МЕТОДЫ СОРТИРОВОК

Сортировка – процесс упорядочения множества объектов по заданному признаку.

Обычно сортировку подразделяют на два класса: ***внутреннюю*** и ***внешнюю***.

При внутренней сортировке все элементы хранятся в оперативной памяти, таким образом, как правило, это сортировка массивов. При внешней сортировке — элементы хранятся на внешнем запоминающем устройстве, это сортировка содержимого файлов.

Одно из основных требований к методам сортировки — экономное использование памяти. Это означает, что переупорядочение нужно выполнять «на том же месте», то есть методы пересылки элементов из одного массива в другой не представляют интереса.

Существуют различные алгоритмы сортировки. Их можно разделить на два класса:

1) Простые и понятные, но неэффективные для больших массивов:
   - Метод «пузырька» (простых обменов)
   - Метод простого выбора
   - Метод вставок
2) Сложные, но эффективные:
   - Быстрая сортировка (сортировка Хоара)
   - Сортировка слияниями
   - Пирамидальная сортировка

Идея сортировок. Массив разбивают на две части: отсортированную и неотсортированную. Количество элементов в отсортированной части растет, а в неотсортированной уменьшается.

Сортировать массив можно:

1) ***по возрастанию*** – каждый следующий элемент больше предыдущего: $a[1]<a[2]<\ldots<a[n]$;

2) ***по неубыванию*** – каждый следующий элемент не меньше предыдущего: $a[1]\leq a[2]\leq\ldots\leq a[n]$;

3) ***по убыванию*** - каждый следующий элемент меньше предыдущего: $a[1]>a[2]>\ldots>a[n]$;

4) ***по невозрастанию*** – каждый следующий элемент не больше предыдущего: $a[1]\geq a[2]\geq\ldots\geq a[n]$.

МЕТОД «ПУЗЫРЬКА» («КАМНЯ»)

Метод простых обменов

Производится последовательное упорядочение смежных пар элементов массива по убыванию:
x[1] и *x*[2], *x*[2] и *x*[3], …*x*[*N*-1] и *x*[*N*].
Если в паре первый элемент меньше второго, то элементы меняются местами. Далее, таким же образом, обрабатываем следующую пару. В итоге, после *N*–1 сравнения минимальное значение переместится на место элемента *X*[*N*], т.е. "вверху" окажется самый "легкий" элемент – отсюда аналогия с пузырьком.

Следующий проход делается аналогичным образом до второго сверху элемента (*X*[*N*–1]), в результате второй по величине элемент поднимется на правильную позицию и т.д. Для сортировки всего массива нужно выполнить *N*–1 проход по массиву. При первом прохождении нужно сравнить *N*–1 пар элементов, при втором – *N*–2 пары, при *k*-м прохождении – *N*-*k* пар.

Если выполнять сортировку по возрастанию, то получим метод «камня», то есть самый тяжелый элемент опустится на дно массива.

Пример. Дан массив из 5 элементов: 5 8 1 6 7. Упорядочить его по возрастанию.

Первый проход по массиву

5	8	1	6	7

5	8	1	6	7

5	1	8	6	7

5	1	6	8	7

5	1	6	7	8

Второй проход по массиву:

5	1	6	7	8

1	5	6	7	8

1	5	6	7	8

1	5	6	7	8

Третий проход по массиву:

1	5	6	7	8

1	5	6	7	8

1	5	6	7	8

Четвертый проход по массиву:

1	5	6	7	8

1	5	6	7	8

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

void input(int *m, int &n)
{
cout<<"Введите количество элементов массива ";
cin>>n;
for (int i=0;i<n;i++)
{
    cout<<"a["<<i<<"]=";
    cin>>m[i];
}

}
void print (int *m, int n)
{
for (int i=0;i<n;i++)
    cout<<m[i]<<" ";
cout<<"\n";

}
```

```
void sort_bubble (int *m, int n)   // Сортировка пузырьком
{
int i, j, t;
for (i=0;i<n-1;i++)
    for(j=0;j<n-i-1;j++)
        if (m[j]>m[j+1])
        {
            t=m[j];
            m[j]=m[j+1];
            m[j+1]=t;
        }
}
void main()
{
const int nmax=20;
int n, a[nmax];
input(a,n);
cout<<"Исходный массив:\n";
print(a,n);
sort_bubble(a,n);
cout<<"Отсортированный массив:\n";
print(a,n);
}
```

Среднее число сравнений и обменов имеют квадратичный порядок роста: $O(n^2)$, отсюда можно заключить, что алгоритм пузырька очень медленен и малоэффективен.

МОДИФИЦИРОВАННЫЙ «ПУЗЫРЕК»

При сортировке методом простого обмена («пузырька») возможна ситуация, когда на каком-либо из проходов не произошло ни одного обмена. Это значит, что все пары расположены в правильном порядке, так что массив уже отсортирован. И продолжать процесс не имеет смысла (особенно, если массив был отсортирован с самого начала).

Можно улучшить эффективность этого метода, просматривая массив только до тех пор, пока существует обмен между элементами. Как только при очередном просмотре не происходит ни одного обмена между элементами, это означает, что массив отсортирован. При таком подходе может потребоваться один просмотр массива, если массив уже был отсортирован.

Поэтому лучше использовать «модифицированный пузырек».

```
void modif_bubble(int *m, int n)   // Модифицированный пузырек
{
  int i=n;              // Длина неотсортированной части массива
  int f,t;
  do {
    f=0;                //Предположим, что массив является
отсортированным
    for (int k=0;k<i-1;k++)
      if (m[k]>m[k+1])
{
        t=m[k]; m[k]=m[k+1]; m[k+1]=t;          // Обмен
        f=1;                          // Массив был неотсортированным
}
    i--;
  } while (f && i>1);
}
```

ШЕЙКЕР-СОРТИРОВКА

Пример. *Упорядочить массив из 6 элементов: 5, 7, 9, 10, 12, 3 по неубыванию методом «модифицированного пузырька».*

Использование «модифицированного пузырька» отсортирует заданный массив за 5 просмотров (*n*-1). Устранить такое «неравноправие» можно путем смены направлений просмотров, т.е первоначально в направлении слева направо получаем 5, 7, 9, 10, 3, 12, а затем справа налево результат – 3, 5, 7, 9, 10, 12.

Этот алгоритм уменьшает количество перемещений, действуя следующим образом: за один проход из всех элементов выбирается минимальный и максимальный, минимальный элемент помещается в начало массива, а максимальный, соответственно, в конец. Далее алгоритм выполняется для остальных данных.

```
void sort_sheiker (int *m, int n)  // Шейкер-сортировка
{
  int left=1;              // левая граница
  int right=n-1;           // правая граница
  int j,t,k=n-1;
  do     // Обратный проход "Пузырька" от правой границы до левой
  {
    for (j=right;j>=left; j--)
      if (m[j-1]>m[j])
  {
        t=m[j-1]; m[j-1]=m[j]; m[j]=t;
        k=j;// фиксирование индекса последнего обмена
   }
    left=k+1;              // Левая граница
    //Прямой проход "Пузырька" от левой границы до правой
    for (j=left; j<=right; j++)
      if (m[j-1]>m[j])
  {
        t=m[j-1]; m[j-1]=m[j]; m[j]=t;
        k=j;               // фиксирование индекса последнего обмена
  }
    right=k-1;             // правая граница
   } while (left<=right);// До тех пор, пока левая граница не станет
больше  правой границы
}
```

СОРТИРОВКА ВСТАВКАМИ

Метод прямого включения

Начиная со 2-го элемента, каждый текущий элемент с номером *i* запоминается в промежуточной переменной.

Затем просматриваются предыдущие *i*–1 элемент и ищется место вставки i-го элемента таким образом, чтобы не нарушить упорядоченности, при этом все элементы, превышающие *i*-й, сдвигаются вправо.

На освободившееся место вставляется *i*-й элемент.

i=2	5	4	3	1
i=3	4	5	3	1
i=4	3	4	5	1
ИТОГ	1	3	4	5

```
    void sort_insert (int *m, int n)
    {
    int j,r;
    for (int i=1;i<n;i++)
    {
        r=m[i]; // Запоминаем текущий элемент в промежуточной
переменной
        j=i-1;
        while (j>=0 && m[j]>r)    // Ищем новое место вставки,
            { m[j+1]=m[j]; j--;}   // сдвигая на 1 элемент
вправо
        m[j+1]=r;  // На освободившееся место вставляется
элемент
    }
    }
```

СОРТИРОВКА МЕТОДОМ ПРОСТОГО ВЫБОРА

Обычно применяется для массивов, не содержащих повторяющихся элементов.

Алгоритм:

1) выбрать максимальный элемент массива;

2) поменять его местами с последним элементом (после этого самый большой элемент будет стоять на своём месте);

3) повторить пункты 1-2 с оставшимися *n*–1 элементами.

Пример. *Дан массив из пяти элементов: 5, 2, 7, 4, 6. Отсортировать по возрастанию методом простого выбора.*

	Элементы массива					Примечание
	5	2	7	4	6	max=7
Шаг 1	5	2	6	4	7	7↔6; max=6
Шаг 2	5	2	4	6	7	4↔6; max=5
Шаг 3	4	2	5	6	7	4↔5; max=4
Шаг 4	2	4	5	6	7	4↔2

```
    void sort_vybor (int *a, int n) // Сортировка простым
выбором
    {
    int k,m;
    for (int i=n-1;i>0;i--)
    {
        k=i;                        // Запоминаем номер и
        m=a[i];                     // значение текущего элемента
        for (int j=0;j<i;j++) // Поиск максимального элемента
            if (a[j]>m) {k=j; m=a[k];}
           if (k!=i)
       {                          // Меняем местами с последним
             a[k]=a[i];
             a[i]=m;
       }
    }
    }
```

МЕТОД БИНАРНОГО ПОИСКА

```
const int NMAX=100;
int a[NMAX];
int binSearch(int data[], int rzm_data, int search_key)
{
int L,H,M;
L=0;
H=rzm_data-1;
while (L<H)
{
    M=(L+H)/2;
    if (data[M]<search_key)
          L=M+1;
    else H=M;
}
if (data[L]==search_key) return L; else return -1;
}
```

СОРТИРОВКА БИНАРНЫМИ ВСТАВКАМИ

Алгоритм сортировки простыми включениями можно легко улучшить, пользуясь тем, что готовая последовательность *a[1]*,..., *a[i-1]*, в которую нужно включить новый элемент, уже упорядочена. Поэтому место включения можно найти значительно быстрее, применив бинарный поиск, который исследует средний элемент готовой последовательности и продолжает деление пополам, пока не будет найдено место включения. Модифицированный алгоритм сортировки называется ***сортировкой бинарными вставками (включениями)***.

```
void sort_bin_insert (int *a, int n) // Сортировка бинарными вставками
{ int x,left,right,sred;
  for (int i=1; i<n; i++)
  {
    if (a[i-1]>a[i])
{
      x=a[i];         // x – включаемый элемент
      left=0;         // левая граница отсортированной части массива
      right=i-1; // правая граница отсортированной части массива
    do {
          sred = (left+right)/2; // sred – новая "середина"
последовательности
          if (a[sred]<x ) left= sred+1;
           else right=sred-1;
    } while (left<=right); // поиск ведется до тех пор, пока левая
граница не окажется правее правой границы
      for (int j=i-1; j>=left;j--) a[j+1]= a[j];
      a[left]= x;
}
  }
}
```

СОРТИРОВКА ПОДСЧЕТОМ

Сортировка подсчётом («черпачная») — алгоритм сортировки, в котором используется диапазон чисел сортируемого массива для подсчёта совпадающих элементов.

Применение сортировки подсчётом возможно и целесообразно лишь тогда, когда сортируемые числа имеют диапазон возможных целых положительных значений, который достаточно мал по сравнению с сортируемым множеством, например, миллион натуральных чисел от 0 до 1000.

Существует несколько вариантов алгоритма. Рассмотрим простейший из них. Пусть есть массив A из N элементов, включающий числа в диапазоне от 0 до k-1 (то есть всего максимум k различных значений)

1) Создать вспомогательный массив C[0..k - 1], состоящий из нулей,
2) Последовательно прочитать элементы входного массива A и для каждого A[i] увеличить C[A[i]] на единицу.
3) Пройти по массиву C и для каждого j от 0 до k-1 в массив A последовательно записать число j C[j] раз.

Пример

Пусть дан массив A=(2 5 6 9 4 1 8 2 9 8 4 1 4 1) из 14 элементов.

Диапазон чисел – от 0 до 9 (для C++ нижняя граница должна быть 0, верхняя – максимальный элемент массива), то есть максимальное количество возможных значений – 10.

1. Создаем массив C=(0,0,0,0,0,0,0,0,0,0)

2. Проходим по массиву A и заполняем массив C: C=(0,3,2,0,3,1,1,0,2,2)

3. Читаем массив C и в массив A записываем указанное в C количество элементов, значение которых совпадает с индексом элемента, то есть 0 элементов со значением 0, 3 элемента со значением 1, 2 элемента со значением 2 и т.д.

Получаем:
A=(1,1,1,2,2,4,4,4,5,6,8,8,9,9)

```
void sort_count(int *A,int n, int k)
{
int *C = new int[k];
for (int i = 0; i < k; i++)
   C[i] = 0;
for (int i = 0; i < n; i++)
    C[A[i]] = C[A[i]] + 1;
int b = 0;
for (int j = 0; j < k; j++) //проходим по массиву C
     for (int i = 0; i < C[j]; i++) //выводим
совпадающие //значения, количество которых равно C[j]
          {
            A[b] = j;
            b = b + 1;
          }
delete [] C;
}
```

Что делать, если диапазон значений (min и max) заранее не известен? Что делать, если минимальное значение больше нуля или в сортируемых данных присутствуют отрицательные числа?

Первый вопрос можно решить линейным поиском min и max, что не повлияет на асимптотику алгоритма.

Второй вопрос. Если min больше нуля, то следует при работе с массивом C из A[i] вычитать min, а при обратной записи прибавлять. При наличии отрицательных чисел нужно при работе с массивом C к A[i] прибавлять min, а при обратной записи вычитать.

ЦИФРОВАЯ СОРТИРОВКА

Алгоритм цифровой сортировки существенно отличается от описанных ранее.
Во-первых, он совсем не использует сравнений сортируемых элементов.
Во-вторых, сортируемые значения необходимо разделить на части. Например, слово можно разделить по буквам, число - по цифрам.

До сортировки необходимо знать два параметра: k и m, где
k - количество разрядов в самом длинном ключе
m - разрядность данных: количество возможных значений разряда ключа

При сортировке русских слов m = 33, так как буква может принимать не более 33 значений.
Если в самом длинном слове 10 букв, k = 10.
Аналогично, для десятичных целых чисел m=10, если в качестве разряда брать цифру. Если самое длинное число содержит 4 разряда, то k=4.
При цифровой сортировке используются вспомогательные массивы, количество которых равно m. Поразрядная сортировка состоит из двух процессов, называемых распределение и сборка, выполняемых для j=1,2,...,k.
Фаза распределения разносит элементы исходного массива по вспомогательным массивам в зависимости от значения разряда.
Фаза сборки состоит в объединении массивов в один массив.

Рассмотрим пример работы алгоритма на входном массиве:

0	8	12	56	7	26	44	97	2	37	4	3	3	45	10

Максимальное число содержит две цифры, значит, разрядность данных k=2.
Числа даны в десятичной системе счисления, значит m=10

Первый проход (j=1)
Распределение по первой справа цифре:

Массив	элементы		
L0	0	10	
L1			
L2	12	2	
L3	3	3	
L4	44	4	
L5	45		
L6	56	26	
L7	7	97	37
L8	8		
L9			

Сборка: соединяем массивы один за другим

0	10	12	2	3	3	44	4	45	56	26	7	97	37	8

Второй проход (j=2)

Распределение по второй справа цифре:

Массив	элементы						
L0	0	2	3	3	4	7	8
L1	10	12					
L2	26						
L3	37						
L4	44	45					
L5	56						
L6							
L7							
L8							
L9	97						

Сборка: соединяем массивы один за другим

0	2	3	3	4	7	8	10	12	26	37	44	45	56	97

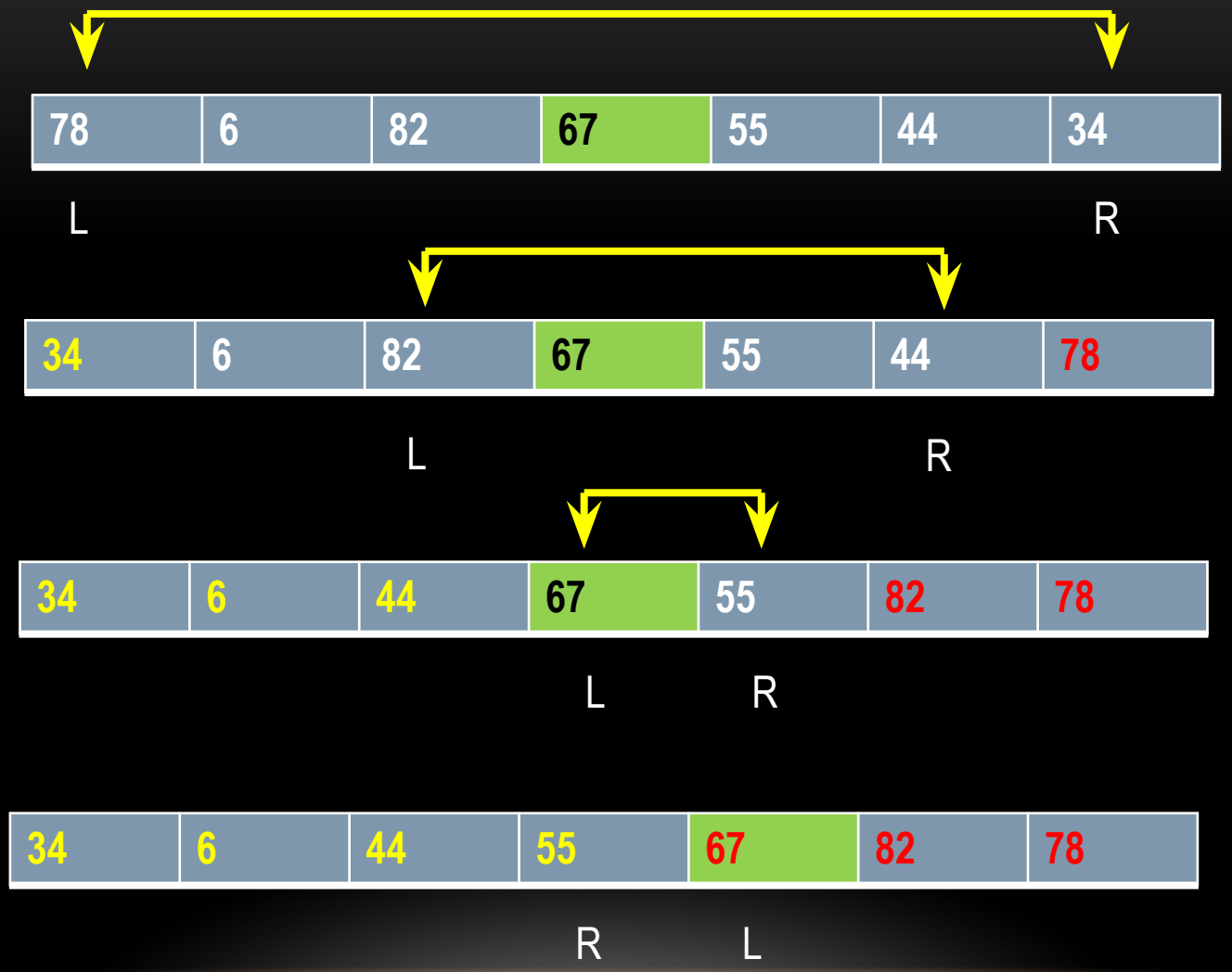
БЫСТРАЯ СОРТИРОВКА

Сортировка Хоара

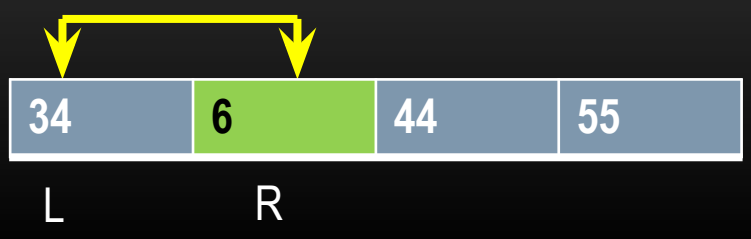
Рассмотрим элемент, находящийся посередине массива *M* (назовем его *X*). Затем начнем осуществлять два встречных просмотра массива: двигаемся слева направо, пока не найдем элемент $M[i]>X$ и справа налево, пока не найдем элемент $M[j]<X$. Когда указанные элементы будут найдены, поменяем их местами и продолжим процесс "встречных просмотров с обменами", пока два идущих навстречу друг другу просмотра не встретятся.

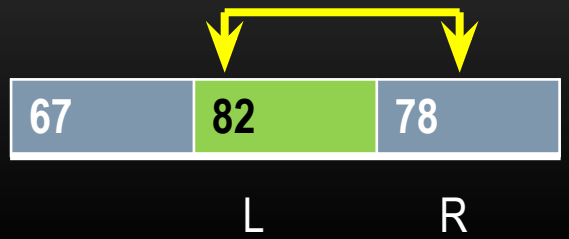
В результате массив разделится на две части: левую, состоящую из элементов меньших или равных *X*, и правую, в которую попадут элементы, большие или равные *X*. После такого разделения массива *M*, чтобы завершить его сортировку, достаточно осуществить то же самое с обеими полученными частями, затем с частями этих частей и т.д., пока каждая часть не будет содержать ровно один элемент.

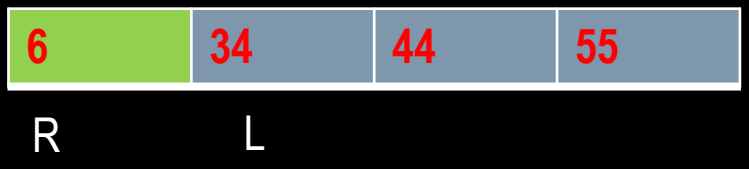
Шаг 1

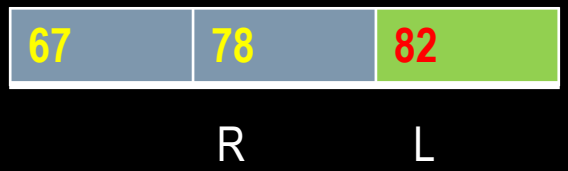

78 6 82 67 55 44 34
L
R
34 6 82 67 55 44 78
L
R
34 6 44 67 55 82 78
L
R
34 6 44 55 67 82 78
R
L

Шаг 2


34
6
44
55
L
R

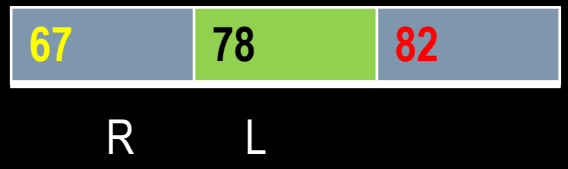

67
82
78
L
R


6
34
44
55
R
L


67
78
82
R
L

Шаг 3


6
34
44
55
R
L


67
78
82
R
L

. . .

```
void Qsort(int a[],int L,int R)
{
  int i=L,j=R,w,x;
  x=a[(L+R)/2];
  do
   {
    while (a[i]<x) i++;
 while (a[j]>x) j--;
 if(i<=j)
       {w=a[i]; a[i]=a[j]; a[j]=w;
   i++; j--;}
   } while (i<j);
  if (L<j) Qsort(a,L,j);
  if (i<R) Qsort(a,i,R);
}
```

Основная программа должна содержать вызов процедуры в виде: **Qsort(a,0,N-1).**
Неэффективна для массивов небольшого размера.

СОРТИРОВКА СЛИЯНИЯМИ

Метод слияний – один из первых в теории алгоритмов сортировки. Он предложен Дж. фон Нейманом в 1945 году.

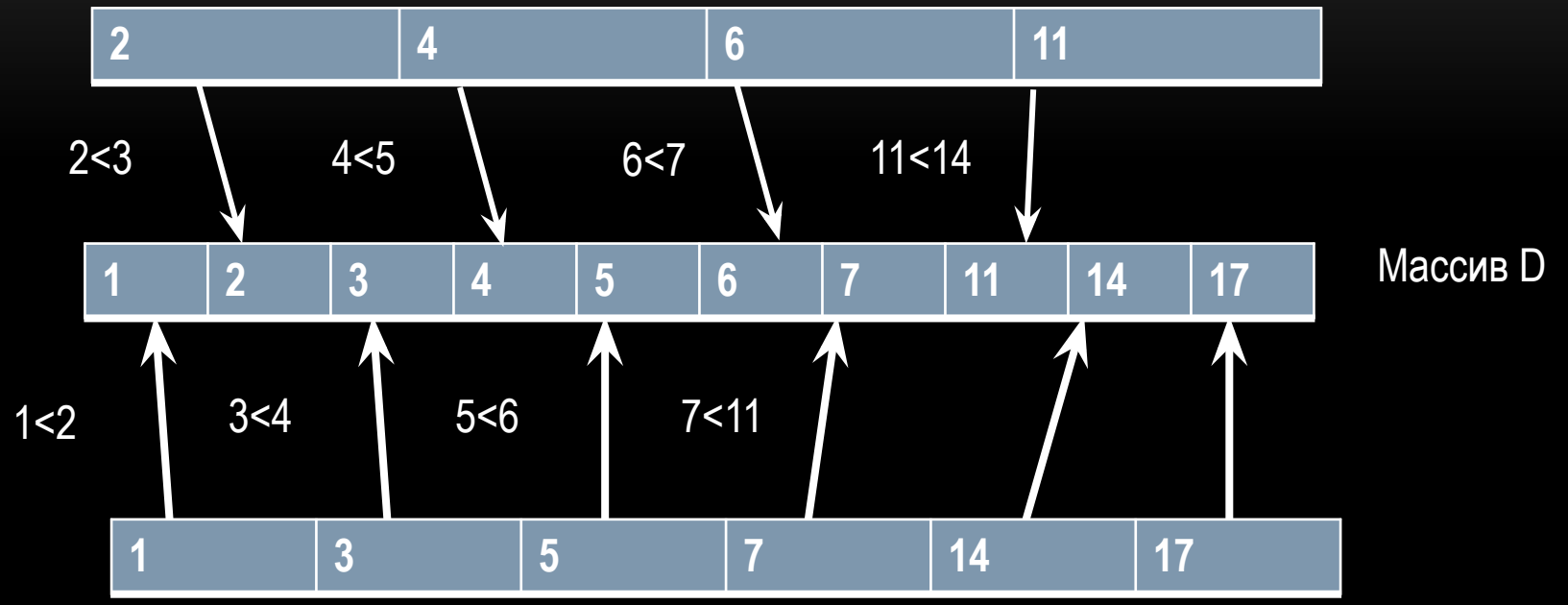
Рассмотрим следующую задачу. Объединить («слить») упорядоченные фрагменты массива A $A[k],...A[m]$ и $A[m+1],...,$ $A[q]$ в один $A[k],...,$ $A[q]$, естественно, тоже упорядоченный ($k \le m \le q$).

Основная идея решения состоит в сравнении очередных элементов каждого фрагмента, выяснении, какой из элементов меньше, переносе его во вспомогательный массив D (для простоты) и продвижении по тому фрагменту массива, из которого взят элемент. При этом следует не забыть записать в D оставшуюся часть того фрагмента, который не успел себя «исчерпать».

Рассмотрим суть метода слияния на примере.

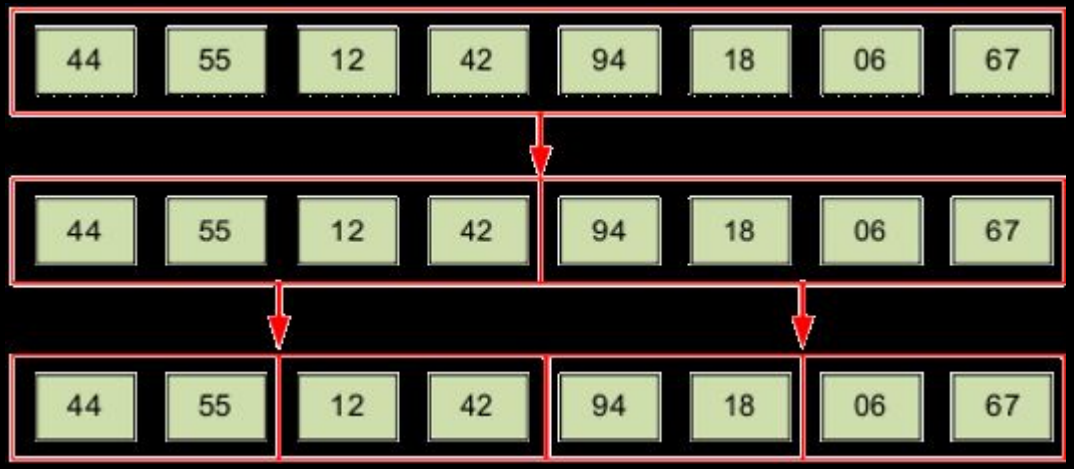
Первый фрагмент: 2 4 6 11

Второй фрагмент: 1 3 5 7 14 17

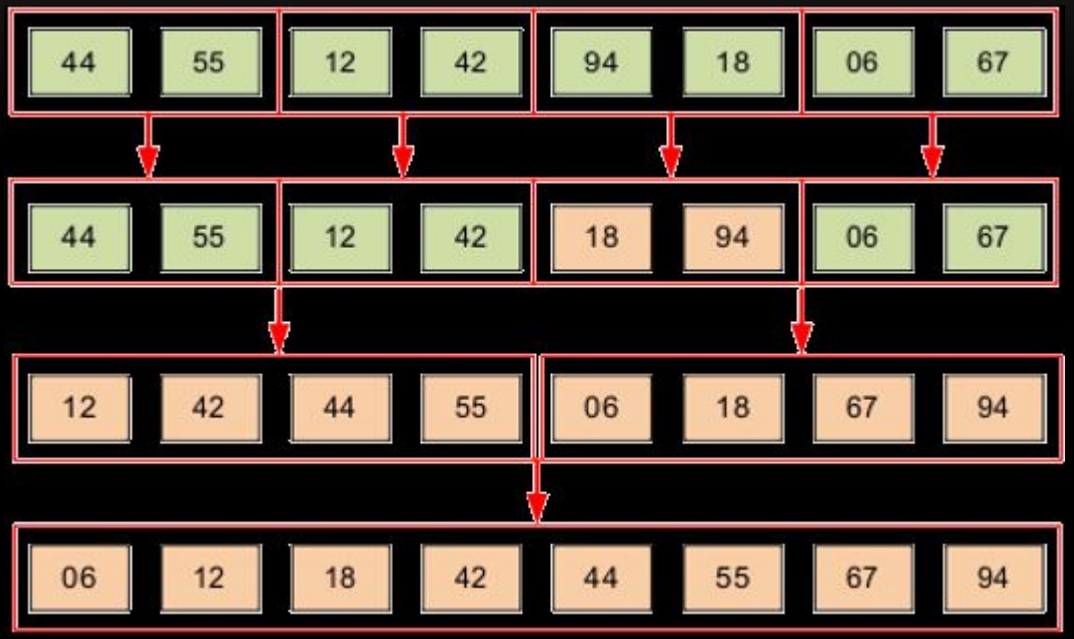


Исходная последовательность рекурсивно разбивается на половины, пока не получим подпоследовательности по 1 элементу. Из полученных подпоследовательностей формируем упорядоченные пары методом слияния, затем - упорядоченные четверки и т. д.

Разбиваем последовательность на 2 половины (рекурсивно, пока не получим пары).



Каждую подпоследовательность упорядочиваем методом слияния и получаем готовую последовательность



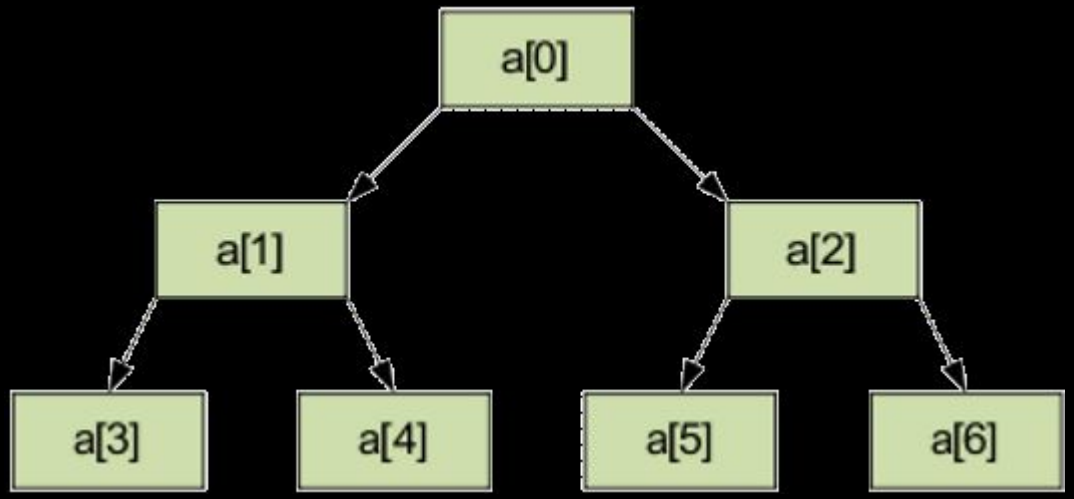
```
void mergeSort(int *a, int l, int r)
{
if (l == r) return; // границы сомкнулись
int mid = (l + r) / 2; // определяем середину последовательности
   // и рекурсивно вызываем функцию сортировки для каждой половины
mergeSort(a, l, mid);
mergeSort(a, mid + 1, r);
int i = l;  // начало первого пути
int j = mid + 1; // начало второго пути
int tmp[20]; // дополнительный массив
for (int step = 0; step < r - l + 1; step++)
// для всех элементов дополнительного массива
{
// записываем в формируемую последовательность меньший из элементов двух
путей
// или остаток первого пути если j > r
      if ((j > r) || ((i <= mid) && (a[i] < a[j])))
      {
            tmp[step] = a[i];
            i++;
      }
      else
      {
            tmp[step] = a[j];
            j++;
      }
}
// переписываем сформированную последовательность в исходный массив
for (int step = 0; step < r - l + 1; step++)
      a[l + step] = tmp[step];
}
```

Первый вызов процедуры – Sort_Sl(a,0,n-1);, где *a* – исходный массив из *N* элементов. Данный алгоритм работает быстрее чем, например, пузырьковая, а недостаток метода – в требовании довольно большого объема дополнительной памяти.

ПИРАМИДАЛЬНАЯ СОРТИРОВКА

Метод предложен Дж. Уильямсом и Р. У. Флойдом в 1964 году.

Элементы массива *A* образуют ***пирамиду***, если для всех значений *i* выполняются условия: *A*[i]<*A*[2*i+1] и *A*[i]<*A*[2*i+2] для всякого *i* от 0 до *N*/2-1, где *N* – размерность массива *A*. Первый элемент пирамиды является наименьшим



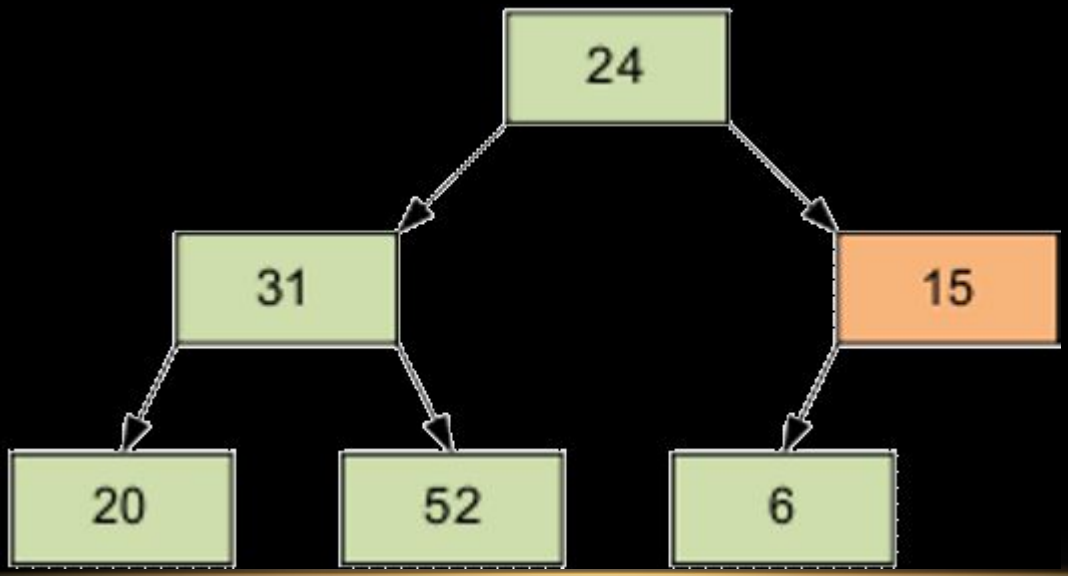
Общая идея пирамидальной сортировки заключается в том, что сначала строится пирамида из элементов исходного массива, а затем осуществляется сортировка элементов.

1 этап Построение пирамиды. Определяем правую часть дерева, начиная с **n/2-1** (нижний уровень дерева). Берем элемент левее этой части массива и просеиваем его сквозь пирамиду по пути, где находятся меньшие его элементы, которые одновременно поднимаются вверх; из двух возможных путей выбираете путь через меньший элемент.

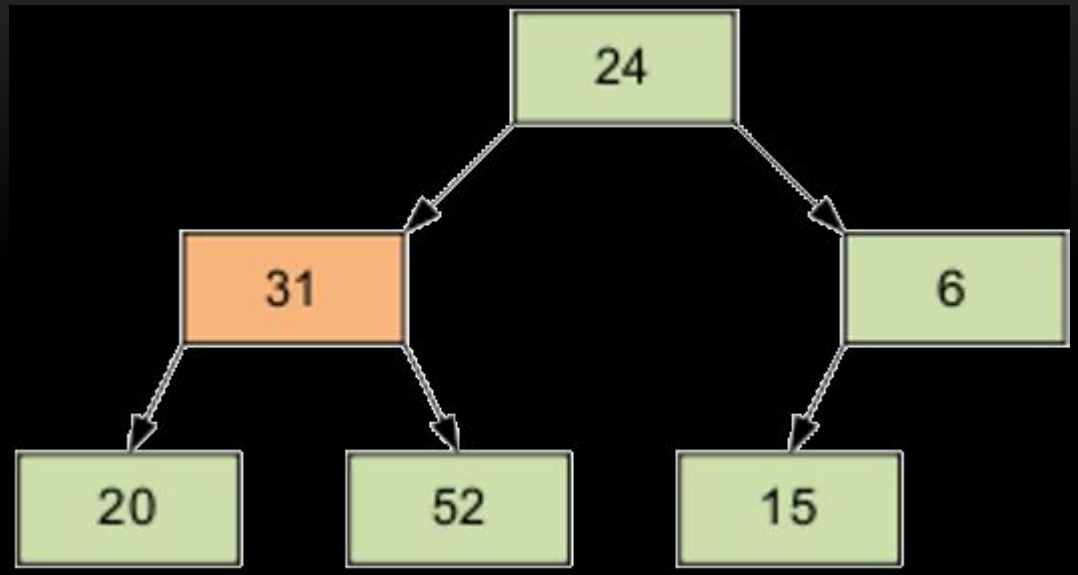
Например, дан массив для сортировки

24, 31, 15, 20, 52, 6

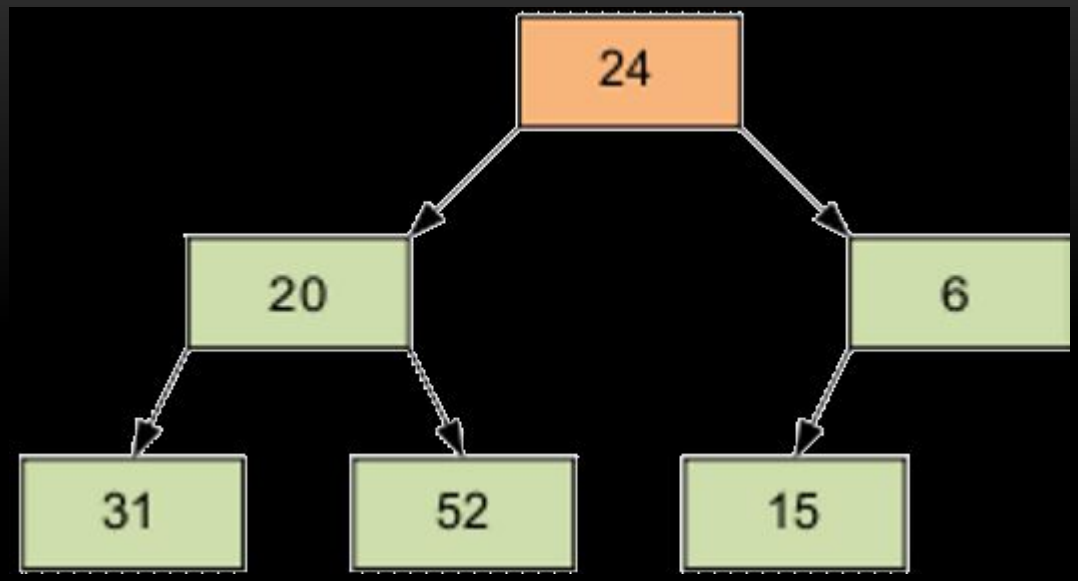
Расположим элементы в виде исходной пирамиды; номер элемента правой части (6/2-1)=2 - это элемент 15. Сравниваем с дочерними элементами и поднимаем наименьшие вверх.



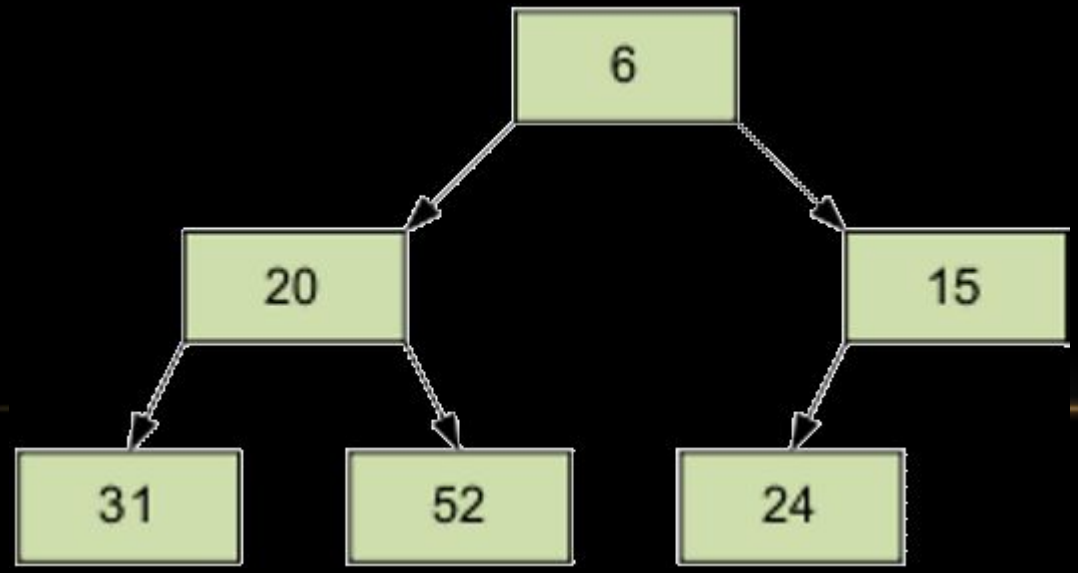
Результат просеивания элемента 15 через пирамиду. Следующие просеиваемый элемент – 31.



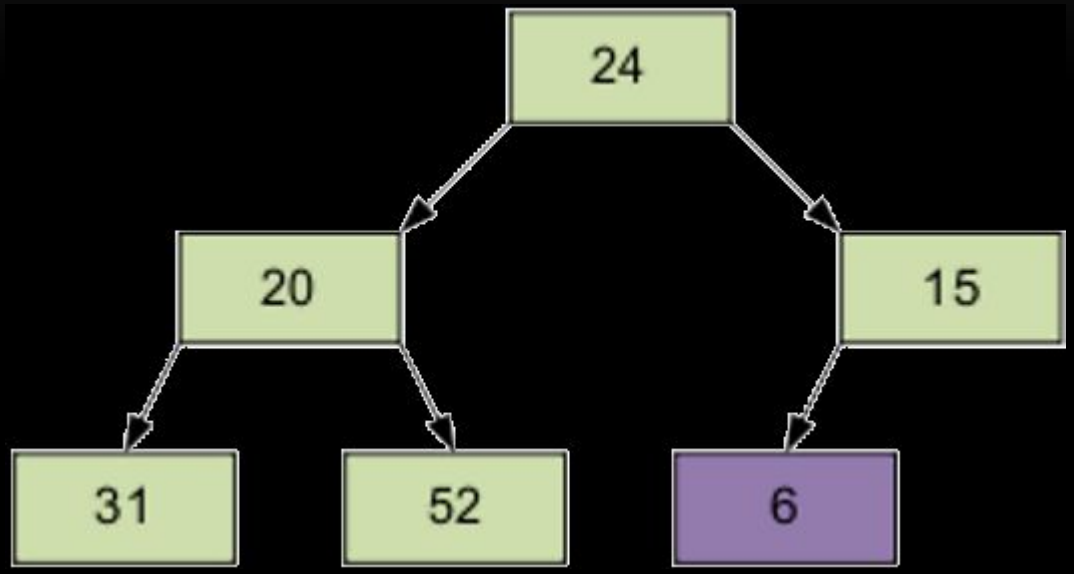
Далее просеиваем с номером 0 - 24.



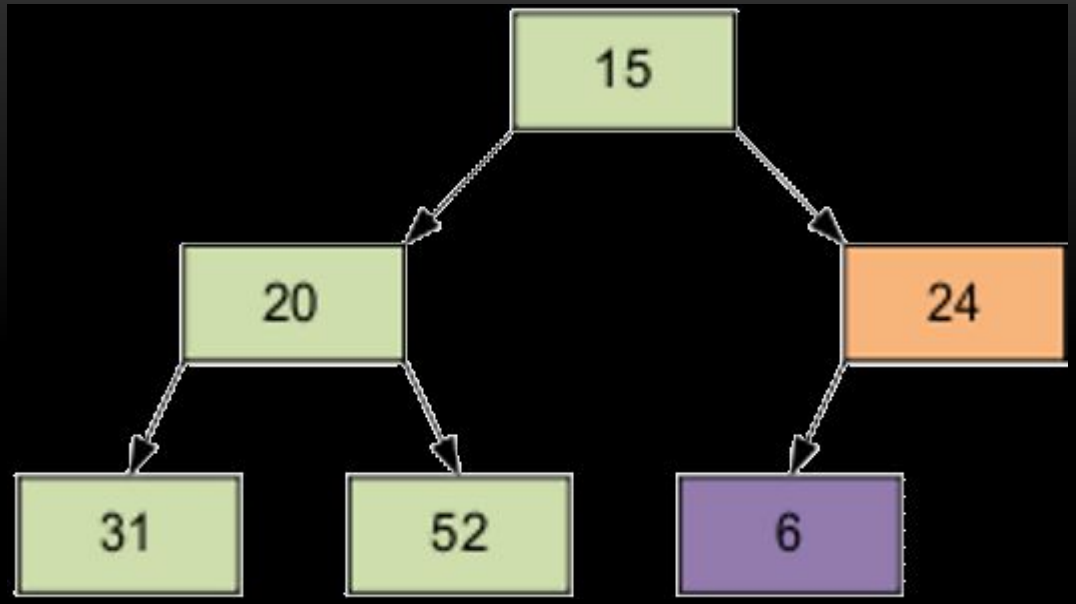
Результат: наименьший элемент находится на вершине пирамиды

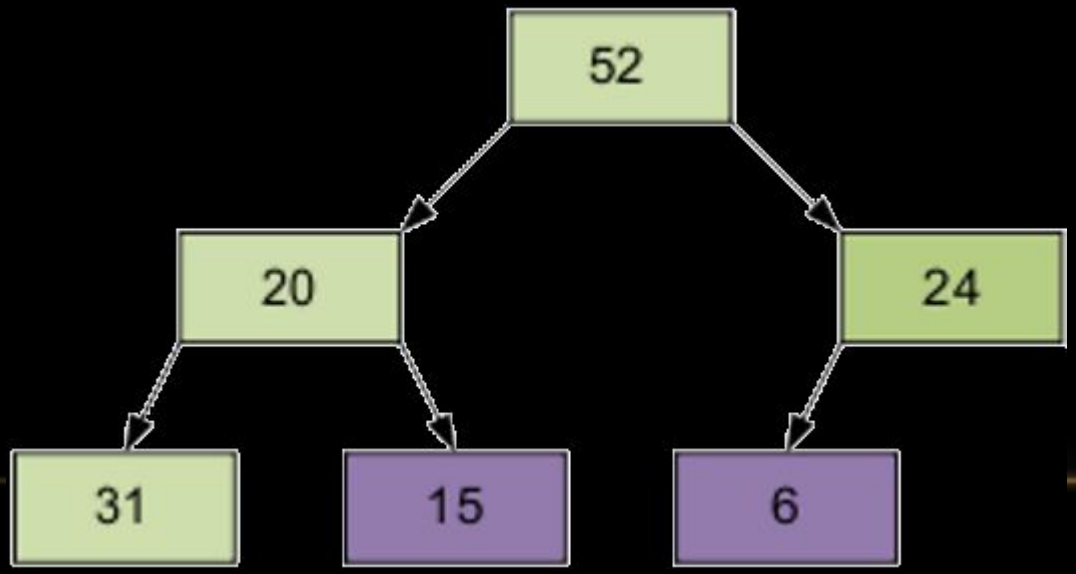


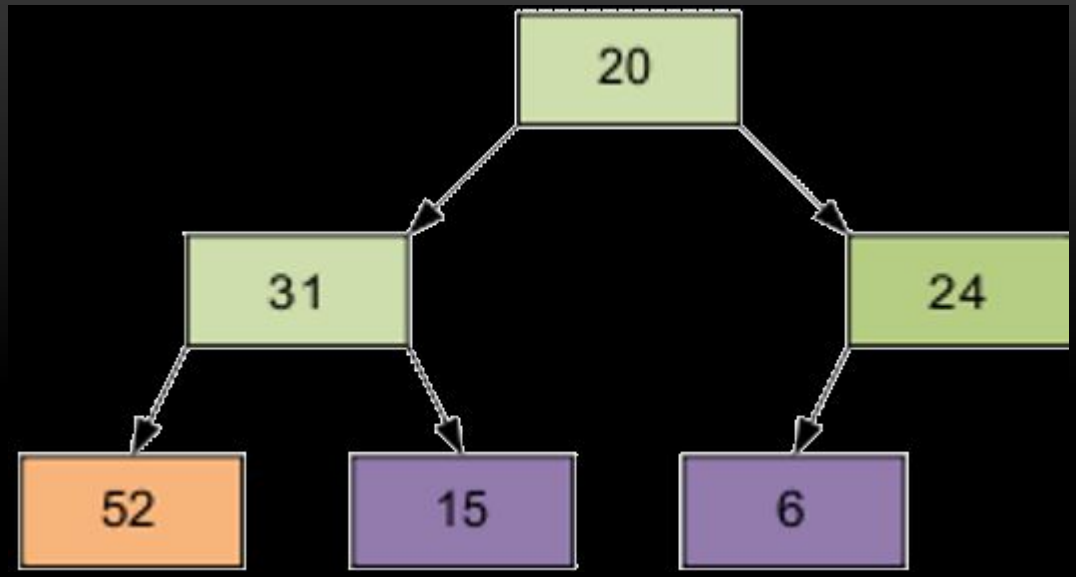
2 этап Сортировка на построенной пирамиде. Берем последний элемент массива в качестве текущего. Меняем верхний (наименьший) элемент массива и текущий местами.

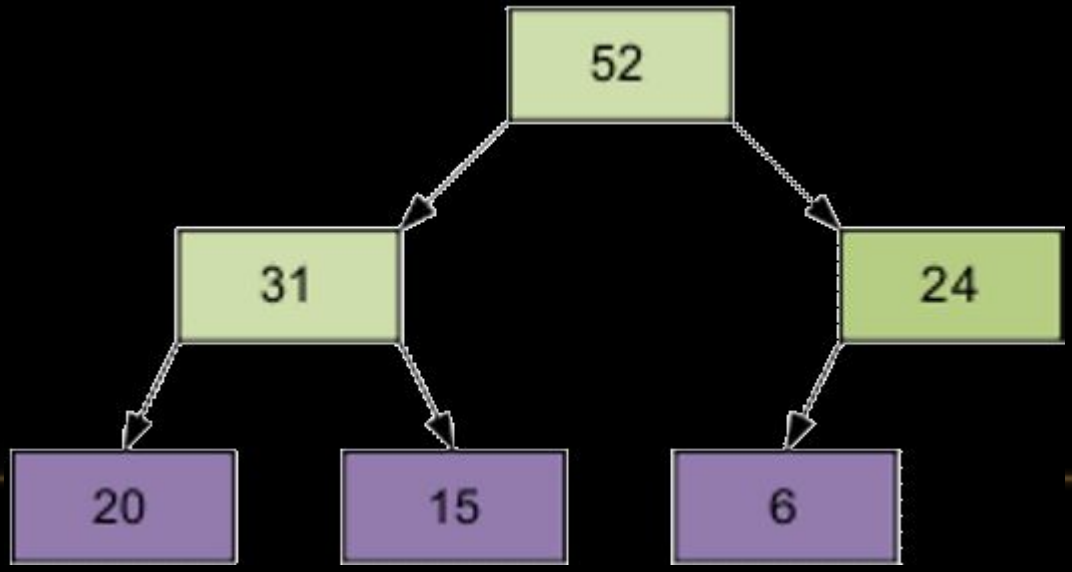


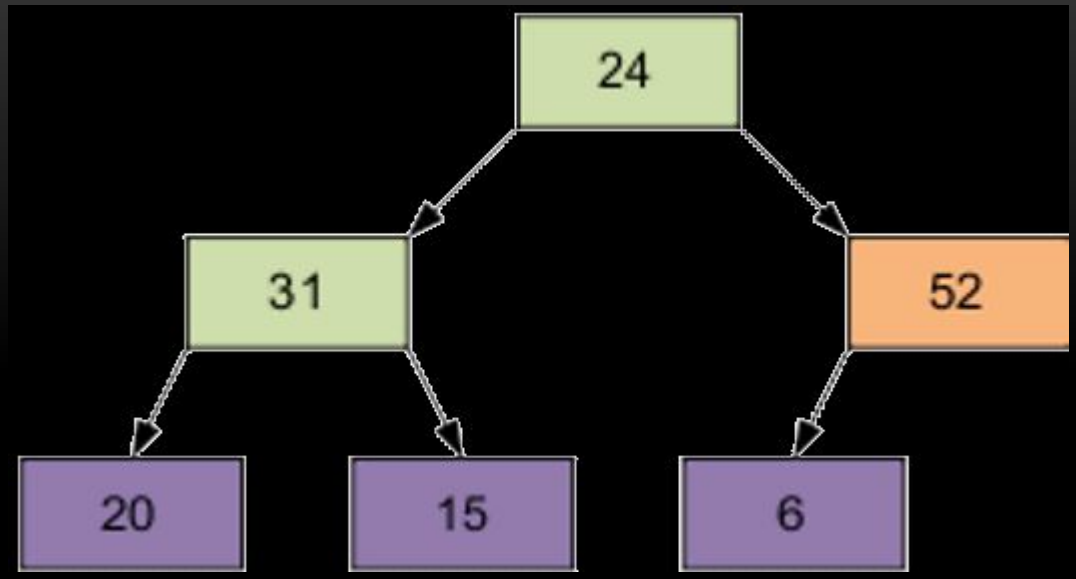
Повторяем операцию просеивания без учета последнего элемента (для N-1 элемента).

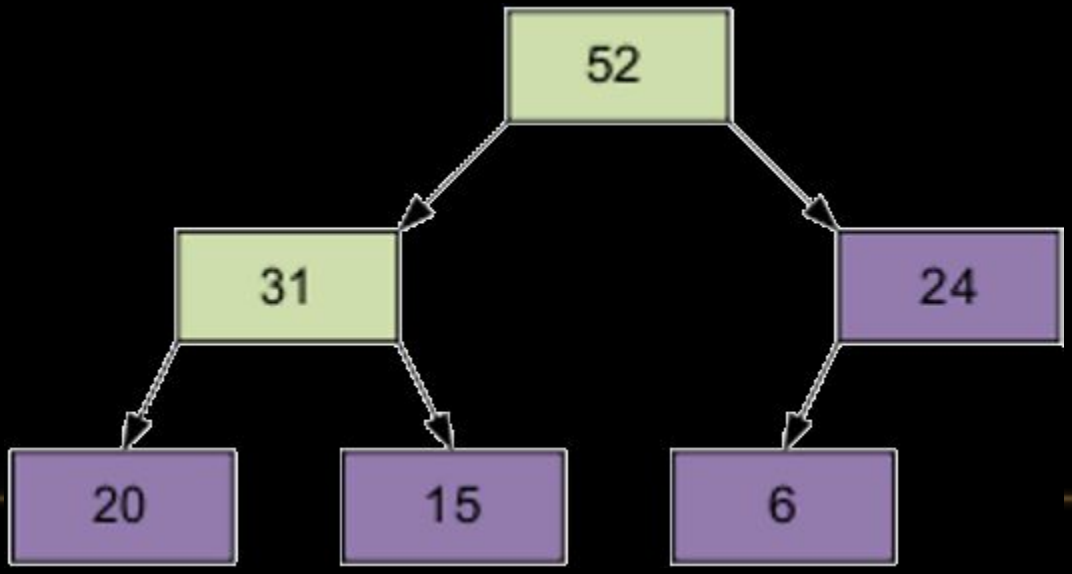


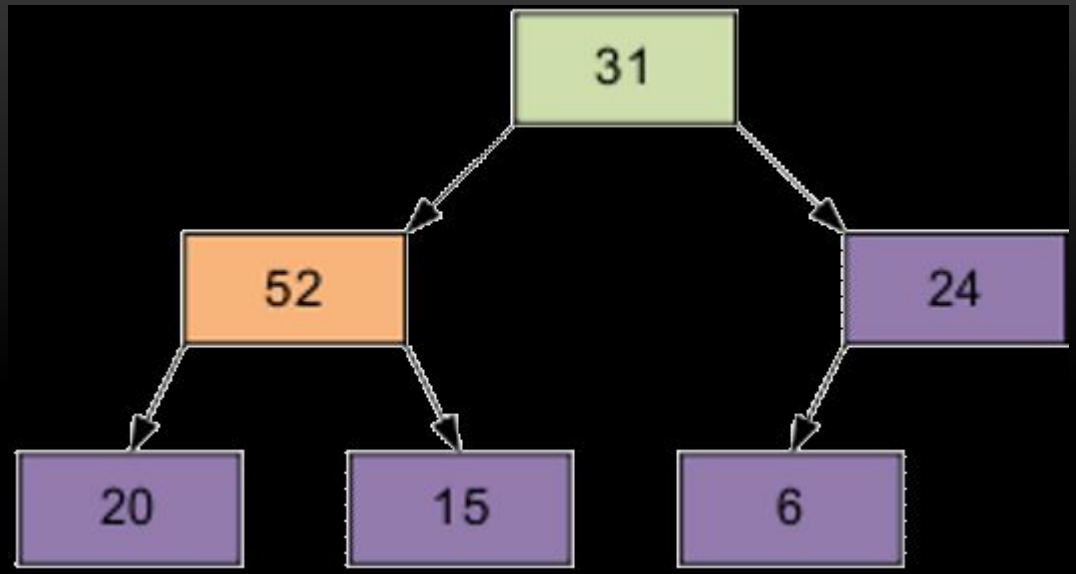


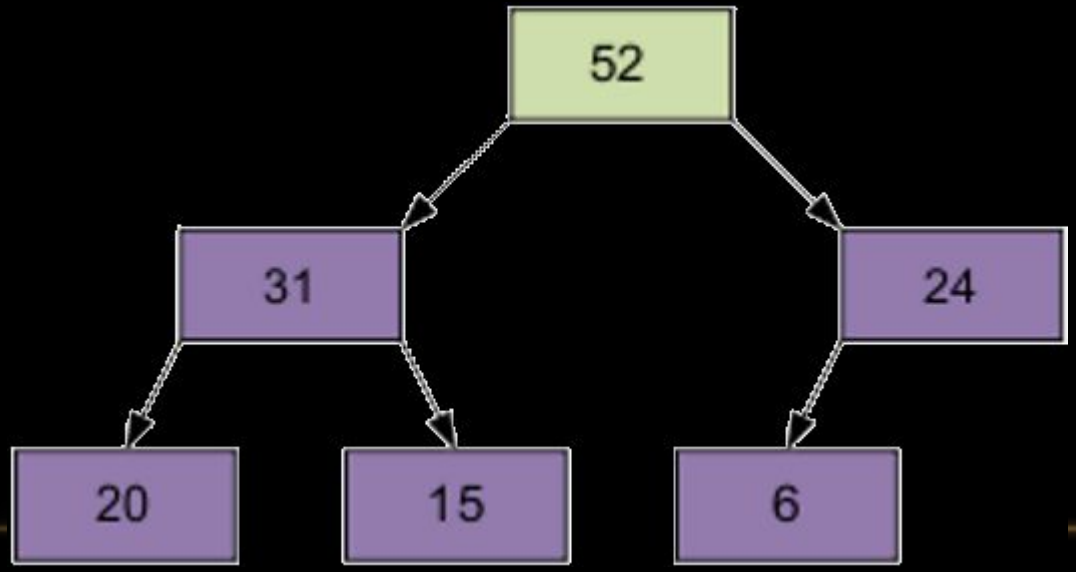

20
31
24
52
15
6

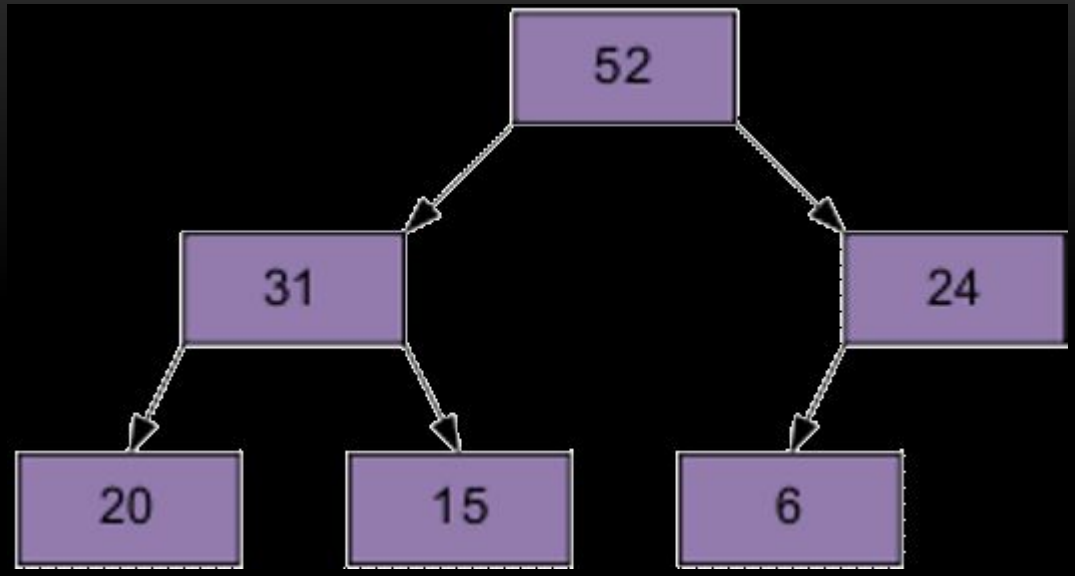

52
31
24
20
15
6


24
31
52
20
15
6


52
31
24
20
15
6


31
52
24
20
15
6


52
31
24
20
15
6



Несмотря на некоторую внешнюю сложность, пирамидальная сортировка является одной из самых эффективных. Алгоритм сортировки эффективен для больших n. В худшем случае требуется $n \cdot \log_2(n)$ шагов, сдвигающих элементы. Среднее число перемещений примерно равно $(n/2) \cdot \log_2(n)$.

```
void heapSort(int a[], int size)
{
    int i, temp;
    // строим пирамиду
    for (i = size / 2 - 1; i >= 0; i--)
           downHeap(a, i, size - 1);
    // теперь a[0]...a[size-1] пирамида
    for (i = size - 1; i > 0; i--) {
    // меняем первый с последним
        temp = a[i]; a[i] = a[0]; a[0] = temp;

    // восстанавливаем пирамидальность a[0]...a[i-1]
        downHeap(a, 0, i - 1);
    }
}
```

```
void downHeap(int a[], int k, int n)
{
    //  процедура просеивания следующего элемента
    //  До процедуры: a[k+1]...a[n]  - пирамида
    //  После:  a[k]...a[n]  - пирамида
    int new_elem, child;
    new_elem = a[k];
    while (k <= n / 2)
              {
               // пока у a[k] есть дети
        child = 2 * k;
        //  выбираем большего сына
        if (child < n && a[child] < a[child + 1])
            child++;
        if (new_elem >= a[child]) break;
        a[k] = a[child];     // переносим сына наверх
        k = child;
    }
    a[k] = new_elem;
}
```