Е.К. Липачёв

Технология программирования. Методы сортировки данных

Учебное пособие

Казанский университет

2017

УДК 004.43

ББК 32.973.26–018

Принято на заседании Учебно-методической комиссии

Института математики и механики им. Н.И. Лобачевского КФУ

Протокол № 7 от 13 апреля 2017 года

Рецензенты:

доктор физ.-мат. наук **А.М. Елизаров;**

доктор физ.-мат. наук **С.Н. Тронин**

Липачёв Е. К.

Технология программирования. Методы сортировки данных: учебное пособие / Е.К. Липачёв. – Казань: Казан. ун-т, 2017. – 58 с.

Дано краткое описание методов сортировки данных на языке С/С++.
За основу взяты лекции по курсу «Технология программирования и работа на ЭВМ» по направлению подготовки 010800 Механика и математическое моделирование, прочитанные автором.

УДК 004.43

ББК 32.973.26–018

© **Казанский университет, 2017**

© **Липачёв Е.К., 2017**

Оглавление

Сортировка: внутренняя и внешняя 4

Метод “пузырька” 5

Метод просеивания 17

Метод Шелла 19

Быстрая сортировка 23

Пирамидальная сортировка 28

- **Представление массива в виде дерева** 28
- **Построение пирамиды** 31
- **Сортировка с помощью пирамиды** 29

Сортировка слиянием 34

Внешняя сортировка 37

- **Напоминание по работе с файлами** 37
- **Пояснение работы метода `eof()`** 40
- **Модель внешней сортировки** 42
- **Естественное слияние** 42

Литература 58

Введение

Задача книги кратко познакомить с основными алгоритмами сортировки данных. Методы сортировки уже реализованы практически во всех языках программирования и для сортировки данных достаточно простого вызова соответствующей функции, которую несложно найти в справочной службе среды программирования. Тем не менее, знание алгоритмов сортировки, их отличий, преимуществ и недостатков, является определенным признаком квалификации программиста. Отметим в связи с этим книгу [1], посвященную подготовке к прохождению тестовых заданий в ведущих IT-компаниях. Отдельная глава этой книги отведена методам сортировки, при этом отмечено, что интервьюеры любят задачи на сортировку, так как они позволяют охватить широкий круг проблем от алгоритмической сложности до использования памяти. Третий том известной серии книг Д. Кнута "Искусство программирования для ЭВМ" [2] имеет подзаголовок "Сортировка и поиск", чем подчеркивается важность методов сортировки при овладении искусством программирования. Отметим, что указанная серия книг Д. Кнута считается "настольной книгой" каждого программиста – в Интернете можно найти много высказываний на эту тему.

Сортировка: внутренняя и внешняя

Слово "сортировка" происходит от английского "sorting", что, по смыслу, означает отбор по сортам. Программисты используют этот термин более узко, как процесс выстраивания элементов в определенном порядке, например, по возрастанию какого-либо значения элементов рассматриваемого списка. Как отмечено в [2], этот процесс правильнее было бы назвать не сортировкой, а упорядочением (в английском языке это было бы ordering), но использование данного слова привело бы к путанице из-за перегруженности значениями слова "порядок".

Под термином *сортировка* понимается процедура перестановки элементов множества в определённом порядке, т.е. если даны элементы

$$a_1, \ldots, a_n,$$

то сортировка означает перестановку этих элементов в таком порядке

$$a_{k_1}, a_{k_2}, \ldots, a_{k_n},$$

при котором для заданной *функции упорядочения* f справедливо соотношение:

$$f\left(a_{k_1}\right) \le f\left(a_{k_2}\right) \le \ldots \le f\left(a_{k_n}\right).$$

Например, если рассмотреть список элементов `12, 4, 8, 3, 11, 5, 7, 4` и функцию упорядочения `a<b`, то результатом сортировки будет массив: `3, 4, 4, 5, 7, 8, 11, 12`.

Методы сортировки классифицируются на внутренние и внешние. При *внутренней сортировке* данные размещаются в оперативной памяти, например в массиве. При *внешней сортировке* данные находятся во внешней памяти. К внешней сортировке прибегают в случаях, когда невозможно разместить в оперативной памяти все данные.

Метод “пузырька”

Это самый известный и простой способ упорядочения данных. Производятся последовательные просмотры массива и каждый раз пару за парой сравниваются соседние числа. Если числа в паре не расположены в порядке возрастания меняем их местами. Затем переходим к следующей паре. Сортировка считается законченной, если в ходе просмотра не была произведена ни одна перестановка. Можно использовать переменную, например, `flag`, чтобы зафиксировать факт перестановки, – в начале каждого просмотра переменной присваивается значение `0` (`False`) и если в ходе просмотра была выполнена хотя бы одна перестановка, значение переменной `flag` меняется на `1` (`True`). Таким образом, по значению этой переменной определяем, нужен или нет ещё один просмотр.

Вариант 1. Метод реализован на стандартном языке С в виде консольного приложения. Для вывода на экран используется функция `printf()`, а операторы, реализующие метод сортировки размещены в главной функции `main()`

```
/* Sortmass.cpp:  Упорядочивание массива по возрастанию методом пузырька */

#include "stdafx.h"
#include <iostream>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    const int n=20;
    int   x[n]   =   {19,20,1,2,3,5,4,7,6,18,8,17,9,11,
10,12,14,13,16,15};
    int i,flag,m,t;
```

```
    printf("\n");
    for (i=0;i<n;i++) printf(" %d ",x[i]);
    m = n;
    do{
        flag=0;
        m--;
        for (i=0;i<m;i++)
            if (x[i]>x[i+1])
             {
                 // Перестановка
                 t=x[i];
                 x[i]=x[i+1];
                 x[i+1]=t;
                 flag=1;
             }
    }
    while (flag!=0);
    printf("\n");
    for (i=0;i<n;i++) printf(" %d ",x[i]);
    return 0;
}
```

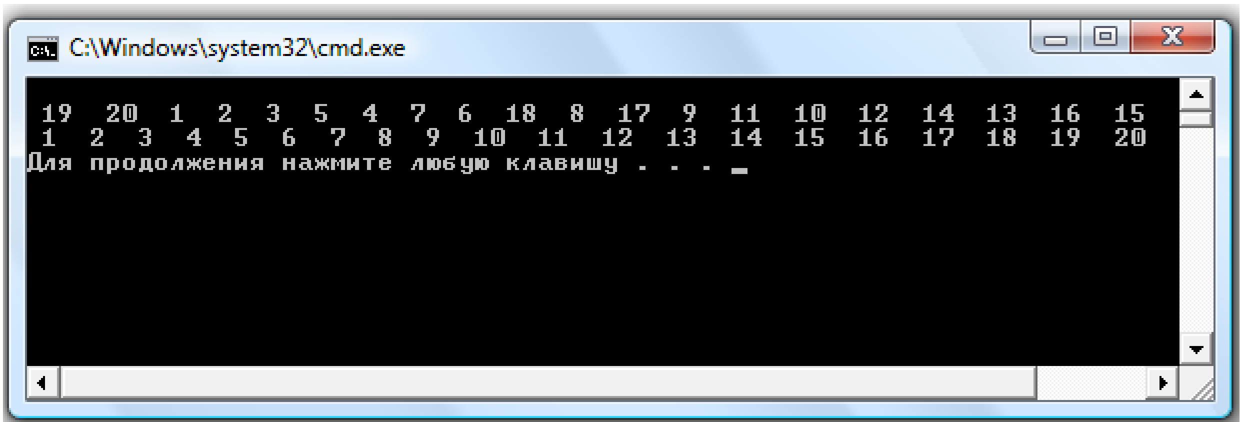


Рис.1. Результат работы программы

Далее будут предложены несколько вариантов стилистического улучшения программы.

В программе есть блок перестановки элементов:

```
if (x[i]>x[i+1])
             {
                 // Перестановка
                 t=x[i];
                 x[i]=x[i+1];
                 x[i+1]=t;
                 flag=1;
             }
```

В языке C часто используют макроопределения (в эпоху до C++ применялись наиболее активно). Макроопределение задается с помощью препроцессорной директивы #define, размещаемой до основного кода программы

```
#include "stdafx.h"
#include <iostream>
#define SWAP(a,b) t=(a); (a)=(b); (b)=t;
```

Блок перестановки в этом случае принимает вид

```
            if (x[i]>x[i+1])
             {
                 // Перестановка
                 SWAP(x[i],x[i+1]);
                 flag=1;
             }
```

Макроопределение состоит из двух частей: в левой задается имя (в нашем случае **SWAP**), а после пробела записывается код. При обработке программы препроцессор заменит макровхождение на код, соответствующий данному макросу. Обращаем внимание на скобки у параметров макроса.

Вариант 2. В отличие от первого варианта, операция перестановки элементов записана с использованием макроопределения

```
#include "stdafx.h"
#include <iostream>
#define SWAP(a,b) t=(a); (a)=(b); (b)=t;
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    const int n=20;
    int x[n]={19,20,1,2,3,5,4,7,6,18,8,17,9,11,
10,12,14,13,16,15};
    int i,flag,m,t;
    printf("\n");
    for (i=0;i<n;i++) printf(" %d ",x[i]);
    m = n;
    do{
        flag=0;
        m--;
        for (i=0;i<m;i++)
            if (x[i]>x[i+1])
             {
                 // Перестановка
                 SWAP(x[i],x[i+1]);
                 flag=1;
```

```
                }
        }
        while (flag!=0);
        printf("\n");
        for (i=0;i<n;i++) printf(" %d ",x[i]);
        printf("\n");
        return 0;
}
```

Вариант 3. В главной программе только формирование данных и опреаторы вывода результатов. Алгоритм сортировки реализован в функции `sort_change()`, перестановка элементов также выполняется отдельной функцией – `Swap()`. Отметим, что имя функции перестановки начинается с прописной буквы во избежание путаницы со стандартной функцией языка С.

```
#include "stdafx.h"
#include <iostream>
using namespace std;

void Swap(int *, int *);
void sort_change(int *, int);

int _tmain(int argc, _TCHAR* argv[])
{
        const int n=20;
        int x[n]={19,20,1,2,3,5,4,7,6,18,8,17,9,11,
10,12,14,13,16,15};
        int i;
        printf("\n");
        for (i=0;i<n;i++) printf(" %d ",x[i]);
        sort_change(x, n);
        printf("\n");
        for (i=0;i<n;i++) printf(" %d ",x[i]);
        printf("\n");
        return 0;
}

void Swap(int *pa, int *pb)
{
        int t;
        t=*pa;
        *pa = *pb;
        *pb =t;
}
```

```
void sort_change(int *px, int dim)
{
   int i,flag,m;
   m = dim;
    do{
        flag=0;
        m--;
        for (i=0;i<m;i++)
            if (px[i]>px[i+1])
             {
                 // Перестановка
                 Swap(&px[i],&px[i+1]);
                 flag=1;
             }
    }
    while (flag!=0);
}
```

Вариант 4. В определении термина “сортировка” присутствовала функция упорядочения. В этом варианте программы добавлена простейшая функция упорядочения – `icom_funct()`. Имя функции передается функции сортировки как параметр.

```
#include "stdafx.h"
#include <iostream>
using namespace std;

void Swap(int *, int *);
int icom_funct(int , int);
void sort_change(int *, int, int (*)(int , int ));

int _tmain(int argc, _TCHAR* argv[])
{
    const int n=20;
    int x[n]={19,20,1,2,3,5,4,7,6,18,8,17,9,11,
10,12,14,13,16,15};
    int i;
    printf("\n");
    for (i=0;i<n;i++) printf(" %d ",x[i]);
    sort_change(x, n,icom_funct);
    printf("\n");
    for (i=0;i<n;i++) printf(" %d ",x[i]);
    printf("\n");
    return 0;
}
```

```
void Swap(int *pa, int *pb)
{
     int t;
     t=*pa;
     *pa = *pb;
     *pb =t;
}

void sort_change(int *px, int dim, int (*funct_ptr)(int
, int ))
{
    int i,flag,m;
    m = dim;
     do{
          flag=0;
          m--;
          for (i=0;i<m;i++)
               if (funct_ptr(px[i],px[i+1])>0)
                {
                    // Перестановка
                    Swap(&px[i],&px[i+1]);
                    flag=1;
                }
     }
     while (flag!=0);
}

int icom_funct(int a, int b)
{
 return (a-b);

}
```

Напомним, как имя функции можно передать в качестве параметра другой функции (подробнее см., например, [3]). В приведенном примере третий параметр функции `sort_change(x, n,icom_funct)` содержит имя функции упорядочения. При объявлении функции `sort_change()` нужно учесть сигнатуру функции `icom_funct(int a, int b)` и тип её возвращаемого значения, в соответствии с которыми и описывается соответствующий формальный параметр функции `sort_change()`:

```
void sort_change(int *, int, int (*)(int , int ));
```

Вариант 5. В этом примере сделан переход от массива чисел к более сложному типу данных. Сортируем массив объектов класса `Cdrom`. Класс

содержат атрибуты speed и year, означающие, соответственно, скорость устройства и год его выпуска, а также операции (методы) доступа к ним. Пример призван показать, что переход от сортировки чисел к сортировке данных, имеющих сложную структуру, достаточно прост.

```
// Сортировка массива объектов
//
#include "stdafx.h"
#include <iostream>
using namespace std;

class Cdrom{
    int speed;   int year;
public:
    Cdrom(int s,int y=2002){speed=s; year=y;}
    Cdrom(){speed=32;year=1999;}
    Cdrom(const Cdrom & c){speed=c.speed;year=c.year;}
    ~Cdrom(){}
    void set_speed(int s) {speed=s;}
    int get_speed()const {return speed;}
    void set_year(int y) {year=y;}
    int get_year()const {return year;}
    const Cdrom & Sravnenie (const Cdrom &p) const
    {return (speed<p.speed) ?  p : *this; }
};

void Swap(Cdrom *, Cdrom *);
int icom_funct_S(Cdrom , Cdrom);
int icom_funct_Y(Cdrom , Cdrom);
void sort_change(Cdrom *, int, int (*)(Cdrom,Cdrom));

int _tmain(int argc, _TCHAR* argv[])
{
    const int n=10;
    int i;
    Cdrom x[n]={Cdrom(52,2005),
                Cdrom(42,2004),
                Cdrom(54,2005),
                Cdrom(40,2003),
                Cdrom(58,2007),
                Cdrom(52,2004),
                Cdrom(62,2008),
                Cdrom(20,2002),
                Cdrom(16,2000),
                Cdrom(12,1999)};
```

```
//
    printf("\n");
    for (i=0;i<n;i++) printf(" %d ",x[i].get_speed());
// Упорядочиваем по скорости:
     sort_change(x, n,icom_funct_S);
//
     printf("\n\n By Speed:\n");
    for (i=0;i<n;i++) printf(" %d ",x[i].get_speed());
    printf("\n");

// Упорядочиваем по году выпуска:
     sort_change(x, n,icom_funct_Y);
//
     printf("\n\n By Speed:\n");
    for (i=0;i<n;i++) printf(" %d ",x[i].get_year());
    printf("\n");
    return 0;
}

void Swap(Cdrom *pa, Cdrom *pb)
{
    Cdrom t;
    t=*pa;
    *pa = *pb;
    *pb =t;
}

void sort_change(Cdrom *px, int dim, int
(*funct_ptr)(Cdrom, Cdrom ))
{
   int i,flag,m;
   m = dim;
    do{
        flag=0;
        m--;
        for (i=0;i<m;i++)
            if (funct_ptr(px[i],px[i+1])>0)
             {
                 // Перестановка
                 Swap(&px[i],&px[i+1]);
                 flag=1;
             }
    }
    while (flag!=0);
}
```

```
int icom_funct_S(Cdrom a, Cdrom b)
{
 return (a.get_speed()-b.get_speed());
}

int icom_funct_Y(Cdrom a, Cdrom b)
{
 return (a.get_year()-b.get_year());

}
```

```
C:\Windows\system32\cmd.exe
52  42  54  40  58  52  62  20  16  12

By Speed:
12  16  20  40  42  52  52  54  58  62


By Year:
1999  2000  2002  2003  2004  2004  2005  2005  2007  2008
Для продолжения нажмите любую клавишу . . .
```

Рис.2. Результат работы программы

Вариант 6. Как можно догадаться, операции, производимые при сортировке данных, совершенно одинаковы для различных типов данных. Но особенности хранения данных и обращения к ним требуют написания отдельного кода (практически повторяющегося) для каждого типа данных. Преодолеть это затруднение можно с помощью шаблонов (см., например, [4]) В следующем примере шаблоны используются при сортировке массивов целых чисел, вещественных чисел и объектов некоторого класса.

```
// Sort-03sh.cpp : Использование шаблонов при сортировке
//
#include "stdafx.h"
#include <iostream>
using namespace std;

class Cdrom{
      int speed;      int year;
public:
      Cdrom(int s,int y=2002){speed=s; year=y;}
      Cdrom(){speed=32;year=1999;}
      Cdrom(const Cdrom & c){speed=c.speed;year=c.year;}
      ~Cdrom(){}
      void set_speed(int s) {speed=s;}
int get_speed()const {return speed;}
```

```
     void set_year(int y) {year=y;}
     int get_year()const {return year;}
     const Cdrom & Sravnenie (const Cdrom &p) const
     {return (speed<p.speed) ?  p : *this; }
};


template <class T> void Swap(T *, T *);
template <class T> void sort_change(T*, int, int (*)(T , T));
int icom_funct_int(int , int);
int icom_funct_float1(float , float);
int icom_funct_float2(float , float);
int icom_funct_S(Cdrom , Cdrom);
int icom_funct_Y(Cdrom , Cdrom);


int _tmain(int argc, _TCHAR* argv[])
{
     const int n=20;
     int x[n]={19,20,1,2,3,5,4,7,6,18,8,17,9,11,
10,12,14,13,16,15};
     int i;
     printf("\n                    Sorting     \n\n");
     printf("\n     Massiv int \n");
    for (i=0;i<n;i++) printf(" %d ",x[i]);
     sort_change(x, n,icom_funct_int);
     printf("\n Sort: \n");
    for (i=0;i<n;i++) printf(" %d ",x[i]);
     printf("\n");
//
// Теперь вещественные числа:
     float v[n]={19.1,20.2,1.1,3.2,5.1,4.2,7.0,6.9,1.8,
0.8,1.7,0.9,1.1,1.0,1.2,1.4,1.3,1.6,1.5,20.0};
     printf("\n    Massiv float \n");
     for (i=0;i<n;i++) printf(" %4.1f",v[i]);
     sort_change(v, n,icom_funct_float1);// по возрастанию
     printf("\n\n  Sort ...<a<b<... \n");
    for (i=0;i<n;i++) printf(" %4.1f",v[i]);
     printf("\n");
//
     sort_change(v, n,icom_funct_float2);// по убыванию
     printf("\n  Sort ...>a>b>...\n");
    for (i=0;i<n;i++) printf(" %4.1f",v[i]);
     printf("\n");
//
//  Теперь объекты некоторого класса:
     const int m=10;
     Cdrom w[m]={Cdrom(52,2005),
                 Cdrom(42,2004),
                   Cdrom(54,2005),
                   Cdrom(40,2003),
                   Cdrom(58,2007),
```

```
                     Cdrom(52,2004),
                     Cdrom(62,2008),
                     Cdrom(20,2002),
                     Cdrom(16,2000),
                     Cdrom(12,1999)};
     printf("\n    Massiv objects \n");
// Упорядочиваем по скорости:
     sort_change(w, m,icom_funct_S);
//
      printf("\n\n Sort By Speed:\n");
    for (i=0;i<m;i++) printf(" %d ",w[i].get_speed());
    printf("\n");

// Упорядочиваем по году выпуска:
     sort_change(w, m,icom_funct_Y);
//
      printf("\n\n Sort By Year:\n");
    for (i=0;i<m;i++) printf(" %d ",w[i].get_year());
    printf("\n");

     return 0;
}

template <class T> void Swap(T *pa, T *pb)
{
     T t;
     t=*pa;
     *pa = *pb;
     *pb =t;
}

template <class T> void sort_change(T *px, int dim, int
(*funct_ptr)(T , T))
{
   int i,flag,m;
   m = dim;
     do{
          flag=0;
          m--;
          for (i=0;i<m;i++)
               if (funct_ptr(px[i],px[i+1])>0)
               // if ((px[i]-px[i+1])>0)
                {
                     // Перестановка
                     Swap(&px[i],&px[i+1]);
                     flag=1;
                }
     }
     while (flag!=0);
}

int icom_funct_int(int a, int b)
```

```
{
 return (a-b);
}

int icom_funct_float1(float a, float b)
{ // по возрастанию
     return (a-b)>0?1:-1;
}

int icom_funct_float2(float a, float b)
{ // по убыванию
     return (b-a)>0?1:-1;
}

int icom_funct_S(Cdrom a, Cdrom b)
{ // По скорости
 return (a.get_speed()-b.get_speed());
}

int icom_funct_Y(Cdrom a, Cdrom b)
{ // По году выпуска
 return (a.get_year()-b.get_year());
}
```

```
C:\Windows\system32\cmd.exe
                   Sorting

       Massiv int
19  20  1  2  3  5  4  7  6  18  8  17  9  11  10  12  14  13  16  15
Sort:
1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20

   Massiv float
19.1 20.2  1.1  3.2  5.1  4.2  7.0  6.9  1.8  0.8  1.7  0.9  1.1  1.0  1.2  1.4
 1.3  1.6  1.5 20.0

 Sort ...<a<b<...
 0.8  0.9  1.0  1.1  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  3.2  4.2  5.1  6.9
 7.0 19.1 20.0 20.2

 Sort ...>a>b>...
20.2 20.0 19.1  7.0  6.9  5.1  4.2  3.2  1.8  1.7  1.6  1.5  1.4  1.3  1.2  1.1
 1.1  1.0  0.9  0.8

   Massiv objects


Sort By Speed:
12  16  20  40  42  52  52  54  58  62


Sort By Year:
1999  2000  2002  2003  2004  2004  2005  2005  2007  2008
Для продолжения нажмите любую клавишу . . .
```

Рис.3. Результат работы программы

Предложено 6 вариантов реализации метода пузырьковой сортировки. Поскольку это самый простой метод сортировки данных, можно было уделить большее внимание оформлению кода, а не реализации самого метода. Последующие методы, как правило, будут предложены только в

вариантах сортировки массивов целых чисел и основное внимание уделено реализации методам сортировки. Однако, простое сравнение с приведенными программами для метода пузырька, позволит выполнить и остальные варианты реализации.

Метод просеивания

Выполняется так же как метод пузырька, но после перестановки элементов величина с меньшим значением передвигается к началу массива, насколько это возможно. Она сравнивается в обратном порядке со всеми предшествующими элементами массива. Если значение меньше, чем у предшествующего элемента массива, то выполняется перестановка. Как только встречается элемент с меньшим значением, то процесс продвижения к началу массива прекращается и нисходящее сравнение возобновляется с той же позиции, с которой начался обратный ход.

Реализация метода просеивания.

```
// Метод просеивания
//
#include "stdafx.h"
#include <iostream>
using namespace std;

void Swap(int *, int *);
int icom_funct1(int , int);
int icom_funct2(int , int);
void sort_sift(int *, int, int (*)(int , int ));


int _tmain(int argc, _TCHAR* argv[])
{
    const int n=20;
    int x[n]={19,20,1,2,3,5,4,7,6,18,8,17,9,11,
10,12,14,13,16,15};
    int i;
    printf("\n");
    for (i=0;i<n;i++) printf(" %d ",x[i]);
    sort_sift(x, n,icom_funct1);// По возрастанию
    printf("\n\n Sort Up:\n");
    for (i=0;i<n;i++) printf(" %d ",x[i]);
    printf("\n");
    sort_sift(x, n,icom_funct2);// По убыванию
    printf("\n Sort Down:\n");
    for (i=0;i<n;i++) printf(" %d ",x[i]);
    printf("\n");
```

```
    return 0;
}

void sort_sift(int *a, int dim, int (*funct_ptr)(int ,
int ))
{
  int i,j,m;
  m=dim-1;
  for (i= 0;i<m;i++)
   if (funct_ptr(a[i],a[i+1])>0)
   {
    Swap(&a[i],&a[i+1]); // перестановка
    /*Обратный ход:
     "проталкиваем" a[i]  в начало списка:*/
   for (j=i;j>0;j--)
     if (funct_ptr(a[j-1],a[j])>0) Swap(&a[j-1],&a[j]);
 }
}


void Swap(int *pa, int *pb)
{
    int t;
    t=*pa;
    *pa = *pb;
    *pb =t;
}

int icom_funct1(int a, int b)
{
 return (a-b);
}

int icom_funct2(int a, int b)
{
 return (b-a);
}
```

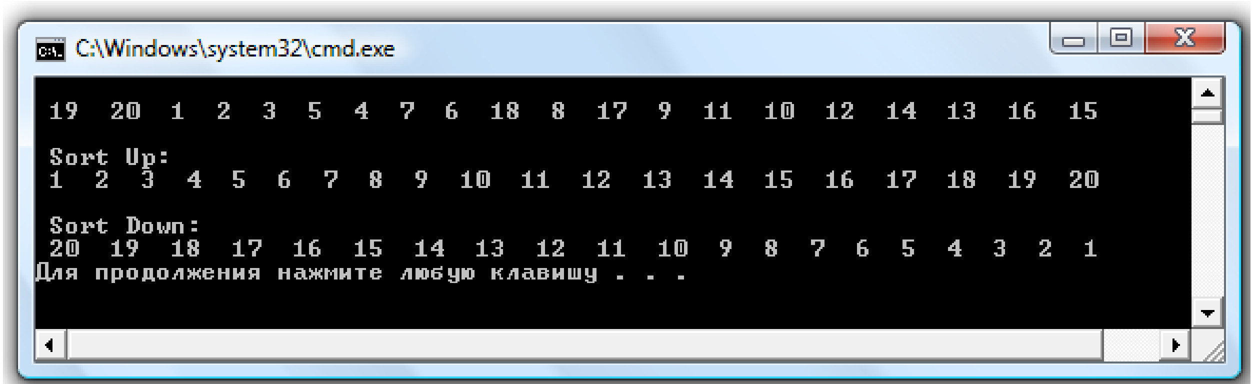


Рис.4. Результат работы программы

Более эффективная (или эффектная) реализация метода просеивания. В предыдущем примере при обратном ходе осуществляли перестановки соседних элементов. Можно этого не делать, а просто перемещать (сдвигать) элементы, освобождая место для “проталкиваемого” в начало элемента

```
void sort_sift(int *x, int dim, int (*sravn)(int, int))
{
      int i,j,t;
      for (i=0;i<dim;i++){
            t=x[i];
            for (j=i-1;(j>=0) && (sravn(x[j],t)>0); j--)
                  x[j+1]=x[j];
            x[j+1]=t;
            }
}
```

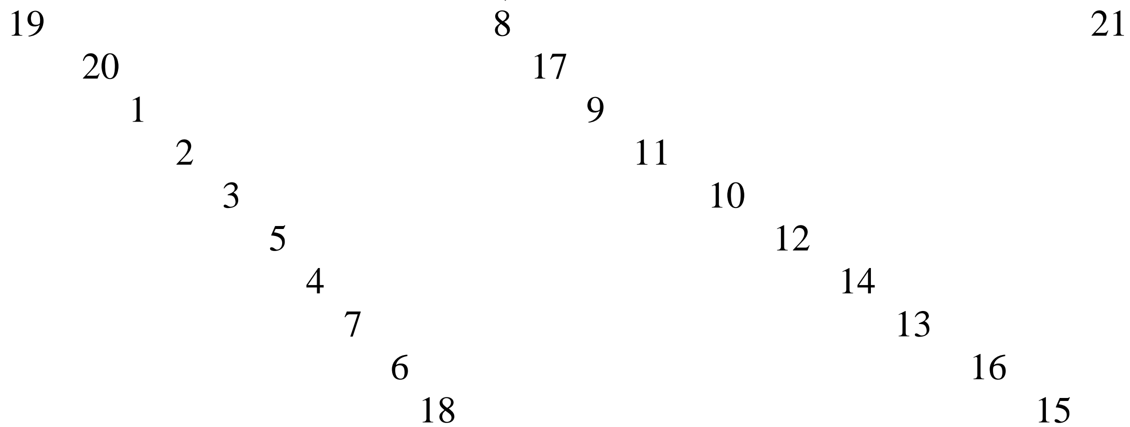
Метод Шелла

Метод предложен в 1959 году и назван по имени автора метода Дональда Шелла (Donald Shell) (https://en.wikipedia.org/wiki/Donald_Shell).
Так же как и метод просеивания, состоит из прямого и обратного хода. Но сравниваются и обмениваются не непосредственные соседи, а элементы, отстоящие на заданном расстоянии. Когда обнаружена перестановка, цепочка вторичных сравнений охватывает те элементы, которые входили в последовательность первичных просмотров.
Каждый последующий просмотр производится с уменьшенным шагом, на последнем просмотре шаг должен равняться 1. Можем использовать следующую процедуру выбора шага. На первом просмотре шаг имеет значение $d = 2^k - 1$, где k выбрано из условия $2^k < n \le 2^{k+1}$. Новый просмотр производим с шагом $d = \left[{(d-1)}/{2} \right]$. Сортировка заканчивается при $d = 0$.

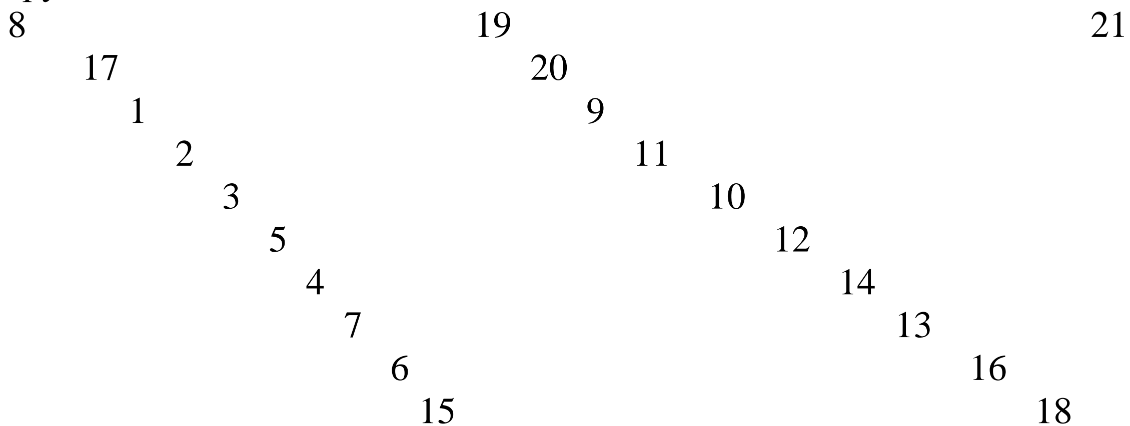
Пример.
`19,20,1,2,3,5,4,7,6,18,8,17,9,11, 10,12,14,13,16,15,21`
`dim=21,  d=dim/2=10`

Проход 1 `d = 10`

Сравниваем группы элементов, отобранные с шагом `d` (в первой группе 3 элемента, в остальных – по два):



Сравнения только внутри групп, если сделана перестановка, то начинаем обратный ход – проталкиваем маленький элемент (внутри группы) к началу группы насколько это возможно.

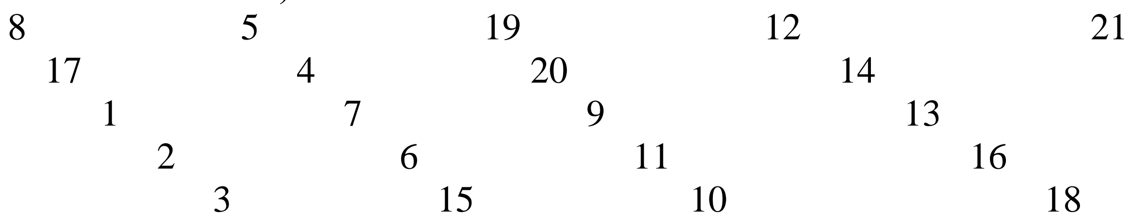


После первого прохода:
8, 17, 1, 2, 3, 5, 4, 7, 6, 15, 19, 20, 9, 11, 10, 12, 14, 13, 16, 18, 21

Проход 2 `d=d/2 = 5`

8, 17, 1, 2, 3, 5, 4, 7, 6, 15, 19, 20, 9, 11, 10, 12, 14, 13, 16, 18, 21

Группы элементов, отобранных с шагом `d=5` (в первой группе 5 элементов, в остальных – по 4):



Производим сравнения и перестановки (как в методе просеивания) только внутри групп

5 8 12 19 21
4 14 17 20
1 7 9 13
2 6 11 16
3 10 15 18

После этого прохода:

5, 4, 1, 2, 3, 8,14, 7, 6, 10, 12, 17, 9, 11, 15, 19, 20, 13, 16, 18, 21

Проход 3

d=d/2 = 5/2 =2

Массив

5, 4, 1, 2, 3, 8,14, 7, 6, 10, 12, 17, 9, 11, 15, 19, 20, 13, 16, 18, 21

Разделим на две группы (с шагом d=2)

5 1 3 14 6 12 9 15 20 16 21
4 2 8 7 10 17 11 19 13 18

Производим сравнения и перестановки внутри каждой из групп.

Результат

1, 2, 3, 4, 5, 7, 6, 8, 9, 10, 12, 11, 14, 13, 15, 17, 16, 18, 20, 19, 21

Последний проход с шагом d=1 окончательно упорядочит массив.

Реализация метода Шелла.

```
//  Метод Шелла
//
#include "stdafx.h"
#include <iostream>
using namespace std;

void Swap(int *, int *);
int icom_funct1(int , int);
int icom_funct2(int , int);
void sort_shell(int *, int, int (*)(int , int ));


int _tmain(int argc, _TCHAR* argv[])
{
    const int n=20;
    int x[n]={19,20,1,2,3,5,4,7,6,18,8,17,9,11,
10,12,14,13,16,15};
    int i;
    printf("\n");
    for (i=0;i<n;i++) printf(" %d ",x[i]);
    sort_shell(x, n,icom_funct1);// По возрастанию
    printf("\n\n Sort Up:\n");
    for (i=0;i<n;i++) printf(" %d ",x[i]);
```

```
    printf("\n");
    sort_shell(x, n,icom_funct2);// По убыванию
    printf("\n Sort Down:\n");
    for (i=0;i<n;i++) printf(" %d ",x[i]);
    printf("\n");
    return 0;
}
void sort_shell(int *x, int dim, int (*sravn)(int, int))
{
    int i,j,d;
    d=dim/2;
    while (d>=1){
      for (i=0;i<dim-d;i++) // прямой ход
         if (sravn(x[i],x[i+d])>0)
         {
              Swap(&x[i],&x[i+d]); // перестановка
              // обратный ход:
              for (j=i;j>=d;j--)
              if (sravn(x[j-d],x[j])>0) Swap(&x[j-d],&x[j]);
         }
         d = d/2;
    }
}

void Swap(int *pa, int *pb)
{
    int t;
    t=*pa;
    *pa = *pb;
    *pb =t;
}

int icom_funct1(int a, int b)
{
 return (a-b);
}

int icom_funct2(int a, int b)
{
 return (b-a);
}
```

Более эффективная (или эффектная) реализация метода Шелла

```
void sort_shell(int *x, int dim, int (*sravn)(int, int))
{
    int i,j,d,t;
    d=dim/2;
```

```
        while (d>=1){
        for (i=d;i<dim;i++){
                t=x[i];
                for (j=i-d;(j>=0) && (sravn(x[j],t)>0); j-=d)
                        x[j+d]=x[j];
                x[j+d]=t;
        }
        d= d/2;
        }
}
```

Быстрая сортировка

1. Выбираем в массиве $a_1,\ldots,a_n$ (случайным образом) какой-нибудь элемент x.
2. Просматриваем массив, двигаясь слева направо, пока не найдем элемент $a_i > x$.
3. Просматриваем список, двигаясь справо налево, пока не найдем элемент $a_j < x$.
4. Меняем местами элементы a_i и a_j.
5. Продолжим процесс просмотра, пока два просмотра не встретятся. В результате массив разделится на две части: левую с ключами, меньшими чем x, и правую — с ключами, большими x.
6. Применяем приведённую процедуру для каждой из полученных частей.

Пример. Упорядочим последовательность чисел: 2, 6, 7, 9, 3, 2, 5, 1, 4. Будем обозначать через ***L*** — левую границу просмотра, через ***R*** — правую границу просмотра, в качестве x возьмем элемент a_k с индексом $k=\left[(L+R)/2\right]$.

Первый просмотр.
***L*=1, *R*=9**, ***x*=*a*₅**

2, 6, 7, 9, **3**, 2, 5, 1, 4 — $a_2 > x$, $a_8 < x$; поменяем их местами:
2, 1, 7, 9, **3**, 2, 5, 6, 4
2, 1, 7, 9, **3**, 2, 5, 6, 4 — $a_3 > x$, $a_6 < x$; поменяем их местами:
2, 1, 2, 9, **3**, 7, 5, 6, 4
2, 1, 2, 9, **3**, 7, 5, 6, 4 — просмотры встретились
2, 1, 2, **3**, 9, 7, 5, 6, 4.
Список разделился на две части: 2, 1, 2, 3 и 9, 7, 5, 6, 4.
Для каждой из частей применим аналогичную процедуру.
Для левой части: 2, 1, 2, 3. ***L*=1, *R*=4**, ***x*=*a*₂**.
2, **1**, 2, 3
1, 2, 2, 3 — левая часть списка упорядочена.
Для правой части: 9, 7, 5, 6, 4. ***L*=1, *R*=5**, ***x*=*a*₃**.
9, 7, **5**, 6, 4 — $a_1 > x$, $a_5 < x$; поменяем их местами:
4, 7, **5**, 6, 9
4, 7, **5**, 6, 9 — просмотры встретились
4, **5**, 7, 6, 9.

Список разделился на две части: 4, 5 и 7, 6, 9. Левая часть упорядочена, а для правой требуется один просмотр:
7, **6**, 9
6, 7, 9.
Соединяя упорядоченные части, получаем весь список:
1, 2, 2, 3, 4, 5, 6, 7, 9 .

Реализация алгоритма. Вариант 1. Использование встроенной функции сортировки MS Visual Studio:

```
void qsort(
   void *base,
   size_t num,
   size_t width,
   int (__cdecl *compare )(const void *, const void *)
);
```

Parameters

base
Start of target array.

num
Array size in elements.

width
Element size in bytes.

compare
Comparison function. The first parameter is a pointer to the key for the search and the second parameter is a pointer to the array element to be compared with the key.

Remarks

The **qsort** function implements a quick-sort algorithm to sort an array of *num* elements, each of *width* bytes. The argument *base* is a pointer to the base of the array to be sorted. **qsort** overwrites this array with the sorted elements. The argument *compare* is a pointer to a user-supplied routine that compares two array elements and returns a value specifying their relationship. **qsort** calls the *compare* routine one or more times during the sort, passing pointers to two array elements on each call:

Рис.5. Фрагмент описания функции, реализующей быструю сортировку, из справочной службы среды MS Visual Studio

Пример. Показано как с помощью встроенной функции MS Visual Studio выполнить сортировку целочисленного массива по возрастанию значений

```
// sort-08.cpp : Быстрая сортировка
//
#include "stdafx.h"
#include <iostream>
using namespace std;
```

```
int icom_funct(const void *px, const void *py);
int (*funct_ptr) (const void *, const void *);

int _tmain(int argc, _TCHAR* argv[])
{
     const int n=20;
     int i;
     int x[n]={19,20,1,2,3,5,4,7,6,18,8,17,9,11,
10,12,14,13,16,15};

     printf("\n");
     for (i=0;i<n;i++) printf(" %d ",x[i]);

     funct_ptr = icom_funct;
     qsort(x,n,sizeof(int),funct_ptr);

     printf("\n\n Sort Up:\n");
     for (i=0;i<n;i++) printf(" %d ",x[i]);
     printf("\n");

     return 0;
}

int icom_funct(const void *px, const void *py)
{
 return ((*(int *) px) - (*(int *) py));
}
```

Реализация метода быстрой сортировки.

```
// sort-08.cpp : Быстрая сортировка
//

#include "stdafx.h"
#include <iostream>
using namespace std;

void Swap(int *, int *);
void quickSortR(int* , int);

int _tmain(int argc, _TCHAR* argv[])
{
     const int n=20;
     int i;
     int x[n]={19,20,1,2,3,5,4,7,6,18,8,17,9,11,
10,12,14,13,16,15};
```

```
     printf("\n");
    for (i=0;i<n;i++) printf(" %d ",x[i]);

     quickSortR(x,n-1);

    printf("\n\n Sort Up:\n");
    for (i=0;i<n;i++) printf(" %d ",x[i]);
     printf("\n");

     return 0;
}


void quickSortR(int * a, int dim) {
// На входе - массив a[], a[dim] - его последний элемент.

  int i = 0, j = dim; // поставить указатели на исходные места
  int temp, x;

  x = a[ dim>>1 ];  // центральный элемент: p=a[dim/2]

  // процедура разделения
  do {
    while ( a[i] < x ) i++;
    while ( a[j] > x ) j--;

    if (i <= j) {
       Swap(&a[i],&a[j]); // перестановка
      i++; j--;
    }
  } while ( i<=j );


  // рекурсивные вызовы, если есть, что сортировать
  if ( j > 0 ) quickSortR(a, j);
  if ( i<dim ) quickSortR(a+i, dim-i);
}
```

Пример. Вариант с функцией сравнения

```
// sort-08.cpp : Быстрая сортировка
//
#include "stdafx.h"
#include <iostream>
using namespace std;

void Swap(int *, int *);
void quickSortR(int *, int, int (*)(int , int ));
int icom_funct1(int , int);
int icom_funct2(int , int);
```

```
int _tmain(int argc, _TCHAR* argv[])
{
     const int n=20;
     int i;
     int x[n]={19,20,1,2,3,5,4,7,6,18,8,17,9,11,
10,12,14,13,16,15};

     printf("\n");
    for (i=0;i<n;i++) printf(" %d ",x[i]);

   quickSortR(x,n-1,icom_funct1);

    printf("\n\n Sort Up:\n");
    for (i=0;i<n;i++) printf(" %d ",x[i]);
     printf("\n");

   quickSortR(x,n-1,icom_funct2);

    printf("\n\n Sort Down:\n");
    for (i=0;i<n;i++) printf(" %d ",x[i]);
     printf("\n");

     return 0;
}

void quickSortR(int * a, int dim, int (*funct_ptr)(int , int ))
{

  int i = 0, j = dim; // поставить указатели на исходные места
  int temp, x;

  p = a[dim>>1 ];   // центральный элемент

  // процедура разделения
  do {
    while ( funct_ptr(x,a[i])>0) i++;
    while ( funct_ptr(a[j],x)>0) j--;

    if (i <= j) {
       Swap(&a[i],&a[j]); // перестановка
      i++; j--;
    }
  } while ( i<=j );


  // рекурсивные вызовы, если есть, что сортировать
  if ( j > 0 ) quickSortR(a, j,funct_ptr);
  if ( i<dim ) quickSortR(a+i, dim-i,funct_ptr);
}


void Swap(int *pa, int *pb)
```

```
{
      int t;
      t=*pa;
      *pa = *pb;
      *pb =t;
}

int icom_funct1(int a, int b)
{
 return (a-b);
}

int icom_funct2(int a, int b)
{
 return (b-a);
}
```

Пирамидальная сортировка

Пирамидальная сортировка предложена в 1964 году Дж. Уильямсом. Основана на использовании бинарного сортирующего дерева (пирамиды).

Представление массива в виде дерева

В качестве примера будем использвать массив чисел из книги [5]. Это позволит “проложить мостик” с этой выдающейся книгой и пропустить ряд деталей, связанных с оценкой эффективности алгоритма пирамидальной сортировки

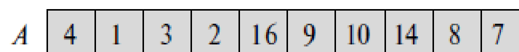


Рис.6. Массив элементов

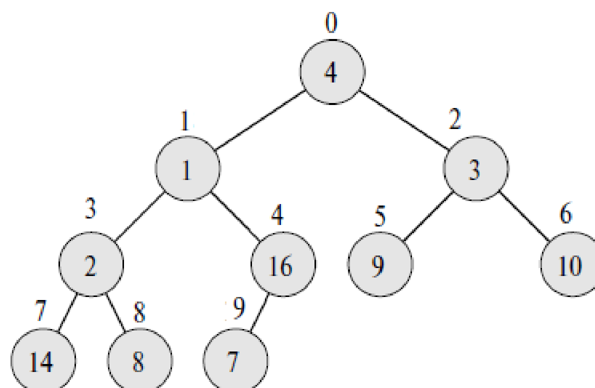


Рис.7. Массив записан в виде дерева. Значения элементов массива записаны в кружках, а индексы указаны рядом с кружками

Пирамида

Пирамида (в английском используется термин binary heap) определяется как структура данных, представляющая собой объект-массив, который можно рассматривать как почти полное бинарное дерево. Каждый узел этого дерева соответствует определенному элементу массива. На всех уровнях, кроме, может быть, последнего, дерево полностью заполнено (заполненным считается уровень, который содержит максимально возможное количество узлов). Последний уровень заполняется слева направо до тех пор, пока в массиве не закончатся элементы. Массив, по которому построена пирамида, можно рассматривать как объект с атрибутам length [A], содержащим количество элементов массива, и атрибутом heap_size [A], значением которого является число элементов пирамиды, содержащихся в массиве A. В корне дерева находится элемент A[0], а дальше оно строится по правилу: если какому-то узлу соответствует индекс i, то индекс его родительского узла вычисляется с помощью представленной функции PARENT(i), индекс левого дочернего узла – с помощью функции LEFT(i), а индекс правого дочернего узла – с помощью функции RIGHT(i).

```
int Parent(int i)
{
return (i-1)/2;
}

int Left(int i)
{
return 2*i+1;
}

int Right(int i)
{
return 2*i+2;
}
```

Различают два вида бинарных пирамид: неубывающие и невозрастающие. Свойство невозрастающих пирамид заключается в том, что для каждого отличного от корневого узла с индексом i выполняется неравенство:

```
A[PARENT (i)] >= A[i] .
```

Другими словами, значение узла не превышает значение родительского по отношению к нему узла. Таким образом, в невозрастающей пирамиде самый большой элемент находится в корне дерева, а значения узлов поддерева, берущего начало в каком-то элементе, не превышают значения самого этого элемента. Принцип организации неубывающей пирамиды прямо противоположный. Свойство неубывающих пирамид заключается в том, что

для всех отличных от корневого узлов с индексом i выполняется такое неравенство:

A[PARENT (i)] <= A[i].

Таким образом, наименьший элемент такой пирамиды находится в ее корне.

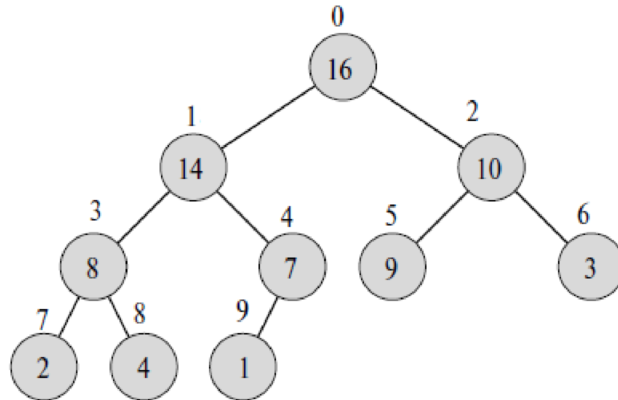


Рис.8. Элементы того же массива, но уже в виде пирамиды

С помощью функции `Heapify()` поддерживается основное свойство пирамиды. Параметрами этой функции являются массив данных и индекс `i`. Предполагается, что поддеревья с корнями `Left(i)` и Right(i) уже обладают основным свойством пирамиды. Вершина `i` переставляется с большим из её “детей” и т.д., пока элемент с индексом `i` не “погрузится” до нужного места.

```
void Heapify(int a[],int i,int heapsize)
{
     int l=Left(i);
     int r=Right(i);
     int largest =i;
     if (l<heapsize && (a[l]>a[i])) largest=l;
     if (r<heapsize && (a[r]>a[largest])) largest=r;
     if (largest != i) // обмен a[i] и a[largest]
     {
          int t= a[i]; a[i]=a[largest]; a[largest]=t;
          Heapify(a,largest,heapsize);
     }
}
```

На следующем рисунке показан пример работы функции `Heapify()`

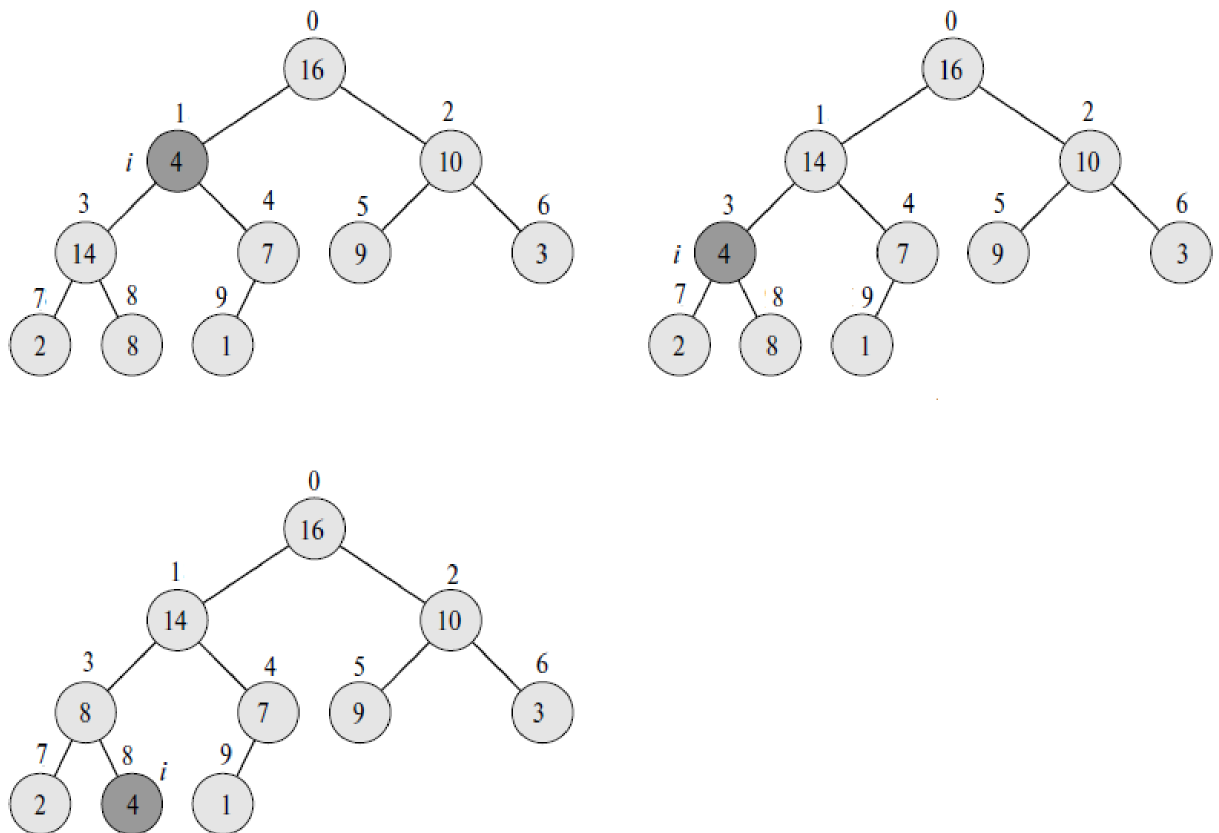


Рис. 9. В вершине i=1 основное свойство нарушено

Построение пирамиды

Если дан массив `a[0]`, `a[1]`, …, `a[n-1]`, то записать его в виде пирамиды можно следующим образом. Начиная с нижних вершин, применяем функцию `Heapify()`.

```
void BuildHeap(int a[],int size)
{
    int i= (size-1)/2;
    while (i>=0){
        Heapify(a,i,size);
        i--;
    }
}
```

На следующем рисунке показан пример работы функции `BuildHeap()`

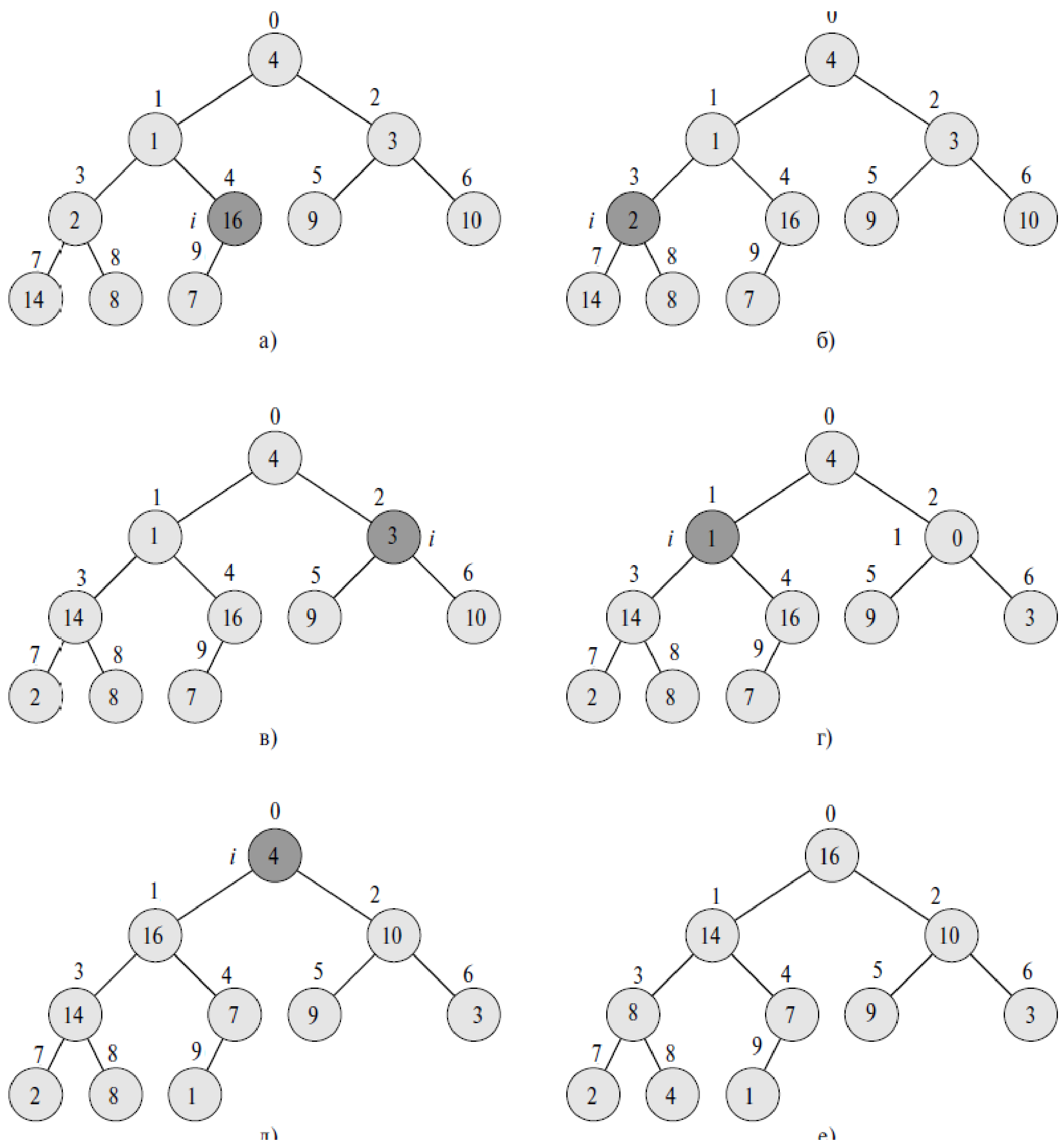


Рис. 10. Построение пирамиды

Сортировка с помощью пирамиды

Алгоритм сортировки с помощью пирамиды состоит из двух частей. Сначала вызывается функция `BuildHeap()`, после выполнения которой массив становится пирамидой. Далее, поскольку максимальный элемент теперь находится в вершине пирамиды, т.е. равен `a[0]`, то его следует поменять с последним элементом массива `a[n-1]`. Затем уменьшить размер пирамиды на 1 и восстановить основное свойство в корневой вершине с помощью функции `Heapify()`. Отметим, что поддеревья `Left(0)` и `Right(0)` не утратили основного свойства пирамиды. После этого шага в корне пирамиды будет находиться максимальный из оставшихся элементов массива. Алгоритм работает пока в пирамиде не останется один элемент.

Реализация метода сортировки с помощью пирамиды

```
//
#include "stdafx.h"
#include <iostream>

using namespace std;

int Parent(int i) {return (i-1)/2;}
int Left(int i) {return 2*i+1;}
int Right(int i) {return 2*i+2;}
void Heapify(int a[],int i,int heapsize);
void BuildHeap(int a[],int size);
void HeapSort(int a[],int size);


int _tmain(int argc, _TCHAR* argv[])
{
    const int n=20;
    int i;
    int x[n]={19,20,1,2,3,5,4,7,6,18,8,17,9,11,
10,12,14,13,16,15};

    printf("\n");
    for (i=0;i<n;i++) printf(" %d ",x[i]);

    HeapSort(x,n);

    printf("\n\n Sort Up:\n");
    for (i=0;i<n;i++) printf(" %d ",x[i]);
    printf("\n");

    system("pause");
    return 0;
}

void Heapify(int a[],int i,int heapsize)
{
    int l=Left(i);
    int r=Right(i);
    int largest =i;
    if (l<heapsize && (a[l]>a[i])) largest=l;
    if (r<heapsize && (a[r]>a[largest])) largest=r;
```

```
    if (largest != i) // обмен a[i] и a[largest]
    {
        int t= a[i]; a[i]=a[largest]; a[largest]=t;
        Heapify(a,largest,heapsize);
    }
}

void BuildHeap(int a[],int size)
{
    int i= (size-1)/2;
    while (i>=0){
        Heapify(a,i,size);
        i--;
    }
}

void HeapSort(int a[],int size)
{
    BuildHeap(a,size);
    int heapsize=size;
    for (int i=size-1;i>0;i--)
    {
        // меняем a[0] и a[i]
        int t=a[0]; a[0]=a[i];a[i]=t;
        heapsize--;
        Heapify(a,0,heapsize);
    }
}
```

```
C:\LEK-Занятия\Projects\Pyramid01\Debug\Pyramid01.exe

 19  20  1  2  3  5  4  7  6  18  8  17  9  11  10  12  14  13  16  15

 Sort Up:
 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20
Для продолжения нажмите любую клавишу . . .
```

Рис.11. Результат работы программы сортировки пирамидой

Сортировка слиянием

Пример работы алгоритма на массиве 3 7 8 2 4 6 1 5..

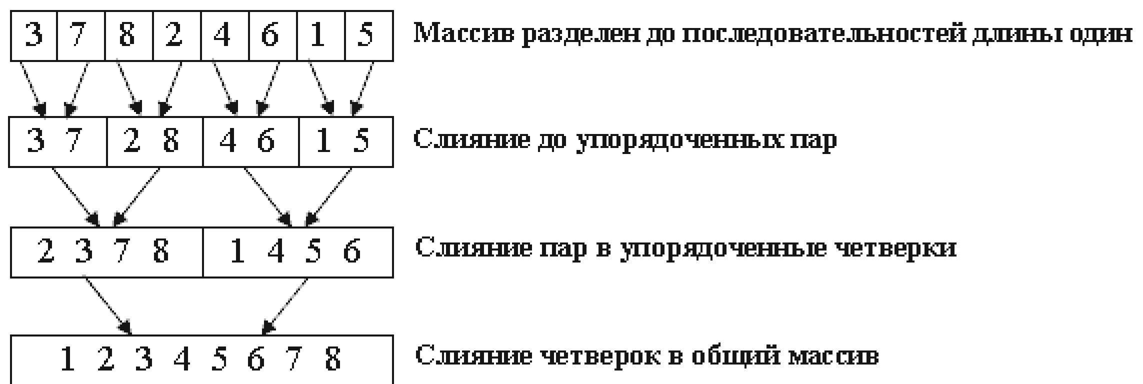


Рис.12. Слияние числового массива

Рекурсивный алгоритм обходит получившееся дерево слияния в прямом порядке. Каждый уровень представляет собой *проход* сортировки слияния - операцию, полностью переписывающую массив.

Реализация метода сортировки слиянием

```
// sort-merge.cpp : Сортировка слиянием
//

#include "stdafx.h"
#include <iostream>
using namespace std;

void mergeSort(int numbers[], int , int );
void merge(int a[], int, int, int);

int _tmain(int argc, _TCHAR* argv[])
{
     const int n=20;
     int i;
     int x[n]={19,20,1,2,3,5,4,7,6,18,8,17,9,11,
10,12,14,13,16,15};

     printf("\n");
    for (i=0;i<n;i++) printf(" %d ",x[i]);
//
    mergeSort(x, 0, n-1);
//
    printf("\n\n Sort Up:\n");
    for (i=0;i<n;i++) printf(" %d ",x[i]);
     printf("\n");

     return 0;
}

void mergeSort(int a[], int lb, int ub) {
  int split;     // индекс, по которому делим массив

  if (lb < ub) {  // если есть более 1 элемента
```

```
    split = (lb + ub)/2;

    mergeSort(a, lb, split);    // сортировать левую половину
    mergeSort(a, split+1, ub); // сортировать правую половину
    merge(a, lb, split, ub);  // слить результаты в общий массив
  }
}

void merge(int a[], int lb, int split, int ub)
{
// Слияние упорядоченных частей массива в буфер temp
// с дальнейшим переносом содержимого temp в a[lb]...a[ub]

  // текущая позиция чтения из первой последовательности
  // a[lb]...a[split]
  int pos1=lb;

  // текущая позиция чтения из второй последовательности
a[split+1]...a[ub]
  int pos2=split+1;

  // текущая позиция записи в temp
  int pos3=0;

  int *temp = new int[ub-lb+1];

  // идет слияние, пока есть хоть один элемент в каждой
  // последовательности
  while (pos1 <= split && pos2 <= ub) {
    if (a[pos1] < a[pos2])
      temp[pos3++] = a[pos1++];
    else
      temp[pos3++] = a[pos2++];
  }

  // одна последовательность закончилась -
  // копировать остаток другой в конец буфера
  while (pos2 <= ub)   // пока вторая последовательность непуста
    temp[pos3++] = a[pos2++];
  while (pos1 <= split) // пока первая последовательность
   //     непуста
    temp[pos3++] = a[pos1++];

  // скопировать буфер temp в a[lb]...a[ub]
  for (pos3 = 0; pos3 < ub-lb+1; pos3++)
    a[lb+pos3] = temp[pos3];
}
```

Внешняя сортировка

Если сортируемый файл с данными настолько велик, что, в силу ограничений оперативной памяти, нельзя создать соответствующий массив, то прибегают к методам сортировки, для которых используют термин *внешняя сортировка* (external sorting). Прежде чем перейти к изложению методов сортировки, напомним основы работы с файлами в С и С++.

Напоминание по работе с файлами

Схема работы с файлами на языке С

В языке С имеется специальная библиотека функций `stdio` – стандартная библиотека ввода-вывода (standard input/output library). Схема работы с файлами с помощью этой библиотеки следующая.

1. Каждый открытый файл должен иметь указатель типа `FILE`, используемый для ссылок на файл. Функции С работают не с именами файлов, а с указателями на файл.
2. С помощью функции `fopen()` указатель на файл связывается с физическим файлом на диске и указывается режим открытия файла (напр., чтение или запись).
3. После выполнения операция с файлом, необходимо вызвать функцию закрытия файла `fclose()`.

Как записать информацию в текстовый файл – простой пример

Пример. Запись в файл (как принято в С).

```
// Пример записи в текстовый файл
//
#include "stdafx.h"
#include <stdio.h>
#include <iostream>
using namespace std;
FILE *f, *fopen();

int _tmain(int argc, _TCHAR* argv[])
{
int i,x;
   f=fopen("c:\\tmp\\result.txt","w");
   for(i=0;i<100;i++)
     {
      x=i*2;
      fprintf(f," %d ",x);
   }
   fclose(f);
    return 0;
}
```

Как прочитать информацию из файла

Пример. Чтение данных из файла

```
// Чтение из файла
//
#include "stdafx.h"
#include <stdio.h>
#include <iostream>
using namespace std;
FILE *f, *fopen();

int _tmain(int argc, _TCHAR* argv[])
{
  int i,t;
  int x[10];
  f=fopen("c:\\tmp\\dannye.txt","r");
  for(i=0;i<10;i++)
     {
       fscanf(f,"%d",&t);
       x[i]=t;
       cout<<"\n x="<<x[i];
      }
       fclose(f);
    return 0;
}
```

Пример. Чтение всей информации из файла. С помощью функции `feof()` можно определить кончились данные в файле или нет

```
// Чтение из файла
//
#include "stdafx.h"
#include <stdio.h>
#include <iostream>
using namespace std;
FILE *f, *fopen();

int _tmain(int argc, _TCHAR* argv[])
{
  int i,t;
  int x[10];
  f=fopen("c:\\tmp\\dannye.txt","r");
  i = 0;
  while (!feof(f))
     {
```

```
        fscanf(f,"%d",&t);
        x[i]=t;
        cout<<"\n x="<<x[i++];
       }
        fclose(f);
     return 0;
}
```

Схема работы с файлами на языке С++

1. Добавить инструкцию
   `#include <fstream>`
2. Создать объект класса `ofstream` или `ifstream` и указать в конструкторе этого класса имя файла
   `char fileOut[]="c:\\tmp\\outtest.txt";`
   `ofstream fout(fileOut);` – если записываем и

   `char fileIn[]="c:\\tmp\\intest.txt";`
   `ifstream f_in(fileIn);` – если читаем
3. После выполнения операция с файлом, необходимо вызвать метод `close()`.

Как записать информацию в текстовый файл

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
      char fileOut[]="c:\\tmp\\outtest.txt";
// Результирующий файл
      ofstream fout(fileOut);
       int x=1000;
       fout<<x; // записываем
       fout.close(); // закрываем
    return 0;
}
```

Как прочитать информацию из файла

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
using namespace std;
```

```
int _tmain(int argc, _TCHAR* argv[])
{
     char fileIn[]="c:\\tmp\\intest.txt";
// Результирующий файл
     ifstream f_in(fileIn);
      int x;
      f_in>>x; //  читаем
      f_in.close(); // закрываем
     return 0;
}
```

Пояснение работы метода `eof()`

```
// TestEof1.cpp:
//

#include "stdafx.h"
#include <stdio.h>
#include <iostream>
#include <fstream>
using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
     char file1[]="c:\\tmp\\test1.txt";
// В файле всего одна запись
     ifstream f1(file1);
     cout<<"\n f1.eof()= "<<f1.eof()<<"\n";// eof=0
     int x;
     f1>>x; // eof=1
     cout<<"\n f1.eof()= "<<f1.eof()<<"  x= "<<x<<"\n";
     f1>>x;
     cout<<"\n f1.eof()= "<<f1.eof()<<"  x= "<<x<<"\n";
     f1.close();
     return 0;
}
```

Файл `test1.txt` (в файле записано одно число, обращаем внимание на то, что после числа нет пробелов!):

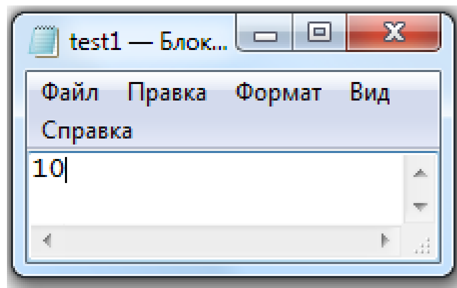


Рис.13. Файл с одной записью, отметим, что после числа нет пробелов

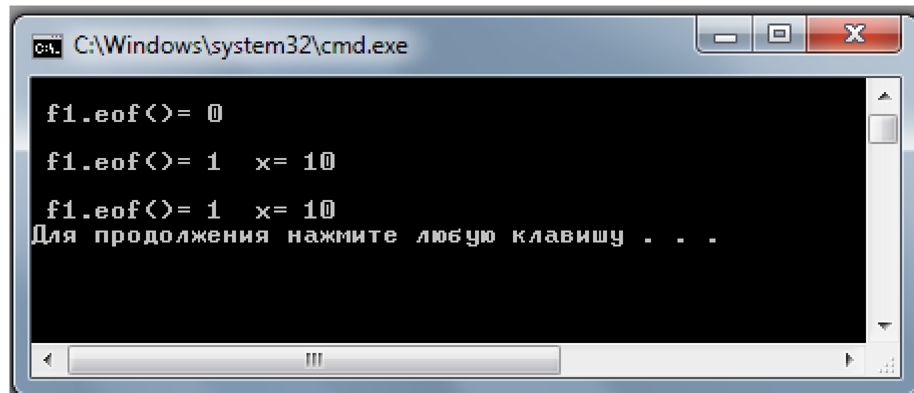


Рис.14. Результат работы программы. После чтения из файла единственного записанного в нем числа функция `eof()` возвращает значение 1

Файл `test1.txt` (в файле записано одно число, обращаем внимание на то, что после числа добавлены пробелы):

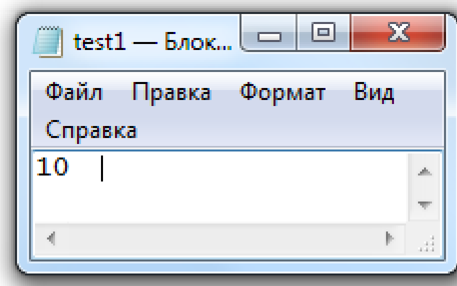


Рис.15. Файл с одной записью, отметим, что после числа по крайней мере один пробел

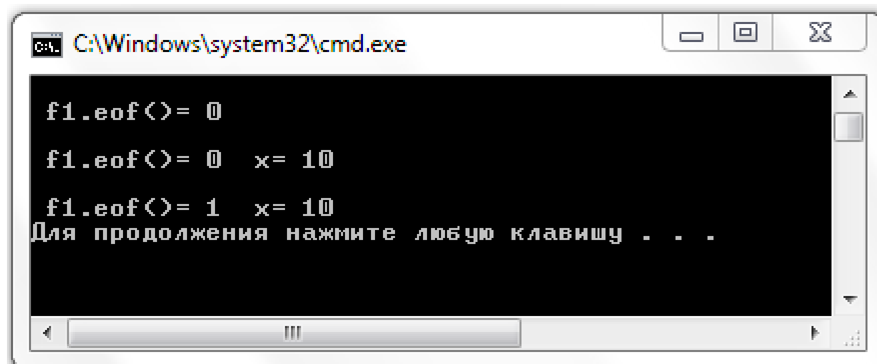


Рис.16. Результат работы программы. После чтения из файла единственного записанного в нем числа функция `eof()` возвращает значение 0, т.е. в этом случае не указывает на отсутствие числовых данных.

Можем сделать вывод, что при работе с данными нужно удалять пробелы в конце файла.

Простой вопрос. Если файл совсем пустой (не только нет чисел, но и пробелов!).

```
int _tmain(int argc, _TCHAR* argv[])
{
    char file1[]="c:\\tmp\\test1.txt";  // Файл пустой
    ifstream f1(file1);
    cout<<"\n f1.eof()= "<<f1.eof()<<"\n";
    int x;
    f1>>x;
    cout<<"\n f1.eof()= "<<f1.eof()<<"  x= "<<x<<"\n";;
    f1>>x;
    cout<<"\n f1.eof()= "<<f1.eof()<<"  x= "<<x<<"\n";;
    f1.close();
    return 0;
}
```

```
C:\Windows\system32\cmd.exe

f1.eof()= 0

f1.eof()= 1  x= -858993460

f1.eof()= 1  x= -858993460
Для продолжения нажмите любую клавишу . . .
```

Рис.16. Пока не попытались прочитать, не смогли узнать об этом!

Модель внешней сортировки

Абстрактная модель сортировки построена на предположении, что сортируемый файл не помещается в оперативной памяти компьютера. В распоряжении имеется:

- N записей на внешнем устройстве, сортировку которых необходимо выполнить;
- объем оперативной памяти, достаточный для хранения M записей;
- $2P$ внешних устройств, которыми можно пользоваться во время сортировки.

Большинство известных методов внешней сортировки основаны на принципах распределения и слияния. Во время первого прохода по файлу производится его разбиение на блоки (размером меньшим, чем объем оперативной памяти) и выполняются операции сортировки этих блоков. Затем отсортированные блоки *сливают* в новый файл.

Естественное слияние

Упорядоченные серии

Серией называется подпоследовательность записей `a[i], a[i+1], ..., a[j]`, такая, что `a[i]<=a[i+1]<= ...<= a[j]`, `a[i] < a[i-1]` и `a[j] > a[j+1]`. Например, 5, 8, 23, 44, 65, 1, 6, 33 содержит 2 серии.

Метод естественного слияния

Метод естественного слияния основывается на распознавании серий, их распределении и последующем слиянии.

Сортировка выполняется за несколько шагов, в каждом из которых выполняются:

– распределение файла A по файлам B и C
– слияние файлов B и C в файл A.

При распределении распознается первая серия элементов и переписывается в файл B, вторая серия записывается в файл C, следующая серия снова в B и т.д.

При слиянии первая серия записей файла B сливается с первой серией файла C, вторая серия B со второй серией C и т.д.

Если просмотр одного файла заканчивается раньше, чем просмотр другого (по причине разного числа серий), то остаток недопросмотренного файла целиком копируется в конец файла A. Процесс завершается, когда в файле A остается только одна серия.

Сортировка естественным слиянием. Вариант 1. Слияние файлов, а не серий!

Сортировка естественным слиянием. Вариант 2.

```
// SortMergeFile2.cpp
//
#include "stdafx.h"
#include <stdio.h>
#include <iostream>
using namespace std;
FILE *fopen();
void MergeSort(char *fileA);
void Distribute(char *fileA, char *fileB, char *fileC,
               int &sB,int &sC);
void Merge(char *fileA, char *fileB, char *fileC);
void show_file(char *fileA);


int _tmain(int argc, _TCHAR* argv[])
{
     char fileA[]="c:\\tmp\\A.txt"; // Файл с данными
     show_file(fileA);
     MergeSort(fileA);
     printf("\n After Sorting \n");
     show_file(fileA);
     return 0;
}

void MergeSort(char *fileA)
{
```

```
     char fileB[]="B.txt"; // Вспомогательный файл
     char fileC[]="C.txt"; // Вспомогательный файл
     int sB, sC;
     sB=sC=2;// начальные значения (произвольно)
     // sB - число переходов с ленты B
     // sC - число переходов с ленты C

     while ((sB>=1) && (sC>=1))
// закончим, когда останется одна серия (переходов не будет)
     {
       // распределение по лентам (файлам):
       Distribute(fileA,fileB,fileC,sB,sC);// sB, sC - по ссылке

      // Слияние:
       Merge(fileA,fileB,fileC);
     }//1

     }//MergeSort()

void Distribute(char *fileA, char *fileB, char *fileC,
               int &sB,int &sC)
{
     FILE *fA, *fB, *fC;
     int  x, y;
     char wher;
     // распределение по лентам (файлам):
     fA=fopen(fileA,"r");
     fB=fopen(fileB,"w");
     fC=fopen(fileC,"w");
     wher='B';
     sB=sC=0;
     fscanf(fA,"%d",&x);
     fprintf(fB," %d ",x);
     while (!feof(fA))//пока не закончился файл  fA
        {
          fscanf(fA,"%d",&y);
          if (y<x) // конец серии
            {
             switch (wher)
               { // меняем ленты (файлы)
                  case 'B': wher='C'; sB++; break;
                  case 'C': wher='B'; sC++; break;
               }
            }// if
          switch (wher)
            { // запись на ленты (файлы)
              case 'B': fprintf(fB," %d",y); break;
              case 'C': fprintf(fC," %d",y); break;
            }
            x=y;
          }
          fclose(fA);
          fclose(fB);
```

```
          fclose(fC);
}//Distribute()

void Merge(char *fileA, char *fileB, char *fileC)
{
     FILE *fA, *fB, *fC;
     int  x, y;
     // Слияние:
       fA=fopen(fileA,"w");
      fB=fopen(fileB,"r");
       fC=fopen(fileC,"r");

       fscanf(fB,"%d",&x);
       fscanf(fC,"%d",&y);
     while ((!feof(fB)) && (!feof(fC))) /* пока не прочитаем
                                          файлы fB и fC */
        {       //3
           if (x<=y)
            {
             fprintf(fA," %d",x);
             fscanf(fB,"%d",&x);
            }
            else
          {
             fprintf(fA," %d",y);
             fscanf(fC,"%d",&y);
            }
         } // 3- while

     while (!feof(fB))  // остаток файла fB
        {
            fprintf(fA," %d",x);
            fscanf(fB,"%d",&x);
         }
     while (!feof(fC))   // остаток файла fC
        {
            fprintf(fA," %d",y);
            fscanf(fC,"%d",&y);
         }
        fclose(fA);
        fclose(fB);
        fclose(fC);
}//Merge()

void show_file(char *fileA)
{
     FILE *fA;
     int a;
     fA=fopen(fileA,"r");
     printf("\n %s \n",fileA);
     while (!feof(fA))
       {
             fscanf(fA,"%d",&a);
```

```
                printf("\n %d",a);
            }
    fclose(fA);
}
```

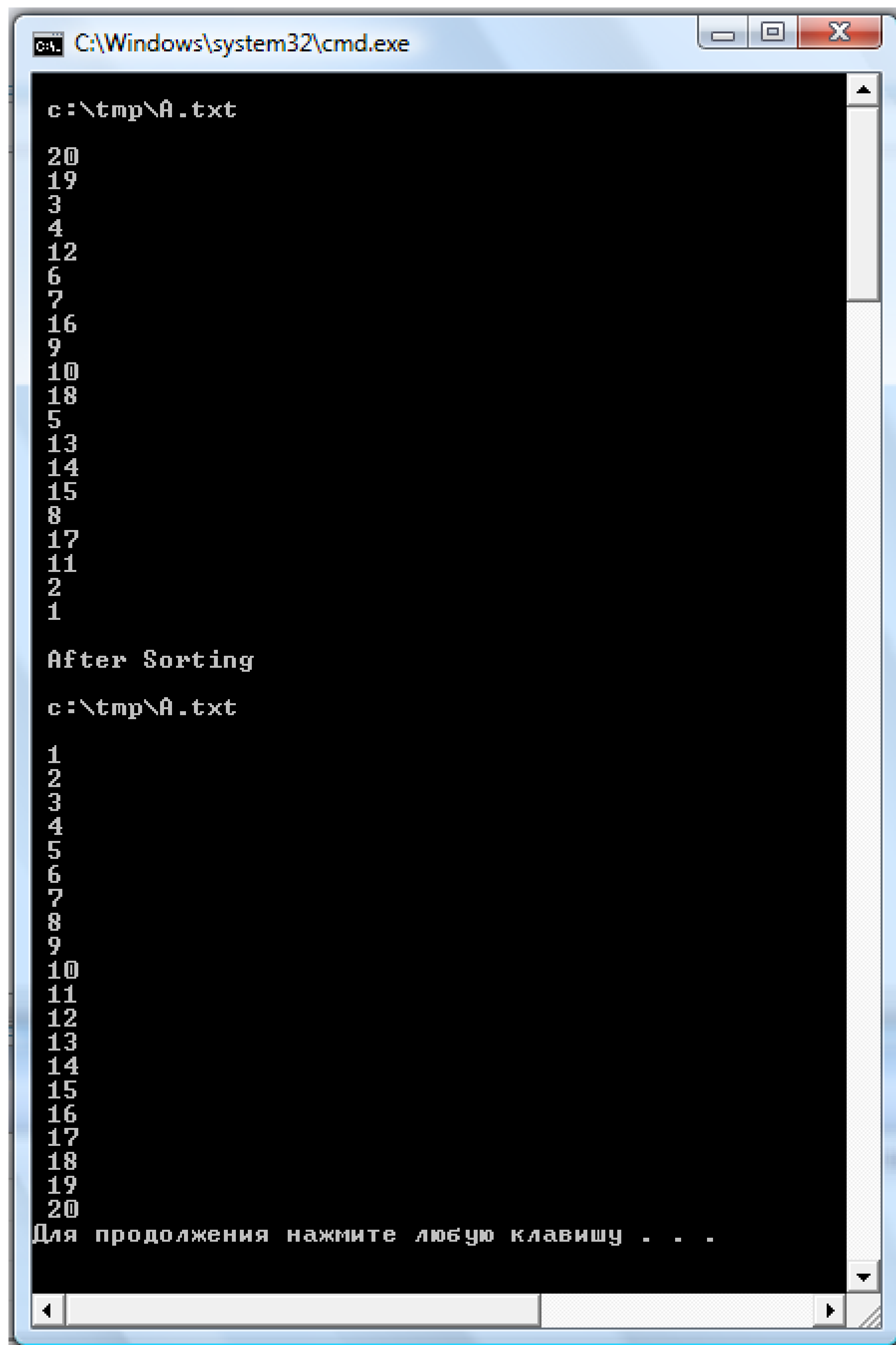


Рис.17. Результат работы программы

Пояснение. Распределение по сериям

Как пример рассмотрим файл

fileA: 19, 20, 1, 2, 3, 5, 4, 7, 6, 18, 8, 17, 9, 11, 10, 12, 14, 13, 16, 15

серии поочередно будут записаны в файлы:

f1: 19, 20, 4, 7, 8, 17, 10, 12, 14, 15

f2: 1, 2, 3, 5,   6, 18,   9, 11,   13, 16

Распределение по сериям.

```
// DistributeTest00.cpp: Распределение файла на серии
//
#include "stdafx.h"
#include <stdio.h>
#include <iostream>
#include <fstream>
using namespace std;
void Distribute(char *fileA, char *file1, char *file2,
                int &s1,int &s2);


int _tmain(int argc, _TCHAR* argv[])
{
     char fileA[]="c:\\tmp\\A.txt"; // Файл с данными
     char file1[]="c:\\tmp\\distr1.txt"; //
    char file2[]="c:\\tmp\\distr2.txt"; //
     int s1,s2;
     Distribute(fileA,file1,file2,s1,s2);
     cout<<"\n s1= "<<s1<<" s2 = "<<s2<<"\n";
     return 0;
}

void Distribute(char *fileA, char *file1, char *file2,
                int &s1,int &s2)
{
     int  x, y;
     char wher;
     // распределение по лентам (файлам):
     ifstream fA(fileA);
    ofstream f1(file1);
     ofstream f2(file2);

     wher='1';
     s1=s2=0;
     fA>>x;
     f1<<x;
     while (!fA.eof())//пока не закончился файл  fA
        {
          fA>>y;
            if (y<x) // конец серии
            {
            switch (wher)
               { // меняем ленты (файлы)
                 case '1': wher='2'; s1++; break;
                 case '2': wher='1'; s2++; break;
               }
            }// if
          switch (wher)
            { // запись на ленты (файлы)
```

```
                case '1': f1<<" "<<y; break;
                case '2': f2<<" "<<y; break;
            }
            x=y;
        }
        fA.close();
        f1.close();
        f2.close();
}
```

Поскольку начинаем запись серий на ленту **1**, затем переходим на ленту **2**, затем очередная серия – снова на ленту **1**, то замечаем, что четный переход – одна лента, нечетный – другая. Если взять переменную **s** с начальным значением **0**, и увеличивать ее значение на **1** при каждом переходе, то при четных **s** должны записывать серию на ленту **1**, а при нечетных – на ленту **2**. Этот вариант распределения представлен на следующем листинге.

Распределение по сериям. Еще один вариант.

```
// DistributeTest01.cpp: Распределение файла на серии
//

#include "stdafx.h"
#include <stdio.h>
#include <iostream>
#include <fstream>
using namespace std;
void Distribute(char *fileA, char *file1, char *file2, int &s);


int _tmain(int argc, _TCHAR* argv[])
{
     char fileA[]="c:\\tmp\\A.txt"; // Файл с данными
     char file1[]="c:\\tmp\\distr01.txt"; //
     char file2[]="c:\\tmp\\distr02.txt"; //
     int s;
     Distribute(fileA,file1,file2,s);
     cout<<"\n  Change tapes = "<<s<<"\n";
     return 0;
}

void Distribute(char *fileA, char *file1, char *file2, int &s)
{
     int  x, y;
     // распределение по лентам (файлам):
     ifstream fA(fileA);
     ofstream f1(file1);
     ofstream f2(file2);

     s=0; // s-четное, пишем на ленту 1, а при нечетном - на
ленту 2
```

```
    fA>>x;
    f1<<x;
    while (!fA.eof())//пока не закончился файл  fA
        {
          fA>>y;
            if (y<x) s++; // конец серии-переходим на другую
ленту
         // запись на ленты (файлы):
            if (s%2 == 0) // при четном s запись на ленту 1 (f1)
                 f1<<" "<<y;
            else // при нечетном s запись на ленту 2 (f2)
                 f2<<" "<<y;
            x=y;
          }
          fA.close();
          f1.close();
          f2.close();
}
```

Процесс слияния файлов

Например,
f1: 2, 4, 6, 8, 10, 12, 14, 16, 18, 20
f2: 1, 3, 5, 7, 9, 11, 13, 15, 17, 19
Результат: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

Реализация слияния файлов

Еще один вариант.

```
// MergeFile01.cpp: Слияние файлов
//

#include "stdafx.h"
#include <stdio.h>
#include <iostream>
#include <fstream>
using namespace std;

void Save_fout(ofstream &f_out, ifstream &f_in, int &x, bool
&endf);
void Merge(char *file1, char *file2, char *fileOut);

int _tmain(int argc, _TCHAR* argv[])
{
     char file1[]="c:\\tmp\\01.txt"; // Файл с данными
     char file2[]="c:\\tmp\\02.txt"; // Файл с данными
     char fileOut[]="c:\\tmp\\outtest0102.txt"; // Результирующий
файл

     // Слияние:
      Merge(file1, file2, fileOut);
```

```
    return 0;
}

 void Save_fout(ofstream &f_out, ifstream &f_in, int &x, bool
&endf)
 {
       if(!f_in.eof())
                  {
                       f_out<<" "<<x; // записываем

                       f_in>>x; // и читаем
                  }
                  else
                  {
                        f_out<<" "<<x; // только записываем
                        endf=1;
                  }
 }

void Merge(char *file1, char *file2, char *fileOut)
{
 ofstream fout(fileOut);
 ifstream f1(file1);
 ifstream f2(file2);
 int a1, a2;
 bool  endf1; // Признак:
        // endf1=1: f1 прочитан, все элементы записаны в fout
 bool  endf2; // Аналогично для f2
 if(!f1.eof()) // f1 - не пустой
  {f1>>a1; endf1=0;}
 else endf1=1;
 if(!f2.eof()) // f2 - не пустой
  {f2>>a2; endf2=0;}
 else endf2=1;
 // Слияние:
 while (!endf1 && !endf2) // пока оба файла не пусты
  {
   // Что записывать: a1 или a2?
   if(a1<=a2) // a1 -> fout
       Save_fout(fout, f1,a1,endf1);
  else // a2 -> fout
       Save_fout(fout, f2,a2,endf2);
  } // while
  // Теперь остатки одного из файлов:
  while(!endf1)
     Save_fout(fout, f1,a1,endf1);
  while(!endf2)
     Save_fout(fout, f2,a2,endf2);
  fout.close();
  f1.close();
  f2.close();
}
```

Слияние серий

Например,
f1: 2, 4, 6, 10, 8, 12, 16, 14, 18, 20
f2: 1, 3, 5, 7, 11, 9, 13, 17, 15, 19
Результат: 1, 2, 3, 4, 5, 6, 7, 10, 11, 8, 9, 12, 13, 16, 17, 14, 15, 18, 19, 20

Вариант со слиянием серий

```
// MergeSeries1.cpp:  слияние серий
//
#include "stdafx.h"
#include <stdio.h>
#include <iostream>
#include <fstream>
using namespace std;

void Save_fout(ofstream &f_out, ifstream &f_in, int &predx,
               int &x, bool &endf, bool &ends);
void Merge(char *file1, char *file2, char *fileOut);

int _tmain(int argc, _TCHAR* argv[])
{
 char file1[]="c:\\tmp\\distr1.txt"; // Файл с данными
 char file2[]="c:\\tmp\\distr2.txt"; // Файл с данными
 char fileOut[]="c:\\tmp\\outtest-s1.txt"; // Результир. файл
 // Слияние:
 Merge(file1, file2, fileOut);
 return 0;
}

void Save_fout(ofstream &f_out, ifstream &f_in, int &predx,
               int &x, bool &endf, bool &ends)
 { // запись x в f_out и чтение нового x из  f_in:
       if(!f_in.eof())
       {
          f_out<<" "<<x; // записываем
          predx=x; // запоминаем предыдущий
          f_in>>x; // и читаем следующий
          ends=(predx > x);//=1, если серия на f_in закончилась
       }
      else
       {
          f_out<<" "<<x; // только записываем
          endf=1;   // все f_in элементы записаны
          ends=1; // серии f_in кончились
       }
 }

void Merge(char *file1, char *file2, char *fileOut)
```

```
{
      ofstream fout(fileOut);
      ifstream f1(file1);
      ifstream f2(file2);
      int pred1, a1; // элементы f1
      int pred2, a2; // элементы f2
      bool  endf1; /* Признак: endf1=1 - f1 прочитан,
                      все элементы записаны в fout */
      bool  endf2; // Аналогично для f2
      bool ends1; // признак конца серии (=1) на f1
      bool ends2; // аналогично для f2

     // читаем по элементу с каждой ленты:
      if(!f1.eof()) // f1 - не пустой
       {f1>>a1; endf1=0;}
      else endf1=1;
      if(!f2.eof()) // f2 - не пустой
       {f2>>a2; endf2=0;}
      else endf2=1;

      // Слияние:
      ends1=ends2=0; // начало серий
      while (!endf1 && !endf2) // пока оба файла не пусты
       {// 0
        // читаем серии из файлов:
        if(!ends1 && !ends2) // обе серии не пусты
         { // -  Что записывать: a1 или a2?
           if(a1<=a2) // a1 -> fout
             Save_fout(fout, f1, pred1, a1, endf1, ends1);
           else // a2 -> fout
             Save_fout(fout, f2, pred2, a2, endf2,ends2);
         } // if-
        else //- какая-то серия закончилась
         {
           if (ends1 && !ends2) /* кончилась серия на f1 и
                                   осталась серия на f2 */
             Save_fout(fout, f2, pred2, a2, endf2, ends2);
           else
           {
              if (ends2 && !ends1) /* кончилась серия на f2 и
                                      осталась серия на f1 */
               Save_fout(fout, f1, pred1, a1, endf1, ends1);
              else // обе серии закончились
                { // нужно брать новые серии
                     ends1=ends2=0
                }
            }
          } // else-
       } // while - 0

       // Теперь остатки одного из файлов:
      while(!endf1)
         Save_fout(fout, f1, pred1, a1, endf1, ends1);
```

```
      while(!endf2)
         Save_fout(fout, f2, pred2, a2, endf2, ends2);

       fout.close();
       f1.close();
       f2.close();
}
```

Реализация алгоритма сортировки слиянием

```
#include "stdafx.h"
#include "stdafx.h"
#include <stdio.h>
#include <iostream>
#include <fstream>
using namespace std;
void Distribute(char *fileA, char *file1, char *file2, int &s);
// распределение
void Save_fout(ofstream &f_out, ifstream &f_in, int &x, bool
&startf, bool &endf);
void Merge(char *file1, char *file2, char *fileOut); //  слияние
void MergeSort(char *fileA);  // сортировка
void show_file(char *fileA); // вывод содержимого файла на экран


int _tmain(int argc, _TCHAR* argv[])
{
     char fileA[]="c:\\tmp\\A.txt"; // Файл с данными

       MergeSort(fileA);

     return 0;
}

void MergeSort(char *fileA)
{
 char fileB[]="c:\\tmp\\fB.txt"; // Вспомогательный файл
 char fileC[]="c:\\tmp\\fC.txt"; // Вспомогательный файл
 int s; // кол-во переходов между файлами при распределении
 s=2;// начальное значения (произвольное, больше 1)

 while (s>=1) /* закончим, когда переходов не будет */
 {
  // распределение A по лентам (файлам) B и C:
  Distribute(fileA,fileB,fileC,s);// s - по ссылке
  // Слияние B и C  в A:
  if (s>=1) Merge(fileB,fileC,fileA);
 }
}//MergeSort()

void Distribute(char *fileA, char *file1, char *file2, int &s)
{
```

```
     // См. ранее

}

  void Save_fout(ofstream &f_out, ifstream &f_in, int &x, bool
&startf_out, bool &endf)
 {
       // См. ранее
 }

 void Merge(char *file1, char *file2, char *fileOut)
{
 // См. ранее
}
```

Сортировка файла распределением по сериям и слиянием файлов

```
// SortMergeFileEnd.cpp: Сортировка файла распределением по
сериям и // слиянием файлов
//

#include "stdafx.h"
#include "stdafx.h"
#include <stdio.h>
#include <iostream>
#include <fstream>
using namespace std;
void Distribute(char *fileA, char *file1, char *file2, int &s);
// распределение
void Save_fout(ofstream &f_out, ifstream &f_in, int &x, bool
&startf, bool &endf);
void Merge(char *file1, char *file2, char *fileOut); //  слияние
void MergeSort(char *fileA);  // сортировка
void show_file(char *fileA); // вывод содержимого файла на экран


int _tmain(int argc, _TCHAR* argv[])
{
     char fileA[]="c:\\tmp\\A.txt"; // Файл с данными
     //show_file(fileA);

      MergeSort(fileA);
     //cout<<"\n After Sorting \n";
     //show_file(fileA);

     return 0;
}

void MergeSort(char *fileA)
{
      char fileB[]="c:\\tmp\\fB.txt"; // Вспомогательный файл
     char fileC[]="c:\\tmp\\fC.txt"; // Вспомогательный файл
```

```
     int s; // кол-во переходов между файлами при распределении
     s=2;// начальное значения (произвольное, больше 1)

     while (s>=1) /* закончим, когда переходов не будет */
     {
       // распределение A по лентам (файлам) B и C:
       Distribute(fileA,fileB,fileC,s);// s - по ссылке
      // Слияние B и C  в A:
       if (s>=1) Merge(fileB,fileC,fileA);
     }

     }//MergeSort()

void Distribute(char *fileA, char *file1, char *file2, int &s)
{
     int  x, y;

     // распределение по лентам (файлам):
     ifstream fA(fileA);
    ofstream f1(file1);
     ofstream f2(file2);
     bool start2=1; // чтобы не записывать нач. пробелов в файле
f2

     s=1; // s-нечетное, пишем на ленту 1, а при четном - на
ленту 2
     fA>>x;
     f1<<x;
     while (!fA.eof())//пока не закончился файл  fA
        {
          fA>>y;
            if (y<x) s++; // конец серии-переходим на другую
ленту
         // запись на ленты (файлы):
            if (s%2 != 0) // при нечетном s запись на ленту 2
(f2)
                 if (start2) {f2<<y; start2=0;}
                 else f2<<" "<<y;
            else // при четном s запись на ленту 1 (f1)
                 f1<<" "<<y;
            x=y;
          }
          fA.close();
          f1.close();
          f2.close();
}

  void Save_fout(ofstream &f_out, ifstream &f_in, int &x, bool
&startf_out, bool &endf)
 {
       if(!f_in.eof())
                 {
```

```
                     if(startf_out) { f_out<<x; // без начального
пробела
                                      startf_out=0;
                     }
                     else f_out<<" "<<x; // записываем

                     f_in>>x; // и читаем
                 }
                 else
                 {
                       if(startf_out) { f_out<<x; // без
начального пробела
                                       startf_out=0;
                       }
                      else f_out<<" "<<x; // только записываем
                      endf=1;
                 }
 }

 void Merge(char *file1, char *file2, char *fileOut)
{
 ofstream fout(fileOut);
 ifstream f1(file1);
 ifstream f2(file2);
 int a1, a2;
 bool  endf1; // Признак:
        // endf1=1: f1 прочитан, все элементы записаны в fout
 bool  endf2; // Аналогично для f2
 bool startf_out=1; //
 if(!f1.eof()) // f1 - не пустой
  {f1>>a1; endf1=0;}
 else endf1=1;
 if(!f2.eof()) // f2 - не пустой
  {f2>>a2; endf2=0;}
 else endf2=1;
 // Слияние:
 while (!endf1 && !endf2) // пока оба файла не пусты
  {
   // Что записывать: a1 или a2?
   if(a1<=a2) // a1 -> fout
       Save_fout(fout, f1,a1,startf_out,endf1);
  else // a2 -> fout
       Save_fout(fout, f2,a2,startf_out,endf2);
  } // while
  // Теперь остатки одного из файлов:
  while(!endf1)
     Save_fout(fout, f1,a1,startf_out,endf1);
  while(!endf2)
     Save_fout(fout, f2,a2,startf_out,endf2);
 // закрываем файлы:
  fout.close();
  f1.close();
  f2.close();
```

```
}


 void show_file(char *fileA)
{
     int a;
     ifstream fA(fileA);
     cout<<"\n"<<fileA<<"\n";
     while (!fA.eof())
       {
             fA>>a;
             cout<<"\n"<<a;
         }
     fA.close();
}
```

Литература

1. Монган Дж., Гижере Э., Киндлер Н. *Работа мечты для программиста. Тестовые задачи и вопросы при собеседовании в ведущих IT-компаниях*. – СПб.: Питер, 2014. – 368 с.
2. Кнут Д. *Искусство программирования для ЭВМ. Том 3. Сортировка и поиск*. – М.: ООО «И.Д. Вильямс», 2014. – 824 с.
3. Липачёв Е.К. *Технология программирования. Базовые конструкции языка С/С++*. – Казань: Казанский университет, 2012. – 142 с. URL: http://repository.kpfu.ru/?p_id=47437.
4. Прата С. *Язык программирования С++. Лекции и упражнения*. – М.: ООО «И.Д. Вильямс», 2012. – 1248 с.
5. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. *Алгоритмы: построение и анализ*, 3-е изд. – М.: ООО «И.Д. Вильямс», 2013. – 1328 с.
6. Динман М.И. *С++. Освой на примерах*. – СПб.: БХВ-Петербург, 2006. – 384 с.
7. Хэзфилд Р., Кирби Л. И др. *Искусство программирования на С. Фундаментальные алгоритмы, структуры данных и примеры приложений. Энциклопедия программиста*. – К.: Издательство «ДиаСофт», 2001. – 736 с.
8. Фридман А., Кландер Л., Михаэлис М., Шильдт Х. *С/С++. Архив программ*. – М.: ЗАО «Издательство БИНОМ», 2001. – 640 с.
9. Weiss M.A. *Data Structures and Problem Solving Using C++*. – Pearson Education International Inc., 2014. – 635 p.
10. Бхаргава А. *Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих*. – СПб.: Питер, 2017. – 288 с.
11. Седжвик Р. *Фундаментальные алгоритмы на С++. Анализ/Структуры данных/Сортировка/Поиск*. – Киев: «ДиаСофт», 2001. – 688 с.