую. Часто (но не всегда) подобные программы ориентированы на обработку текстовых файлов. Примерами могут служить программы, форматирующие текстовые файлы для удобства чтения человеком, различные перекодировщики, архиваторы и пр.

Готовые исполняемые программы могут быть реализованы в двух вариантах: 1) программы берут исходный текст из стандартного ввода (stdin) и выводят результат в стандартный вывод (stdout); 2) программы получают в командной строке регулярное выражение (шаблон), задающее имя файла, и в результате получают преобразованные файлы с тем

же именем, что и исходные (по поводу разбора файловых регулярных выражений см. задачу XX.XX).

6.1. Перекодировки, фильтры, преобразования текстов

Фильтрами обычно называют программы, выполняющие преобразования последовательности байтов, поступающих со стандартного входа, (stdin) и направляющие результат в стандартный вывод (stdout). К фильтрам примыкают перекодировщики — программы, заменяющие значения байтов в входном потоке в соответствием с заданными формулами или таблицами, например, программы для преобразования кодировок представления символов русского алфавита в текстовых файлах. Фактическим стандартом для представления печатных символов латинского алфавита является кодировка ASCII. К сожалению, для представления русских букв нет единого стандарта. Наиболее часто используемые кодировки — альтернативная (???), KOI8-R, Windows-1251. Таблицы кодов ASCII, Alt, KOI8-R, Windows-1251 приведены в приложении.

6.1. Составьте программы, выполняющие преобразование текстовых файлов из одной кодировки в другую. Назовите эти программы именами win2alt, alt2koi, win2alt и т.п. (предполагается, что программа win2alt производит преобразование файла с кодировкой Windows-1251 в альтернативную кодировку, программа alt2koi — из альтернативной — в кодировку KOI8-R, и т.д.)

Идеи реализации. Для каждого преобразования следует создать массив (скажем, с именем `code`) из 256 целых чисел, где i-тый элемент содержит код i-того символа в другой кодировке. Тогда преобразование файла, определяемого указателем `in_file` и выводимого в файл по указателю `out_file`, фактически сводится к циклическому выполнению оператора

```
fputc (code[fgetc(in_file)], out_file);
```

с соответствующими проверками на окончание входного файла.

Организуйте программу следующим образом:

1) имя входного файла задается в виде параметра командной строки;

2) программа создает временный файл, в который записывается перекодированное содержимое исходного файла;

3) исходный файл удаляется;

4) выполняется переименование временного файла с именем исходного.

В результате выполнения программы получится файл с тем же именем, но другой кодировкой.

6.2. В операционной системе MS-DOS концы строк в текстовых файлах принято кодировать двумя байтами, имеющими шестнадцатеричные значения 0D, 0A. В операционной системе UNIX концы строк в текстовых файлах кодируются одним байтом 0A. Напишите программы преобразования текстовых файлов из одной формы представления в другую, назовите эти программы dos2unix и unix2dos и организуйте их так, как это было предложено в замечании к предыдущей задаче.

6.3. Напишите программу, которая заменяет каждый символ табуляции `'\t'` в текстовом файле на 8 пробелов. Напишите программу, выполняющую обратное преобразование.

6.4. Напишите программу, которая заменяет всюду в файле один заданный набор символов на другой заданный набор символов (возможно другой длины).

6.5. Напишите программу, которая отфильтровывает из входной последовательности символов (т.е. не выдает на выход) все символы, содержащиеся в заданной текстовой строке.

6.6. Реализуйте функцию с заголовком

```
size_t  clean (char *str);
```

которая заменяет в строке `str` каждую серию подряд идущих пробелов на один пробел. Возвращаемое значение — новая длина строки `str`.

6.7. Реализуйте функцию с заголовком

```
size_t  clean_p (char *str, char *p);
```

которая заменяет в строке `str` каждую серию подряд идущих символов, встречающихся в строке `p`, на один пробел. Возвращаемое значение — новая длина строки `str`.

6.8. Реализуйте функции преобразования строчных букв в прописные и обратно в указанной строке с учетом русского алфавита в различных кодировках.

6.9. При передаче данных по сетевым каналам некоторые системные программы предполагают, что старший бит каждого байта есть 0. Это может привести к искажению информации, если передаются данные не удовлетворяющие такому условию. Существуют программы, кодирующие и декодирующие данные для возможности такой передачи (uuencode и uudecode). Реализуйте программы для подобной кодировки.

Идеи реализации. Каждые 3 байта входного потока делятся на 4 группы по 6 бит. К каждой такой группе приписываются два старших бита 01 и полученные 4 байта выдаются в выходной поток. Поскольку длина исходного файла не обязательно кратна 3, его можно дополнить одним или двумя байтами и в начале закодированного файла передать общее количество байт в исходном файле. Обратное преобразование выполняется очевидным образом.

6.2. Форматирование текстов

6.10. Напишите программу, которая преобразует длинные строки текстового файла в более короткие, содержащие не более указанного количества символов. При этом разрыв строки может выполняться только на символах из указанного набора символов-разделителей (например, пробелах и табуляциях). Если подобное разбиение невозможно, то программа выдает информационное сообщение и строка остается без изменения.

6.11. Назовем абзацем последовательность строк, не содержащую внутри себя пустых строк (т.е. пустые строки являются разделителями между абзацами). Напишите программу, которая собирает каждый абзац текстового файла в одну длинную строку.

6.12. Допустим, что вид абзаца текста определяется тремя параметрами: позицией начала строки, позицией правой границы абзаца, количеством пробелов в начале красной строки. Напишите программу, которая форматирует абзацы текстового файла в соответствии с данными значениями этих трех параметров без разбиения (переноса) слов. Строки абзаца должны быть выровнены по левой и правой границе. Предполагается, что между словами можно оставлять промежутки любой длины, а в случае невозможности заполнить строку выравнивание выполняется по левой границе в ущерб правой.

6.13. Решите предыдущую задачу при разрешении переносить слова между строками по слогам.

Идеи реализации. Пусть “x” обозначает любую согласную, “o” — любую гласную, “$*$” — любой набор символов (в том числе и пустой), “+” — любой не пустой набор символов, “-” — позицию переноса. Для построения разбиения слова для переноса можно использовать следующие правила:

1) буквы ‘ь,ъ’ составляют единое целое с предшествующей согласной;

2) буква ‘й’ составляет единое целое с предшествующей гласной;

3) нельзя переносить одну букву;

4) слово можно разделить по паре гласных, если разбиение имеет вид $*$xo-o+;

5) слово можно разделить по паре согласных, если разбиение имеет вид $*$ox-x$*$o$*$;

6) слово можно разделить, если разбиение имеет вид $*$xo-xo$*$;

7) слово можно разделить , если разбиение имеет вид +o-$*$o$*$;

Указанные правила выстроены по приоритетности. В частности, правилом 7 следует пользоваться только тогда, когда не удается разбить слово по правилам 4,5,6.

6.14. Напишите программу, форматирующую абзацы текстового файла в соответствии с заданной разметкой. Под разметкой будем понимать следующее. Если в начале очередной строки стоит запись вида

```
\{a,b,c}
```

то это означает, что начиная со следующей строки, все абзацы должны форматироваться с левой границей в позиции `a`, правой границей в позиции `b` и отступом красной строки в `c` пробелов. Аналогичная разметка вида

```
\local{a,b,c}
```

относится только к одному следующему абзацу, после которого восстанавливаются предыдущие параметры форматирования.

6.15. Пусть каждый печатный символ имеет определенную ширину, выражаемую целым числом (эти числа записаны в отдельном массиве). Модифицируйте программы форматирования (задачи XX.XX–XX.XX) в предположении, что параметры абзаца задаются в единицах измерения ширины символов.

6.16. Отформатированный абзац с ровной правой границей, но большими промежутками между словами часто выглядит менее “красиво”, чем абзац с неровной правой границей, но небольшими и равномерно распределенными межсловными промежутками. Придумайте способ вычисления меры “неровности” правой границы x и меры “неравномерности” распределения межсловных промежутков y. Введите коэффициенты штрафов a_x, a_y и реализуйте форматирование абзаца таким образом, чтобы минимизировать функцию общего штрафа $f = a_x x + a_y y$. Посмотрите как меняется результат форматирования в зависимости от значения коэффициентов штрафа a_x, a_y.

6.17. Рассмотрим задачу форматирования страницы при наличии иллюстраций. Пусть заданы размеры страницы (количество строк и количество символов в строке) и некоторое количество запрещенных прямоугольных областей (мест для иллюстраций). Требуется реализовать программу форматирования так, чтобы абзацы текста размещались в разрешенных участках страницы и “огибали” запрещенные области. Запрещенные области могут задаваться позициями левой и правой границ по горизонтали и диапазоном номеров строк по вертикали. Текст, не поместившийся в страницу можно игнорировать.

6.18. Рассмотрим еще один способ выделения места для иллюстраций в тексте. Пусть заданы размеры страницы (количество строк и количество символов в строке) и в тексте помещена разметка размещения иллюстраций следующего вида: `\leftpic{a,b}`, `\midpic{a,b}`, `\rightpic{a,b}`, где `a` есть ширина иллюстрации (количество символов по горизонтали), `b` есть высота иллюстрации (количество строк по вертикали), `\leftpic` означает иллюстрацию, прижатую к левому полю страницы, `\rightpic` — к правому полю, `\midpic` — расположенную строго посередине. Требуется реализовать форматирование текста так, чтобы иллюстрация размещалась целиком на странице и располагалась по возможности ближе к тому месту текста, где была размещена команда разметки этой иллюстрации.

При подготовке документа в текстовых редакторах для форматирования текста используют символы пробела, табуляции и перехода на новую строку. Программы, обрабатывающие html-документы, игнорирую последовательности из нескольких пробелов, заменяя их одним. Переходы на новую строку так же заменяются одним пробелом. При этом форма-

тирование текста осуществляется с учетом текущей ширины окна, типа шрифта и встречающихся внутри документа тегов. С одной стороны это позволяет не заботиться о предварительном форматировании, с другой стороны - внешний вид документа не совпадает с тем, который он имел при подготовке в текстовом редакторе.

6.19. Считая, что в начале html-файла прописаны параметры просмотра: длина выходной строки, размер отступа в красной строке, размер табуляции, тип выравнивания текста, напишите программу, отображающую html-файл с учетом следующих команд:

 (break) - вынужденный перевод строки.

<P> (paragraph) - начало абзаца. По сравнению с
, абзацы разделяются дополнительно пустыми строками и начинаются с красной строки.

<PRE> </PRE> (preformated text) - заключенный в теги текст выводится без изменения.

<H> </H> - заключенный в теги текст выводится заглавными буквами.

<UL> - дальнейший вывод текста осуществляется с отступом от края на шаг табуляции. Каждое новое применение тэга приведет к увеличению сдвига на один шаг. Закрывающий тэг </UL> отменяет действие последнего тэга <UL>. Аналогичным образом отрабатываются команды <OL> </OL>.

<LI> </LI> - пункты списка. Каждый пункт начинается с новой строки. При использовании внутри пары <UL> </UL> в начале каждого пункта автоматически ставится определенный символ (например, точка). При использовании с <OL> пункты автоматически нумеруются числами. По поводу правил форматирования см. задачи 6.12-6.16

6.3. Сжатие и архивация

Программы-архиваторы преобразуют исходные данные в более компактную форму, используя информационную избыточнось кодируемой информации. Подобные программы обычно производят предварительный статистический анализ входной последовательности байтов и заменяют повторяющиеся наборы данных более короткими кодами. Поскольку алгоритмы сжатия, как правило, ориентированы на работу с байтами, мы будем называть байты входного потока символами.

6.3.1. Групповое кодирование (RLE)

Этот способ сжатия заменяет повторяющиеся группы одинаковых входных символов на пару (количество, символ). Например, строка “aaaabbbbbc” будет закодирована в последовательность 4a5b1c. Ясно, что сжатие будет эффективным, если во входном потоке много длинных групп из одинаковых символов. Обычно применяются различные модификации RLE кодирования, позволяющие сократить накладные расходы при появлении непостоянных групп символов.

6.20. Реализуйте кодировщик и декодировщик файлов на основе следующего варианта RLE кодирования.

Для повторяющихся групп длиной от 2 до 127 символов выходным кодом являются два байта (количество,символ). Для групп неповторяющихся символов длиной $n \le 127$ байтов выходным кодом является $n+1$ байт (n+128, исходная группа). Более длинные группы в обоих случаях разбиваются на части длиной не более 127 символов, и каждая кодируется отдельно.

6.21. Реализуйте кодировщик и декодировщик файлов на основе трехбайтового варианта RLE кодирования.

Идеи реализации. Выходным кодом для повторяющихся групп из четырех и более символов являются три байта (флаг,количество, символ). Флаг — это некоторое выделенное значение (например, 255), которое редко встречается во входном потоке. Неповторяющиеся группы передаются на выход как есть, при этом во избежание путаницы сам символ с кодом 255 кодируется тройкой (255,1,255).

6.22. Базовый вариант RLE кодирования хорошо работает для сжатия черно-белых изображений, строки которых представляются длинными последовательностями черных и белых пикселов. Реализуйте RLE кодировщик и декодировщик битового уровня.

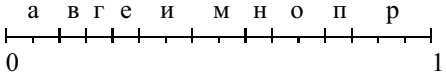
Идеи реализации. В данном случае входным потоком является последовательность битов, кодирующих два возможных цвета пиксела изображения. Выходным кодом является байт, в котором старший бит определяет цвет пиксела, а оставшиеся 7 младших битов дают количество повторений.

6.3.2. Арифметическое кодирование

Суть кодирования состоит в сопоставлении цепочкам символов некоторой (конечной) двоичной дроби, которая строится на основе априорной информации о частотах появления каждого символа в исходном файле. Рассмотрим идею алгоритма на примере кодирования слова “программирование”. Частотный анализ дает следующий результат:

символ	количество	частота
а	2	0.001_2
в	1	0.0001_2
г	1	0.0001_2
е	1	0.0001_2
и	2	0.001_2
м	2	0.001_2
н	1	0.0001_2
о	2	0.001_2
п	1	0.0001_2
р	3	0.0011_2

Разобьем отрезок $[0,1)$ на части, пропорциональные этим частотам и сопоставим каждой части соответствующий символ



Пусть элементы массивов `lb[i]` и `rb[i]` задают границы этих частей для символа с кодом `i`. Опишем алгоритм построения кодирующей дроби.

```
left = 0;
right = 1;
while ( (k=NextSym()) != EOI )
{
   d = right - left;
   right = left + d*rb[k];
   left  = left + d*lb[k];
}
```

По окончании этого цикла в качестве кодирующей дроби можно взять любое число `x`, удовлетворяющее условию `left` $\le$ `x` $<$ `right`. Кодирование по сути представляет собой построение системы вложенных

отрезков, где каждый следующий занимает в предыдущем то место, какое отведено данной букве в исходном разбиении отрезка $[0,1]$. Для обозначения конца кодируемой информации можно дополнительно передавать общее число исходных символов, либо заканчивать входной поток специальным символом (маркером) конца файла EOF. Для слова “программирование” первые несколько отрезков имеют в двоичном представлении следующий вид:

буква	`left`	`right`
п	0.1100	0.1101
р	0.11001101_2	0.11010000_2
о	0.11001110111_2	0.11001111010_2
г	0.110011101111001_2	0.110011101111100_2

а окончательное значение границ `left` и `right` есть

$$\texttt{left} = 0.11001110111101101111000111110101110010011111111110001,$$
$$\texttt{right} = 0.11001110111101101111000111110101110010011111111110010.$$

Таким образом, исходная 16-и байтовая строка сжалась до 51 бита.

Декодирование производится обращением описанного выше процесса.

```
x = <кодирующая дробь>
n = <общее число символов в исходном файле>
while (n--)
{
  k = Interval (x);     /* найти номер интервала, содержащего x */
  Output(Symbol(k));    /* вывести k-й символ */
  d = rb[k]-lb[k]       /* длина интервала для k-го символа */
  x = x - lb[k];        /* преобразуем дробь для определения */
  x /= d;               /*   следующего символа */
}
```

Заметим, что для выполнения кодирования и декодирования следует сначала определить таблицу частот символов в исходном файле. Таким образом, кодировка требует двух просмотров файла, а для декодирования нужна таблица частот, которую приходится помещать в сжатый файл.

6.23. Реализуйте кодировщик и декодировщик файлов по алгоритму арифметического кодирования.

Идеи реализации. Следует заранее задать ограничение на длину двоичного представления кодирующей дроби (например, 6 байт). Вычисление дроби ведется до тех пор, пока различие между битами двоичного

представления границ `left` и `right` находится в пределах этого ограничения. Как только различие между представлениями выйдет из заданных границ, полученная дробь выводится в выходной поток и формирование дроби для следующих символов начинается сначала.

6.24. Реализуйте алгоритм адаптивного арифметического кодирования, не требующий передачи таблицы частот и основанный на следующей идее. В начальный момент таблица частот инициализируется значениями 1/256 (т.е. все символы считаются равновероятными). Очередной символ обрабатывается в соответствии с таблицей, после чего таблица корректируется с учетом поступившего символа. Аналогично, при декодировании таблица частот корректируется после декодирования каждого выходного символа.

6.3.3. Алгоритм Хаффмена

Алгоритм Хаффмена (иногда его называют CCITT кодирование) состоит в сопоставлении каждому символу входного потока некоторого кода переменной длины так, что наиболее часто встречающиеся символы кодируются наиболее короткими кодами. Для этого строится бинарное дерево (дерево Хаффмена), в концевых вершинах которого располагаются все возможные символы, а ветви, ведущие к этим вершинам, соответствуют кодам символов. Для построения дерева нужно иметь таблицу количеств появления каждого символа в исходном файле. Каждая вершина дерева имеет свой вес, который для концевых вершин равен количеству появлений символа, записанного в этой вершине. Процесс построения дерева описывается следующей схемой.

1. Создаем концевые вершины, содержащие все символы входного алфавита, задаем каждой вершине вес в соответствии с таблицей количеств, объявляем все эти вершины свободными.

2. В множестве свободных вершин находим две вершины (обозначим их A и B), имеющие наименьшие веса, создаем новую родительскую вершину C, присоединяем A и B к C справа и слева, устанавливает вес C равным сумме весов A и B, исключаем A и B из множества свободных вершин, добавляем C к множеству свободных вершин.

3. Если в множестве свободных вершин более одного элемента, то перейти к п.2, иначе единственный оставшийся элемент есть корень дерева Хаффмена.

Далее, левым ребрам построенного дерева сопоставляется число 0, а правым ребрам — 1. Теперь, спускаясь от корня к конкретной концевой вершине и последовательно выписывая нули и единицы, сопоставленные проходимым ребрам, мы получаем код символа в этой вершине.

Кодирование и декодирование осуществляется заменой каждого символа его кодом и наоборот. При практической реализации биты кодов символов выписываются подряд, а затем делятся на группы по 8 бит (байты), которые и выдаются в выходной поток. При декодировании входной поток рассматривается как последовательность битов, определяющих направления спуска по ветвям дерева Хаффмена. При достижении концевой вершины соответствующий символ выдается в выходной поток и спуск опять начинается от корня.

6.25. Реализуйте кодировщик и декодировщик файлов по алгоритму Хаффмена. Таблица количеств появлений символов также должна сохраняться в сжатом файле.

Существует адаптивный вариант алгоритма Хаффмена, который не требует передачи дерева (или частотной таблицы) вместе с закодированными данными. Идея этого алгоритма заключается в адаптивном перестроении дерева с учетом поступивших символов при кодировании и соответствующем преобразовании дерева при декодировании. Пусть мы имеем некоторое дерево Хаффмена. Пронумеруем вершины этого дерева “слева-направо, снизу-вверх”, т.е. нумерация начинается с элементов самого нижнего уровня дерева, потом переходит на предыдущий уровень, и т.д. Корень дерева получает максимальный номер.

Будем называть дерево упорядоченным, если веса вершин не убывают в порядке построенной нумерации.

Адаптивное перестроение дерева. Построим начальное дерево Хаффмена, считая, что все символы имеют вес 1. Теперь при появлении нового символа мы должны перестроить дерево так, чтобы сохранить свойство упорядоченности. Во-первых, надо увеличить вес соответствующей концевой вершины и всех родителей по ветви, ведущей к корню. Если после этого дерево остается упорядоченным, то ничего больше делать не надо. Если же вес какой-либо вершины A становится больше веса правого соседа (по нумерации), то среди правых соседей (по нумерации) ищется последний, вес которого меньше веса A, и эта вершина обменивается с A. Затем корректируются веса по ветвям, ведущим к корню от

переставленных вершин. Указанный процесс продолжается, пока дерево не станет упорядоченным.

6.26. Реализуйте кодировщик и декодировщик файлов по адаптивному алгоритму Хаффмена.

Идеи реализации. Обе программы имеют одну и ту же структуру, которая описывается следующей схемой

1. Инициализация дерева равномерным распределение символов.
2. Получить очередной символ (или код) и закодировать (раскодировать) его в соответствии с текущим состоянием дерева Хаффмена.
3. Перестроить дерево на основе полученного символа.
4. Если есть еще символы (коды) во входном потоке, то перейти к п.2, иначе закончить работу.

6.27. Другой способ построения адаптивного алгоритма Хаффмена состоит в инициализации дерева одним специальным символом ESC с количеством появлений 1. Перестройка заключается в *добавлении* в дерево новой концевой вершины для этого символа с количеством появлений 1; если символ поступает повторно, то перестройка проводится по так же как и в предыдущем варианте. Схема кодирования:

1. Инициализация дерева символом ESC.
2. Получить очередной символ.
3. Если символ уже есть в дереве, то выдать его код, иначе выдать код символа ESC и выдать сам символ.
4. Перестроить дерево на основе полученного символа.
5. Если есть еще символы во входном потоке, то перейти к п.2, иначе закончить работу.

Схема декодирования:

1. Инициализация дерева символом ESC.
2. Получить очередной код.
3. Если код есть код символа ESC, то следующие 8 бит представляют собой сам символ, выдать этот символ, иначе выдать символ, соответствующий коду.
4. Перестроить дерево на основе полученного символа.
5. Если есть еще символы во входном потоке, то перейти к п.2, иначе закончить работу.

6.3.4. Алгоритм LZW

В этом разделе описывается простейший вариант Lempel–Ziv–Welch алгоритма (LZW), который в различных модификациях используется в программах-архиваторах.

Суть алгоритма состоит в обнаружении во входном потоке повторяющихся цепочек байтов, составлении таблицы таких обнаруженных цепочек и выдаче в выходной поток кодов (номеров строк таблицы), соответствующих обнаруженной цепочке. Таким образом, если исходный файл имеет много повторяющихся последовательностей байтов, то каждая такая последовательность будет заменена на ее порядковый номер в таблице. Важной особенностью алгоритма является то, что распаковщик, анализируя сжатые данные в процессе работы, может построить копию исходной таблицы и, следовательно, ее не надо передавать вместе с сжатыми данными.

Опишем схему алгоритма, используя С-подобный псевдокод.

Кодирование. Создается таблица, способная вместить достаточно большое количество строк. Первые 256 строк этой таблицы инициализируются всеми возможными односимвольными цепочками (т.е. соответствуют символам с кодами от 0 до 255). Дальнейший алгоритм выглядит так:

```
s = <пустая цепочка>;
while (есть символы во входном потоке)
{  c = NextSym();   /* взять очередной символ */
   if ( InTable (s+c) )
   {  /* цепочка s+c уже есть в таблице */
       s = s+c;  /* удлиняем цепочку прочитанным символом */
   }
   else
   {  /* цепочки s+c нет в таблице */
  OutCode (s);  /* вывод кода, соответствующего цепочке s */
  AddString (s+c); /* добавляем новую цепочку s+c в таблицу */
     s = c;  /* готовимся к обнаружению следующей цепочки */
   }
}
OutCode (s); /* вывод кода для оставшейся цепочки s */
```

Здесь знак “+” обозначает конкатенацию цепочек. Алгоритм кодирования каждый раз пытается найти в таблице наиболее длинную цепочку,

соответствующую читаемой последовательности символов. Если это в какой-то момент не удается, то накопленная к этому времени цепочка заносится в таблицу. В какой-то момент может наступить переполнение таблицы. В этом случае кодировщик выводит в выходной поток специальный код очистки и таблица цепочек инициализируется заново. Обычно в реальных алгоритмах применяется таблица с 4096 входами для цепочек, при этом число 256 является кодом очистки (обозначим его CLC), а число 257 — кодом конца информации (EOI), эти строки таблицы не используется для цепочек. Заметим, что в этом случае для каждого выходного кода достаточно 12 бит. Поэтому при выводе целесообразно упаковывать каждые два кода в три байта. Для упрощения логики декодера обычно первым кодом сжатых данных является CLC, что сразу вызывает инициализацию таблицы цепочек.

Декодирование. Распаковка сжатых данных основывается на построении идентичной таблицы цепочек. Инициализация таблицы выполняется так же как и при кодировании.

```
while ( (k=NextCode()) != EOI )  /* пока не конец информации */
{
   if ( k == CLC )
   {  /* k есть код очистки */
      InitTable();       /* заново инициализируем таблицу */
      k = NextCode();
      if ( k == EOI ) break;
      OutString(k);   /* выводим цепочку, соответствующую
                         коду k в таблице цепочек */
      old = k;        /* запоминаем текущий код для последующей
                         модификации таблицы цепочек */
   }
   else
   {  if ( InTable(k) )
      {  /* в таблице есть строка для кода k */
         OutString(k);  /* выводим цепочку */
           /* формируем и добавляем новую цепочку */
         AddString( String(old)+Char(k) );
         old = k;
      }
      else
      {  /* в таблице нет строки для кода k */
```

```
            s = String(old)+Char(old);  /* формируем цепочку */
            OutString(s);      /* выводим цепочку          */
            AddString(s);      /* и добавляем ее в таблицу */
            old = k;
        }
    }
}
```

В этом алгоритме функция `String(i)` возвращает строку из таблицы, соответствующую коду `i`, а функция `Char(i)` возвращает только первый символ такой строки. Заметим, что функция `NextCode()` должна извлекать 12-битные коды из входного потока байтов.

6.28. Реализуйте описанные алгоритмы в виде программ упаковки и распаковки файлов. Для представления таблицы цепочек можно использовать следующие структуры:

а) массив указателей на текстовые строки;

б) дерево поиска (сбалансированное) для элементов типа `char*`;

в) дерево символов (см. задачу XX.XX).

6.29. Улучшите степень сжатия на основе следующих соображений. Таблица цепочек разрастается по мере просмотра входного файла. Следовательно, пока в таблице цепочек заполнено менее 512-и строк, кодировщик может выдавать 9-битные коды. При появлении 512-й строки, кодировщик переходит на 10-битные коды, после 1024-й строки — на 11-битные, и т.д. Декодер также должен анализировать длину своей таблицы в процессе декодирования и соответственно переходить на извлечение более длинных кодов из входного потока.

7. Грамматический разбор и компиляция

7.1. Формальные грамматики

Мы будем использовать упрощенное определение формальной грамматики как тройки $G = (V, W, F)$, где V есть множество терминальных символов, W — множество нетерминальных (выводимых) символов, F — множество правил вывода. Языком, порожденным грамматикой G будем

называть множество цепочек символов из V и W, образованных в соответствии с правилами вывода F. Для задания формальных грамматик мы будем использовать нормальную форму Бекуса–Наура (возможно, с незначительными модификациями).

7.1. Постройте формальную грамматику для описания всевозможных идентификаторов в языке C.

Решение. Зададим множества символов (терминальных и нетерминальных):

```
V = { 0,1,2,...,9,_,A,B,...,Z,a,b,...,z },
W = { <идентификатор>,<буква>,<цифра> }.
```

Опишем правила вывода в форме Бекуса–Наура.

```
<идентификатор> ::= <буква>|<идентификатор><буква>
                  | <идентификатор><цифра>
<цифра> ::= 0 | 1 | 2 | 3 | 4| 5 | 6 | 7 | 8 | 9
<буква> ::= _ | A | ... | Z | a | ... | z
```

7.2. Постройте формальную грамматику для описания всевозможных десятичных числовых констант, записанных по правилам языка C.

Решение. Выпишем здесь только правила вывода.

```
<константа> ::= <константа без знака>|<знак><константа без знака>
<знак> ::= +|-
<константа без знака> ::= <целое>|<вещественное>
<вещественное> ::= <число>|<число><порядок>
<число> ::= <целое>.|.<целое0>|<целое>.<целое0>
<порядок> ::= E<целое0>|E<знак><целое0>
<целое0> ::= <цифра>|<цифра><целое0>
<цифра>  ::= 0|<цифра1>
<цифра1> ::= 1|2|3|4|5|6|7|8|9
<целое>  ::= 0|<целое1>
<целое1> ::= <цифра1>|<целое1><цифра>
```

Замечание. Для сокращения записи можно в квадратных скобках указывать необязательный фрагмент конструкции. Например, правило вывода для символа `<порядок>` запишется так:

```
<порядок> ::= E[<знак>]<целое0>
```

7.3. Постройте формальную грамматику для описания всевозможных (не только десятичных и числовых) констант, допустимых в языке C.

7.4. Постройте формальную грамматику для описания объявлений переменных, массивов, указателей и массивов указателей в языке C.

Идеи реализации. Используйте метасимволы `<описание>`, `<тип>`, `<список объектов>`, `<имя>`, `<массив>`, `<константа>`, `<указатель>` и др. Например,

```
<описание> ::= <тип><список объектов>;
<указатель> ::= *<имя>|*<указатель>
```

7.5. Постройте формальную грамматику для описания прототипов функций языка C.

7.6. Постройте формальную грамматику для описания арифметических выражений, включающих операции сложения, вычитания, умножения, деления, а также унарные $+$ и $-$ и использующего в качестве операндов простые переменные и числовые десятичные константы.

7.7. Добавьте к предыдущей грамматике возможность включить в выражение одномерные массивы. Учтите возможность появления формулы в индексном выражении.

7.8. Добавьте к предыдущей грамматике возможность включить в выражение вызовы функций.

7.2. Лексический анализатор, конечные автоматы

Основной задачей лексического анализа является выделение из входного текста определенных групп символов — лексем, которые могут являться терминальными или нетерминальными метасимволами в некоторой формальной грамматике. Типичная частная задача подобного рода — обнаружение в тексте вхождений ключевых слов из заданного набора. Наиболее эффективно подобная задача обнаружения решается с помощью конечного автомата. Не вдаваясь в строгую математическую теорию, отметим, что конечный автомат для обнаружения ключевых слов может быть легко реализован в виде ориентированного графа, в котором вершины соответствуют состояниям автомата, а orientorovannie ребра — переходам между состояниями, выполняемым при получении

очередного символа из входного текста. В идейном плане подобная реализация близка к представлению словаря в виде дерева символов (см. задачу XX.XX), которое мы и возьмем в качестве модели для построения конечного автомата.

Итак, пусть мы имеем некоторое дерево символов, ветви которого соответствуют ключевым словам. Пусть одна из вершин этого дерева является текущей. В начальный момент текущей является корневая вершина. При поступлении на вход очередного символа этот символ ищется среди вершин, являющихся непосредственными потомками текущей. Если поиск оказывается успешным, то текущей вершиной становится та, которая содержит этот символ. Если нет, то текущей становится корневая вершина (вообще говоря, это может быть и другая вершина, если ключевое слово имеет несколько совпадающих друг с другом подстрок, см. задачу XX.XX о поиске подстроки). Если текущая вершина соответствует последнему символу в ключевом слове, то фиксируется обнаружение данного ключевого слова и текущей вершиной становится корень дерева.

7.9. Реализуйте конечный автомат для обнаружения набора слов, заданного в виде массива элементов типа `char*`. На основе этой реализации напишите программу, которая читает текстовый файл и выдает на экран все строки, в которых встречается хотя бы одно слово из заданного набора ключевых слов. Нужно предусмотреть возможность задания произвольного набора ключевых слов. Например, эти слова могут считываться из другого текстового файла в начале работы программы.

Некоторые из последующих задач этого пункта, касающиеся лексического разбора и анализа программ на языке C, достаточно сложны и в идеальном варианте требуют учета многих тонкостей синтаксиса языка. Поэтому в начальном варианте реализации целесообразно рассмотреть некоторое подмножество синтаксических правил построения языка, чтобы получить работоспособный вариант программы, который затем можно совершенствовать.

7.10. Реализуйте отдельные функции препроцессора текста программы на языке C.

1. Требуется вставить в текст программы tekst из файла задаваемого директивой `#include`.

2. Требуется обработать директивы `#ifdef` и `#ifndef` и оставить

только тот текст программы, который соответствует определениям, заданным в директивах `#define`.

7.11. Назовем "парой скобок" два заданных набора символов. Один их этих наборов будет "открывающей скобкой", другой — "закрывающей". Примерами подобных скобок могут служить `{` и `}`, `begin` и `end`, `/*` и `*/` и т.п. Целью обработки может быть анализ, сохранение (или уничтожение) символов находящихся между двумя парными скобками.

1. Удалите из файла все комментарии в стиле языка C или C++ (т.е. между `/*` и `*/`, а также `//` и концом строки).

2. Проверьте сбалансированность системы вложенных скобок в программе на языке C или C++. Имеются ввиду скобки `{` и `}` а также скобки `(` `)` и `[` `]` в арифметических и индексных выражениях.

3. Проверьте сбалансированность системы вложенных "скобок" в файле, при условии, что строки, определяющие скобки и правила из взаимного расположения определены в некотором файле. В качестве примеров можно рассмотреть "скобки" в TeX- или HTML-файле.

7.12. Требуется выделить из текста некоторые фрагменты (лексемы), синтаксис появления которых в тексте определяется известной системой правил. Результатом работы программы должен быть некоторый список таких лексем.

1. Выделите из (арифметического) выражения, записанного по правилам языка C, имена переменных (функций) и символьные представления констант.

2. Выделите из файла с программой на языке C объявления функций и сформируйте строки с прототипами этих функций.

3. Выделить из текста программы на языке C объявления и вызовы функций и сформируйте список перекрестных вызовов (т.е. список функций, вызываемых каждой функцией).

4. Имея текст отдельной функции на языке C составьте список локальных и глобальных имен, на которые есть ссылки внутри данной функции.

7.13. Реализуйте программу форматирования текста на заданном алгоритмическом языке (например, C). Выходной текст должен удовлетворять правилам "хорошего стиля", т.е. иметь систему отступов во вложенных конструкциях циклов и разветвлений, промежутки между определениями отдельных функций и т.п.

7.14. Реализуйте программу выделения из HTML-файла всех ссылок на URL (универсальный указатель ресурсов). Форматы HTML-файлов и записей URL описаны во множестве руководств, и мы их здесь приводить не будем.

7.3. Построение дерева грамматического разбора

Анализ формальной грамматики, описывающей некоторый язык, позволяет построить дерево разбора, в котором концевые вершины будут соответствовать терминальным метасимволам, а все остальные вершины — нетерминальным метасимволам. В случае арифметического выражения с обычными арифметическими операциями это дерево будет бинарным, причем концевым вершинам можно сопоставить значения переменных или констант, участвующих в выражении, а остальные вершины — с операциями и значениями результатов этих операций. Обходя дерево снизу-вверх и выполняя операции, сопоставленные вершинам, в корне дерева мы получим результат вычисления всего выражения.

7.15. Реализуйте алгоритм построения дерева разбора для арифметического выражения, заданного текстовой строкой. На основе построенного дерева реализуйте функцию вычисления значения выражения при заданных значениях переменных, участвующих в выражении.

Идеи реализации. Определим тип вершины дерева как

```
typedef struct _T {
        char *text;
        double value;
        struct _T *left, *right;
} TreeNode;
```

где `text` содержит имя переменной или знак операции, `value` — значение переменной или результат операции.

Введем скобки `{` `}` для обозначения фрагмента, который может повторяться нуль или более раз. Определим формальную грамматику для выражения следующим образом:

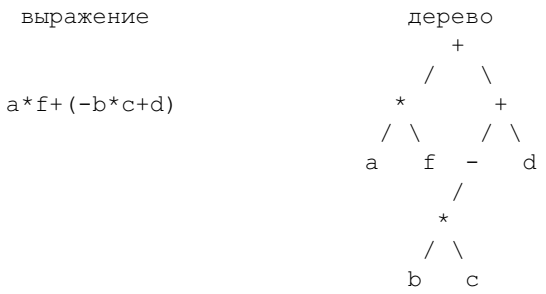
```
<выражение> ::=  [<плюс/минус>]<терм>{<плюс/минус><терм>}
<терм> ::= <множитель>{<умножить/разделить><множитель>}
<множитель> ::= <переменная>|<константа>|(<выражение>)
<плюс/минус> ::= +|-
```

```
<умножить/разделить> ::= *|/
<переменная> ::= <идентификатор>
```

Символы <идентификатор> и <константа> мы определили ранее. Итак, выражение без знака есть последовательность термов с аддитивными операциями, а терм — последовательность множителей с мультипликативными операциями. Построение дерева реализуется набором рекурсивных процедур, которые строят деревья для выражения без знака, терма и множителя. Например, для выражений A+B+C (A,B,C — термы) и a*b*c (a,b,c — множители) должны получиться деревья

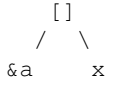
В случае обработки множителя, являющимся выражением в скобках, строится дерево для этого выражения и его корень присоединяется к вершине, соответствующей данному множителю. Унарные операции плюс и минус реализуются как вершины дерева с одним потомком расположенные над соответствующим термом.

Пример.



7.16. Добавьте к предыдущей реализации возможность использования в выражении одномерных массивов.

Идеи реализации. Имя массива определяет базовый адрес, а индекс — смещение относительно этого адреса. Таким образом операция индексирования [] по значению базового адреса и смещения определяет значение элемента. Если обозначить &a — базовый адрес массива a, то выражению a[x] можно сопоставить дерево



7.4. Применение деревьев грамматического разбора

Деревья грамматического разбора активно применяются в компиляторах алгоритмических языков и других программах, где требуется обработка текстов, записанных по правилам некоторой формальной грамматики (например, WEB-браузерах, системах подготовки печатных изданий и пр.). В этом разделе мы рассмотрим примеры применения деревьев грамматического разбора для решения нескольких интересных задач.

7.17. Реализуйте интерпретатор модельного алгоритмического языка со следующим синтаксисом:

`var x,y[10]`	объявление переменной x и массива y на 10 элементов;
`x=<выражение>`	операторы присваивания;
`y[x]=<выражение>`	
`print x,y[2]`	вывод значений переменных;
`end`	конец программы.

Идеи реализации. Результат обработки инструкций описания переменных заключается в резервировании необходимого количества памяти и распределении этой памяти между заданными переменными. Таким образом, получается таблица распределения памяти, устанавливающая соответствие между именем переменной (или массива) и базовым адресом этой переменной. Обработка оператора присваивания выполняется аналогично задаче XX.XX, при этом значения переменных пишутся и читаются в памяти в соответствии с таблицей распределения памяти.

Если выполнить обход дерева разбора арифметического выражения снизу вверх, выписывая при этом имена переменных и операции, сопоставленные вершинам, то мы получим так называемых постфиксный код, который оказывается очень удобным для вычисления выражения с использованием стекового калькулятора. Так, например, выражение a*b+c*(d+e) дает постфиксный код ab*cde+*+. Стековый калькулятор имеет стек, содержащий числа, и может выполнять действия с

двумя верхними элементами. Результат операции замещает эти два элемента в стеке. Выпишем состояния стекового калькулятора при просмотре указанного постфиксного кода (все выражения представляют собой вычисленные значения).

```
                                    e
стек                          d     d    d+e
            b           c     c     c     c     c*(d+e)
       a    a    a*b   a*b   a*b   a*b   a*b     a*b      a*b+c*(d+e)
      ---  ---  ---   ---   ---   ---   ---    -------    -----------

код    a    b    *     c     d     e     +        *            +
```

Имея реализацию стекового калькулятора, мы может разделить процесс интерпретации на два этапа: построение постфиксного кода и интерпретацию кода стековым калькулятором. Этот вывод приводит к следующей задаче.

7.18. Преобразуйте интерпретатор из задачи XX.XX так, чтобы он сначала строил некоторый код, аналогичный постфиксному, а затем с помощью стекового калькулятора его интерпретировал. В результате должен получиться “компилятор” и “исполнитель” программы на исходном модельном языке.

Идеи реализации. В рассматриваемом модельном языке присутствуют операции индексирования и чтения/записи в память. Поскольку адреса элементов массива тоже приходится вычислять, следует дополнить стековый калькулятор операциями чтения из памяти и записи в память. Для простоты можно считать, что “адрес” в нашем понимании — это индекс элемента в объединенном массиве всех данных компилируемой программы. Вот одно из возможных решений.

Чтение из памяти. Вершина стека есть адрес, в результате операции вершина заменяется на значение по этому адресу.

Запись в память. Вершина стека есть значение, под ней — адрес, в результате операции происходит запись значения по указанному адресу и оба элемента удаляются из стека.

Заметим также, что при формировании кода нет необходимости строить дерево разбора целиком. Действительно, если сформирован родитель c двух концевых вершин a, b, то мы можем сразу приписать вершине c кодовую строку abc, а вершины a, b удалить. С идейной точки зрения это

означает, что, просматривая арифметическое выражение слева-направо, мы сразу выполняем те вычисления, которые можно выполнить с учетом скобок и приоритетов операций (а также ранее произведенных вычислений).

7.19. Добавьте в модельный алгоритмический язык инструкции условного и безусловного переходов и реализуйте их в компиляторе и исполнителе.

Идеи реализации. Требуемые инструкции можно задать в виде

```
<метка> : <оператор программы>
goto <метка>
if (<выражение>) goto <метка>
```

В систему команд стекового калькулятора можно добавить команду безусловного перехода на команду с заданным порядковым номером и команду перехода на команду с заданным порядковым номером при условии, что текущая вершина стека калькулятора отлична от нуля. Обнаружив в тексте программы метку, следует запоминать в отдельной таблице номер первой команды, соответствующей помеченному оператору. Переход на предшествующую метку не вызывает проблем (номера команд уже присутствуют в таблице). Для перехода на метки, которые пока еще не встретились, можно формировать код перехода с неопределенным номером и запоминать в таблице позиции кода этих неопределенных ссылок. После обработки всего текста, все объявленные метки будут зафиксированы в таблице и мы сможем записать истинные номера команд в неопределенные операторы переходов.

7.20. Добавьте в модельный алгоритмический язык инструкции циклов и реализуйте их в компиляторе и исполнителе.

Идеи реализации. Циклы легко реализуются с помощью команд переходов. Вот один из возможных вариантов введения и интерпретации цикла:

```
while (<выражение>)          m1 : if ( !<выражение> ) goto m2
   <тело цикла>                   <тело цикла>
endwhile                          goto m1
                             m2 :
```

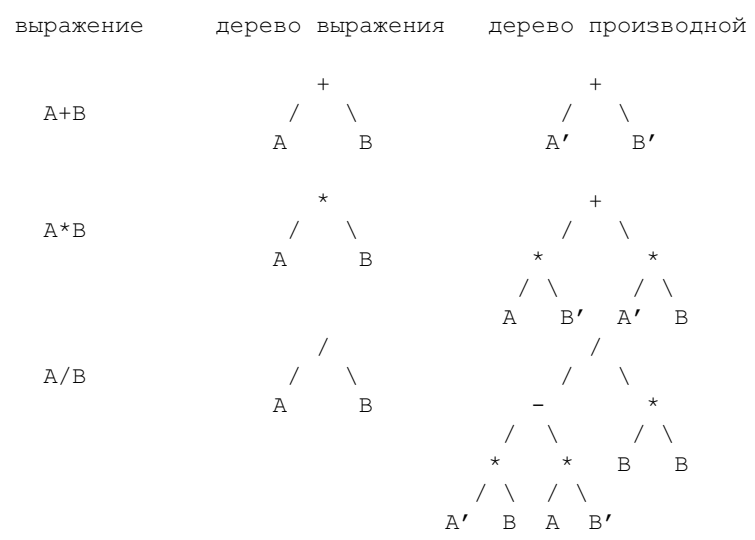
7.21. Добавьте в модельный алгоритмический язык возможность использовать вызовы стандартных функций из некоторого заданного набора (например, элементарных математических функций).

7.22. Реализуйте программу рисования графика функции $y = f(x)$ по заданной формуле для функции $f(x)$.

Идеи реализации. Выражение, задающее функцию $f(x)$, переводится в код стекового калькулятора. Затем этот код многократно используется для вычисления координат точек, принадлежащих графику.

7.23. Реализуйте программу, которая по заданной формуле для функции $f(x)$ формирует выражения для производной $f'(x)$.

Идеи реализации. Построим дерево разбора для выражения функции $f(x)$. Далее строится дерево разбора для производной по правилам дифференцирования. Например, если A и B — деревья для двух выражений, A′ и B′ — деревья для их производных, то имеем следующие правила формирования деревьев:



Обходя дерево производной слева-направо и расставляя скобки, где это необходимо, можно получить формулу для производной. Отметим, что этот вариант формулы может оказаться весьма громоздким (см. задачу XX.XX).

7.24. Пусть заданы формула и ее дерево разбора. Предложите и реализуйте алгоритм для упрощения этой формулы (приведение подобных

членов, вычисление константных выражений, вынос общих множителей).

Идеи реализации. Эта задача достаточно сложна. Скажем только, что здесь нужно анализировать структуру дерева разбора, отыскивать в нем подобные поддеревья, находить преобразования дерева, сохраняющие результирующее значение в корне и уменьшающие сложность.

7.25. Реализуйте оптимизирующий компилятор модельного языка.

Идеи реализации. Ограничимся линейными последовательностями операторов присваивания. Результатом грамматического разбора исходного текста является набор деревьев для каждого оператора присваивания. Можно провести анализ этих деревьев по следующим критериям:

а) константные поддеревья заменить на соответствующие константы;

б) обнаружить одинаковые поддеревья и выделить их в отдельное дерево с сохранением промежуточных результатов в памяти;

в) вынести вычисление выражений, не меняющихся в цикле, из тела цикла.

7.26. Пусть даны тексты двух функций f() и g(), написанных на языке C, C++, которые правильно решают некоторую задачу. Требуется оценить степень подобия данных функций. На основании полученных результатов попытаться определить, является ли текст g() прототипом текста f().

Идеи реализации. При решении данной задачи удобно выделить характерные типы модификаций, которые используются для изменения внешнего вида программы с минимальными усилиями и для каждого из них придумать алгоритм, учитывающий подобные изменения.

а) текст f() получен из текста g() переформатированием и переименованием переменных. Исключим из обоих функций разделительные символы (пробелы, табуляции, символы новых строк) и имена переменных. Полученные в итоге записи F и G побайтно сравним.

б) в текст f() дополнительно добавлены некоторые операторы. При сравнении F и G будем считать, что записи равны, если они совпадают после выбрасывания некоторого количества байт из F.

в) в тексте f() некоторые операторы "расщеплены"на несколько и возможно изменен порядок некоторых операторов. При сравнении F и G разрешается пропускать лишние байты из обоих записей.

В ходе работы программы необходимо подсчитать длины совпадающих и не совпадающих участков, закон их чередования и отдельно

распечатать не совпадающие части функций.

В качестве иного подхода можно рассмотреть статистический метод. Основная идея заключается в следующем: в зависимости от содержимого текста введем понятие ключевых объектов. Для литературного текста это могут быть имена людей, названия местностей, ссылки на других авторов и т.д. При разборе С-функций естественно рассмотреть ключевые слова языка: `for`, `while`, `int`, `#define` и т.д.

Каждый объект характеризуется своим типом и некоторой количественной величиной: частотой использования для переменных, `define`-констант, количеством операторов для циклов и т.д. При сравнении функций будем учитывать взаимное расположение ключевых объектов. Для этого для каждой функции построим таблицу встречающихся в тексте объектов.

```
F:                                G:
Формальное  ключевое  число       Формальное  ключевое  число
   имя       слово                   имя       слово
---------------------------       ---------------------------
   A:       int (i)    10            e:       int (i)    10
---------------------------       ---------------------------
   B:      float (tmp)  5            a:      float (tmp)  5
---------------------------       ---------------------------
   C:       for         1            c:       for         1
........                          ........
```

и запишем порядок использования объектов в функциях: `F: ABCAB ...`, `G: efacd ...`. Будем считать, что две функции близки, если их строки чередования ключевых слов похожи. Очевидно, что для данного подхода можно указать две совершенно разные функции с одинаковыми статистическими характеристиками.

Список литературы

1. Документация по различным версиям UNIX AIX, DEC OSF/1, Linux, Solaris, System V.
2. Джеймс С. Армстронг. **Секреты UNIX.** “Диалектика”. Киев. 1996.
3. Керниган Б. В., Пайк Р. **UNIX — универсальная среда программирования.** “Финансы и Статистика”. Москва 1992.
4. Matt Welsh. **Linux Installation and Getting Started.** HTTP-version. Version 2.2.2, (c)1992-1994 by Matt Welsh, mdw@sunsite.unc.edu 12 February 1995

Учебное издание

Валединский Владимир Дмитриевич, Корнев Андрей Алексеевич

Методы программирования в примерах и задачах.

М.: Издательство механико-математического факультета МГУ,
2000.— 152 с.

Подписано в печать ..... 25.02.2000 г.
Формат 60×90 1/16. Объем 5.5 п.л.
Заказ 4. Тираж 400 экз.

Издательство механико-математического факультета МГУ
г. Москва, Воробьевы горы.
Лицензия на издательскую деятельность ЛР В 020806,
от 23.08.1993г.

Отпечатано с оригинал–макета на типографском оборудовании
механико-математического факультета и Франко-русского
центра им. А.М. Ляпунова.