БЫСТРЫЕ СОРТИРОВКИ

- Быстрая обменная сортировка (сортировка Хоара).
- Быстрая сортировка вставкой (сортировка Шелла).
- Быстрые сортировки выбором.
- Сравнительный анализ методов сортировки.

4.10. Быстрая обменная сортировка (сортировка Хоара)

- Считается самой эффективной из известных внутренних сортировок.
- Получила название – **Quicksort**

Быстрая обменная сортировка (сортировка Хоара)

Идея алгоритма:

- Из исходного массива ***выбирается некоторый элемент***, который принимается в качестве ***разделителя*** или опорного элемента.
- Все ключи, **меньшие разделителя**, располагаются **до него**, а все **большие** – **после него**.
- Перестановка элементов выполняется путём обмена местами ключей, которые необходимо переместить в другую часть массива.
- При этом обмениваются ключи, расположенные на большом расстоянии друг от друга и этим достигается наивысший эффект упорядочивания.

В примере реализации алгоритма создаётся процедура

`Hoor(left, right:integer)` - *в Pascal*

или функция

`void hoor(int left,right)` – *в C++*

Алгоритм является рекурсивным.

Обозначения переменных:

`right` – правая граница рассматриваемой части массива **A**;

`left` – левая граница рассматриваемой части массива **A**;

`x` – разделитель.

При первом вызове процедуры полагается:

`right = n;` ***в C++ ->*** `right = n-1;`

`left = 1;` ***в C++ ->*** `left = 0;`

Сортировка Хоара (Pascal)

```
procedure Hoor(left, right:integer);
var i,j,x:integer;
begin
  i:=left; j:=right;
  x:=a[(left+right) div 2];
  repeat
    while a[i]<x do i:=i+1;
   {поиск с левой стороны элемента большего, чем разделитель}
    while x<a[j] do j:=j-1;
       {поиск с правой стороны элемента меньшего, чем разделитель}
    if i<=j then begin меняем местами a[i] и a[j]; i:=i+1;j:=j-1 end;
  until i>j;
  if left<j then Hoor(left,j);
           {применить процедуру сортировки для левой части массива}
  if i<right then Hoor(i,right);
           {применить процедуру сортировки для правой части массива}
end;
```

Сортировка Хоара (C/C++)

```
void hoor(int left, int right)
{
  int i=left;
  int j=right;
  int x=a[(left+right)/2];
  do {
    while (a[i]<=x) i++;
      //поиск с левой стороны элемента большего, чем разделитель
    while (x<a[j]) j--;
      //поиск с правой стороны элемента меньшего, чем разделитель
    if (i<=j) { меняем местами a[i] и a[j]; i++;j--; }
    }
  while (i<=j);
  if (left<j) hoor(left,j);
      //применить процедуру сортировки для левой части массива
  if (i<right) hoor(i,right);
      //применить процедуру сортировки для правой части массива
}
```

- Среднее время выполнения алгоритма:

$$T(n) = O(n \times \log n)$$

- Если массив почти упорядочен, время работы алгоритма может возрасти до квадратичного!
- Чтобы избежать этого, выбор разделителя производится **методом медианы случайной выборки**.
- Доказано, что **наилучший обмен ключами достигается**, **когда разделитель разбивает массив** по значениям «меньше разделителя» и «больше разделителя» **примерно на равные части**, т.е. **разделитель близок к медиане**.

Суть метода **медианы случайной выборки**:

- Из массива **произвольно выбирается группа элементов** (обычно до ста элементов), которые **сортируются любым простым методом**.
- Из **середины** отсортированной последовательности **выбирается элемент**, который далее используется в качестве разделителя.

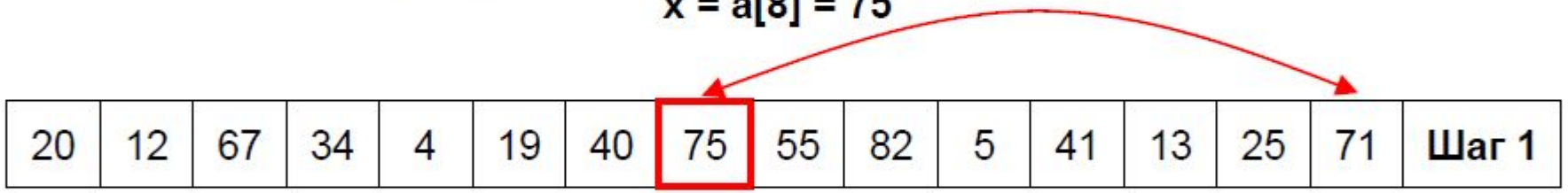
Быстрая обменная сортировка (сортировка Хоара) - Пример

Дан исходный массив:

20	12	67	34	4	19	40	75	55	82	5	41	13	25	71	Исходный массив

На 1-ом шаге выбран разделитель:

x = a[8] = 75



20	12	67	34	4	19	40	75	55	82	5	41	13	25	71	**Шаг 1**

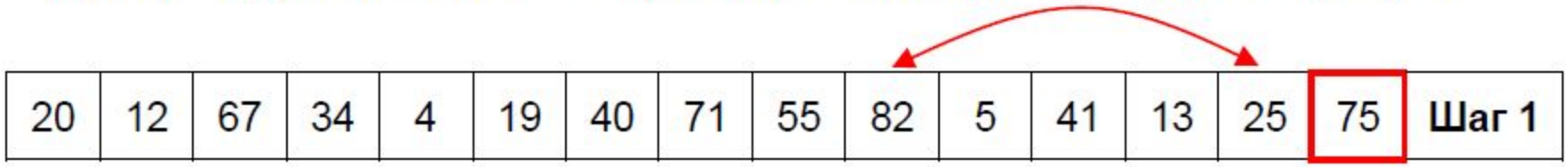
С левой стороны ищется элемент **a[i]>x**, **а с правой стороны** – элемент **a[j]<x**.

Если с одной стороны от разделителя найден элемент, который надо перенести на другую сторону, а с другой стороны такого элемента нет, то **меняются местами разделитель и элемент, стоящий неправильно.**

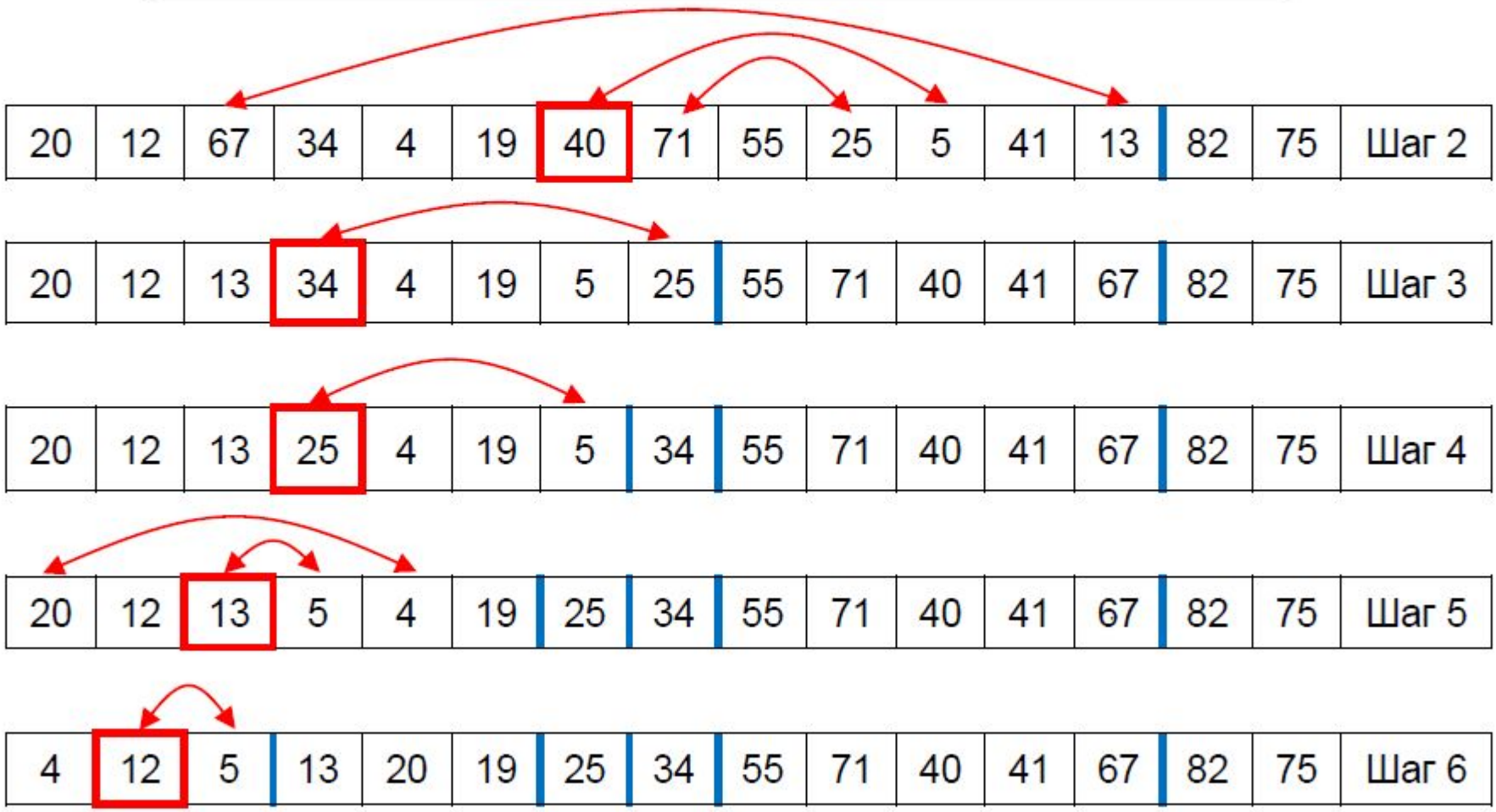
Далее продолжается поиск элементов **a[i]>x** и **a[j]<x**.

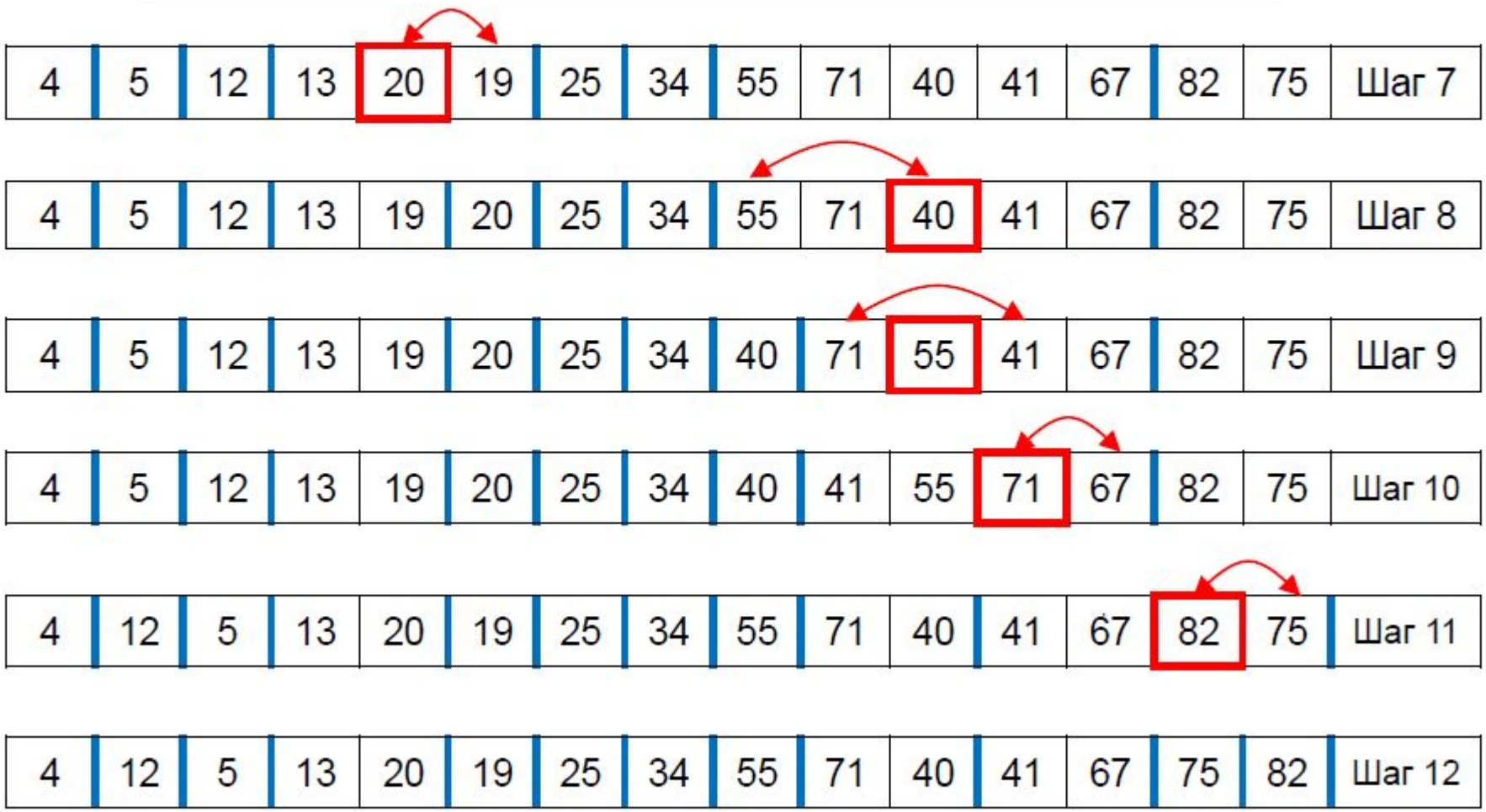
Если разделитель переместился, то перестановки относительно данного разделителя продолжаются до тех пор, пока есть что поменять местами.

Для разделителя **75** в примере имеем: **a[10]>x** и **a[14]<x**:



На 2-м шаге, когда все перестановки относительно x=75 произведены, массив делится на 2 подмассива, в каждом из них находится новый разделитель, и выполняется перестановка элементов.
Далее каждый из подмассивов снова делится на две части, и процесс повторяется:


20 12 67 34 4 19 40 71 55 25 5 41 13 82 75 Шаг 2
20 12 13 34 4 19 5 25 55 71 40 41 67 82 75 Шаг 3
20 12 13 25 4 19 5 34 55 71 40 41 67 82 75 Шаг 4
20 12 13 5 4 19 25 34 55 71 40 41 67 82 75 Шаг 5
4 12 5 13 20 19 25 34 55 71 40 41 67 82 75 Шаг 6


4 5 12 13 20 19 25 34 55 71 40 41 67 82 75 Шаг 7
4 5 12 13 19 20 25 34 55 71 40 41 67 82 75 Шаг 8
4 5 12 13 19 20 25 34 40 71 55 41 67 82 75 Шаг 9
4 5 12 13 19 20 25 34 40 41 55 71 67 82 75 Шаг 10
4 12 5 13 20 19 25 34 55 71 40 41 67 82 75 Шаг 11
4 12 5 13 20 19 25 34 55 71 40 41 67 75 82 Шаг 12

4.11. Быстрая сортировка вставкой – сортировка Шелла

- Сортировка Шелла является **видом сортировки вставкой с изменяющимся расстоянием между сравниваемыми ключами**.
- Наибольшее в начале, оно сокращается до 1 по мере упорядочения ключей.
- Этим достигается скорейшее продвижение ключей к своим истинным местам.
- **Временная сложность алгоритма:**

$$T(n) = O(n \times \log n)$$

https://www.youtube.com/watch?v=CmPA7zE8mx0

Идея алгоритма:

- Исходный массив разбивается **на несколько подмассивов**.
- В качестве ***подмассива*** выбираются элементы, удалённые на ***d*** шагов, т.е. значения индексов соседних элементов подмассива отличается на величину, равную ***d***.
- Сортируем массивы и уменьшаем ***d***, процесс продолжается до тех пор, пока ***d*** не станет равно 1.

- На эффективность сортировки Шелла существенным образом влияет выбор закона изменения расстояния ***d***.

Основное требование:

- расстояния $d_i, d_{i-1}, \ldots, d_1$ **не должны быть кратны** одно другому.
- Используют один из двух вариантов расчёта элементов последовательности $d_i, d_{i-1}, \ldots, d_1$.

Вариант №1

$d_i = 2d_{i-1} + 1$,

где $i = 2, \ldots, t$,

t – количество используемых расстояний,

$t = [\log_2 n] - 1$, где [b] – целая часть числа b, $d_1 = 1$;

Вариант №2

$d_i = 3d_{i-1} + 1$,

где $i = 2, \ldots, t$,

$t = [\log_3 n] - 1$, $d_1 = 1$.

Быстрая сортировка вставкой – сортировка Шелла - пример

Обозначения:

A – исходный массив;

t – количество расстояний;

d – массив расстояний;

x – вставляемый на текущем шаге элемент.

Сортировка Шелла (Pascal)

```
t:=trunc(log2(n))-1; {количество расстояний}
d[1]:=1;
for i:=2 to t do d[i]:=2*d[i-1]+1;
    {формирование последовательности d_i по первому варианту}
for m:=t downto 1 do {выбор текущего расстояния}
  begin
    k:=d[m];
    for i:=k+1 to n do
      begin
        x:=a[i]; {запомнить вставляемый ключ}
        j:=i-k;
        while (j>0) and (x<a[j]) do
          {сравнение элементов, находящихся на расстоянии d
           с вставляемым ключом}
              begin a[j+k]:=a[j]; j:=j-k end;
        a[j+k]:=x; {вставка ключа}
      end
  end;
```

Сортировка Шелла (C++)

```
t=(int)log((double)n)-1; //количество расстояний
d[0]=1;
for (i=1; i<t; i++) d[i]=2*d[i-1]+1;
  //формирование последовательности d_i по первому варианту
for (m=t-1;m>=0;m--) { //выбор текущего расстояния
  k=d[m];
  for (i=k;i<n;i++) {
    x=a[i]; //запомнить вставляемый ключ
    j=i-k;
    while ((j>=0) && (x<a[j])) {
          сравнение элементов, находящихся на расстоянии d
                с вставляемым ключом
       a[j+k]=a[j];
       j=j-k; }
    a[j+k]=x; //вставка ключа
} }
```

Первые 4 шага сортировки массива:

20	12	67	34	4	19	40	75	55	82	5	41	13	25	71	Исходный массив

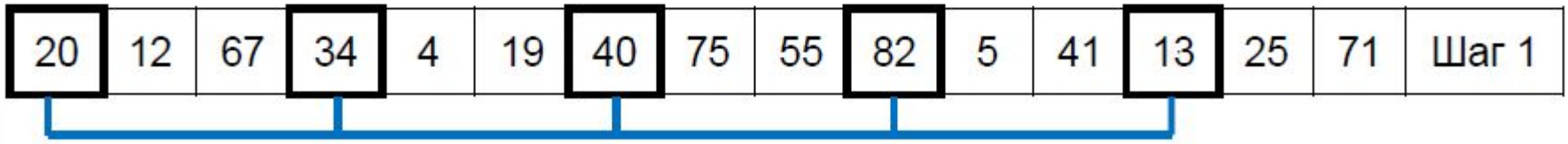
Рассчитываем ***t*** и ***d*** для массива из 15 элементов по формулам Варианта №1:

t = [log$_2$15] – 1 = 2

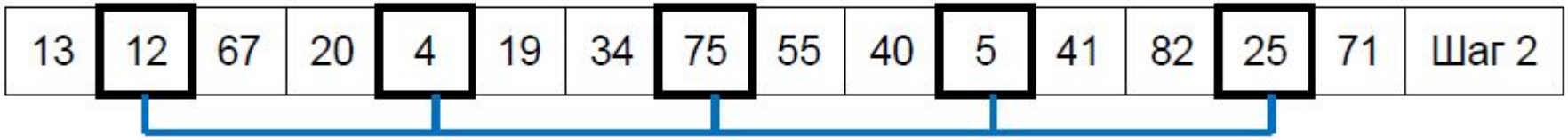
d[0] = 1;

d[1] = 3;

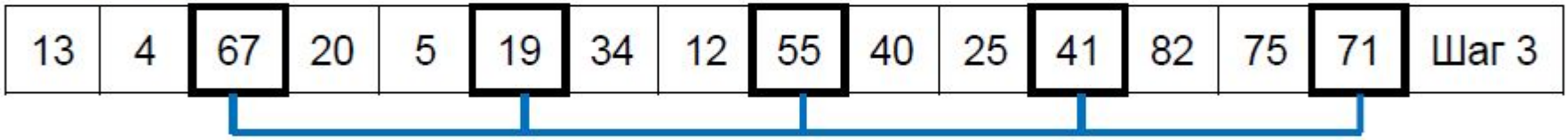
Шаг 1. Сортируем выделенные элементы простой вставкой:



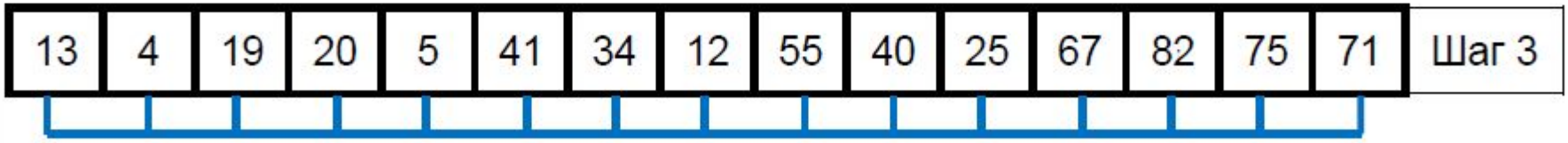
Шаг 2



Шаг 3



Шаг 4



Далее полученный массив сортируется простой вставкой.

Быстрые сортировки выбором

На практике используется несколько сортировок выбором.

Наиболее известные:

- «Турнир с выбыванием»;
- Пирамидальная сортировка.

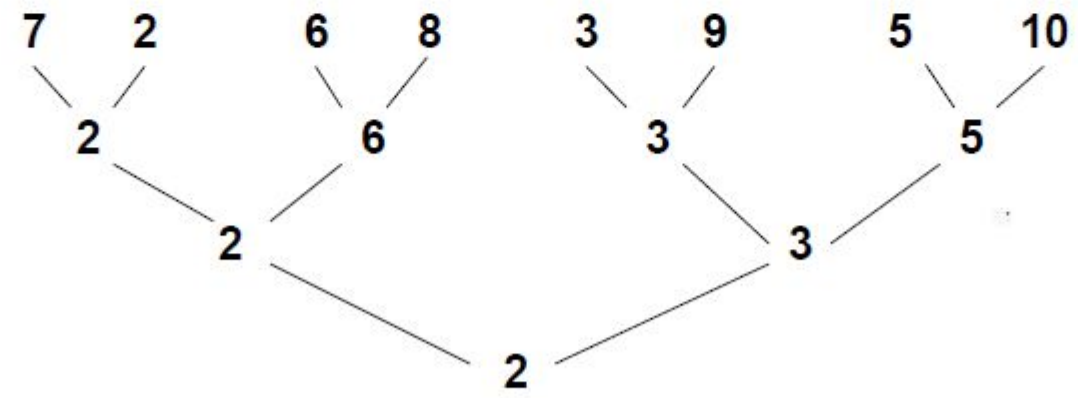
4.12. Сортировка «Турнир с выбыванием»

Идея алгоритма:

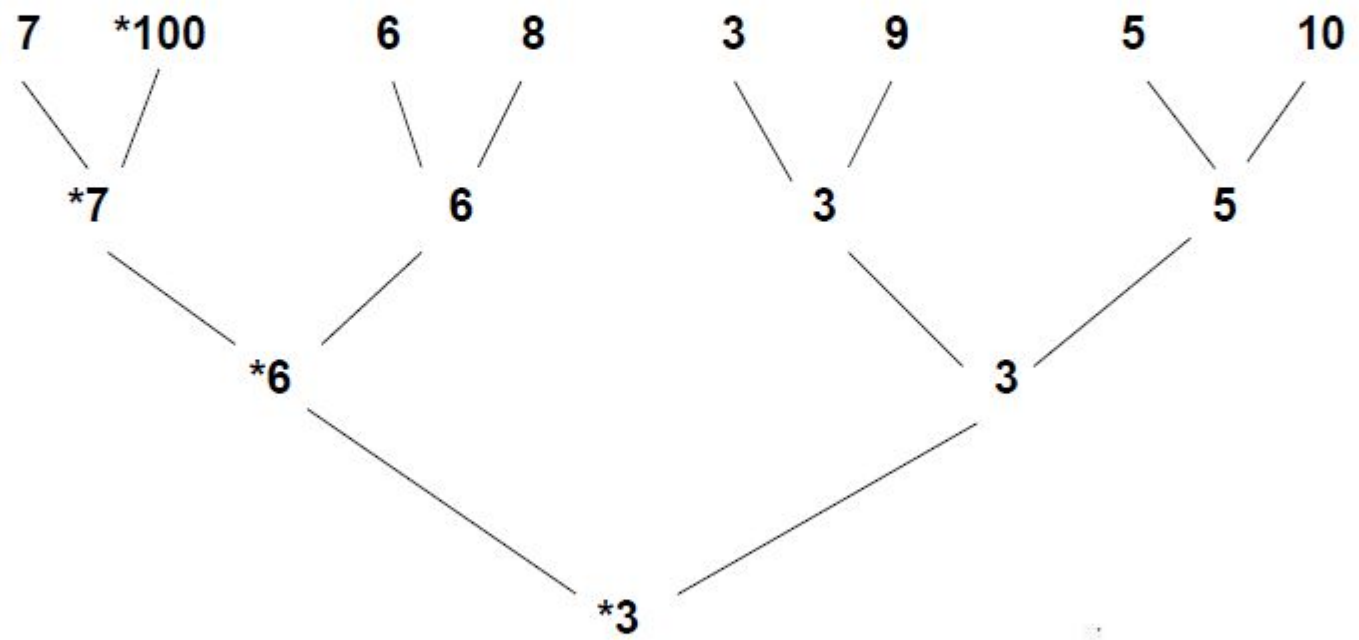
- Элементы массива разбиваются на пары.
- Из каждой пары выбирается ***победитель*** (меньший из элементов).
- Из победителей пар образуется новый массив, и процесс отбора победителей повторяется до тех пор, пока не будет найден победитель турнира.
- Этот победитель исключается из исходного массива и заносится в выходной массив.
- В изменённом исходном массиве находится новый победитель.
- Процесс повторяется до тех пор, пока в исходном массиве будут оставаться элементы.

Алгоритм «Турнир с выбыванием»

1. Сравниваем пары соседних ключей и запоминаем значение меньшего ключа из каждой пары.
2. Выполняем пункт 1 по отношению к значениям, полученным на предыдущем шаге, до тех пор, пока не определим наименьший ключ – «победитель турнира» и не построим дерево турни|



3. Вносим значение, найденное в п.2 в массив упорядоченных ключей (дополнительный массив).
4. **Проходим от корня к листу дерева**, следуя **по пути**, отмеченному значениями **«победителя турнира»** (на схеме отмечены *), и заменяем значение в листе на ***max*** – наибольшее допустимое целое число (например, 100).
5. **Проходим от листа к корню**, по пути обновляя значения в узлах дерева, и определяем нового победителя турнира.



6. Повторяем пункты 3-5, пока победителем не станет число **max = 100**.

Оценка временной сложности алгоритма:

- Для выполнения пунктов 1-2 потребуется сделать ***n-1*** сравнений.
- Для коррекции дерева потребуется не более ***log n*** сравнений, т.е. длина пути от корня к листу не превышает величину ***log n***.
- Дерево придётся корректировать ***n-1*** раз, поэтому временная сложность алгоритма равна:

$$T(n)=k1\times(n-1)+k2\times(\log n)\times(n-1)=O(n\times\log n),$$

k1, k2 – константы, зависящие от реализации алгоритма на компьютере.

Достоинства:

- **Быстрота.** Оценки худшего и среднего времени выполнения алгоритма совпадают.

Недостатки:

- **Дополнительный расход памяти** на хранение дерева турнира и результатов сортировки.

  Этот недостаток устранён в пирамидальной сортировке.

4.13. Пирамидальная сортировка

Выполняется на базе дерева.

Алгоритм состоит из 2-х этапов:

1. **Построение пирамиды** на месте исходного массива.
2. **Сортировка пирамиды**.

- На данном этапе **концевые элементы пирамиды меняются значениями**, и она укорачивается справа на один элемент, который является вершиной пирамиды.
- Полученный укороченный массив уже не может быть пирамидой, поэтому **снова делается пирамидой** с помощью специальной процедуры.
- Повторяя этот этап ***n*** раз, получаем отсортированный массив ***A***, в котором: ***a[1]<=a[2]<= … <=a[n]***

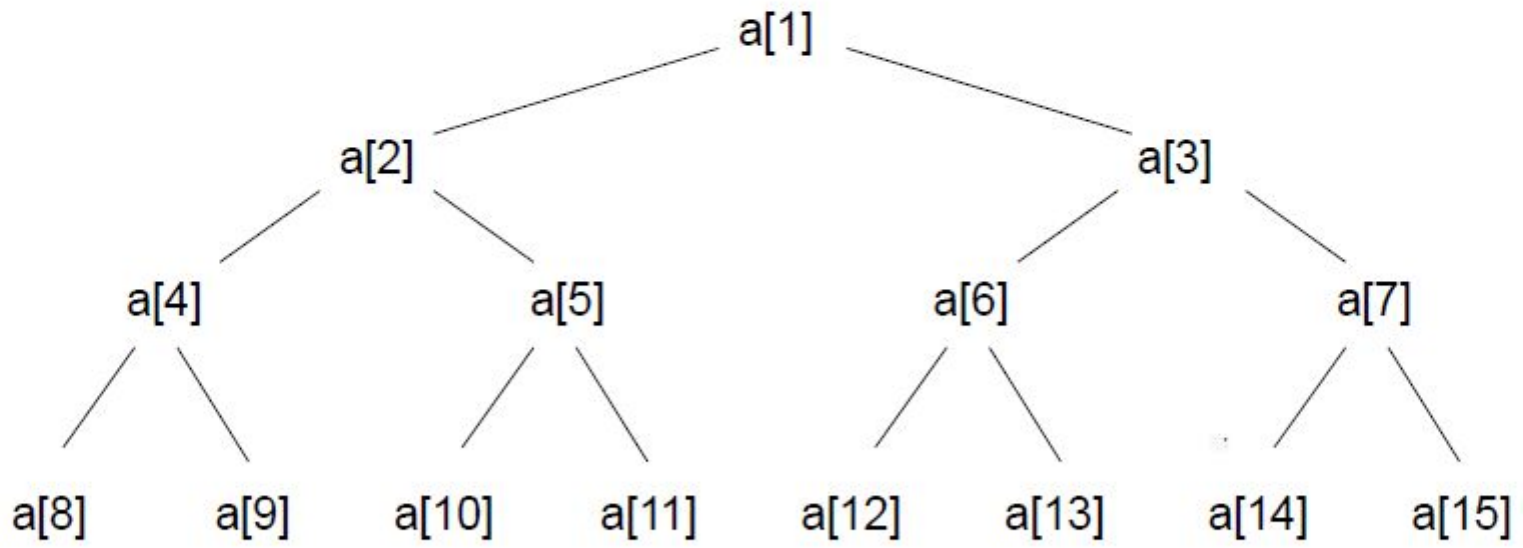
Пирамидальная сортировка

Бинарное дерево можно представить в виде одномерного массива, в котором:

a[1] – корень дерева;

a[2i] – левый сын **a[i]**;

a[2i+1] – правый сын **a[i]**.



- Данное дерево будет являться **пирамидой**, если для любого **i-го** элемента, имеющего потомков, выполняются неравенства:

$$a[i] >= a[2i]$$

$$a[i] >= a[2i+1]$$

Пусть имеется массив: $a_1, \dots, a_n$, причём его часть

$a_m, \dots, a_n$ **(m=n div 2 + 1)** уже образует пирамиду
(у этих элементов нет потомков – это нижний слой соответствующего дерева и для них никакой упорядоченности не требуется).

Далее пирамида расширяется влево, при этом каждый раз добавляется и сдвигами ставится в надлежащую позицию новый элемент.

Если у нас уже готова пирамида **a[k+1], … , a[n],**

то можно её расширить до пирамиды **a[k], … , a[n],** используя для a[k] процедуру добавления элемента.

Процедура добавления элемента в готовую пирамиду (Pascal)

```
procedure Shift(left,right:integer);
begin
  i:=left; j:=2*left; x:=a[left];
{Новый элемент x помещаем а вершину дерева, left – номер элемента,
включаемого в пирамиду,  right – номер последнего элемента массива}
if (j<right) and (a[j+1]>a[j]) then j:=j+1;
     {Условие j<right контролирует выход за пределы массива,
       определяется наибольший потомок}
while (j<=right) and (x<a[j]) do
   begin x:=a[i]; a[i]:=a[j]; a[j]:=x;
                {Обмен элементов массива}
      i:=j; j:=2*j;
      if (j<right) and (a[j+1]>a[j]) then j:=j+1
         {Определение нового узла}
end;
end;
```

Функция добавления элемента в готовую пирамиду (C++)

```
void shift(int left,right) {
int i,j,x;
i=left; j=2*left; x=a[left];
     // Новый элемент x помещаем в вершину дерева,
     // left - номер элемента, включаемого в пирамиду,
     // right - номер последнего элемента массива
if ((j<right) && (a[j+1]>a[j])) j++;
     //Условие j<right контролирует выход за пределы
     // массива, определяется наибольший потомок
while ((j<=right) && (x<a[j])) {
   x=a[i]; a[i]=a[j]; a[j]=x; //Обмен элементов массива
   i=j; j=2*j;
   if ((j<right) && (a[j+1]>a[j])) j++;
            //Определение нового узла
   }
a[i]=x;
}
```

Таким образом, процесс формирования пирамиды из ***n*** элементов $a_1, \ldots, a_n$, описывается следующим образом:

Pascal:

```
left:=n div 2;
while left>1 do
  begin
   left:=left-1;
   Shift(left,n)
  end;
```

C/C++:

```
left=n/2;
while (left>1)
{
 left--;
 shift(left,n);
}
```

- После превращения массива в пирамиду получается частично упорядоченная последовательность элементов.
- Для достижения полной упорядоченности надо проделать ***n*** сдвигающих шагов, причём после каждого шага на вершину (в корень) дерева помещается очередной наибольший элемент.
- В пирамиде первый элемент не меньше всех остальных.
- Для хранения «всплывающих» в корень элементов обменяем значениями первый и последний элементы пирамиды и укоротим пирамиду на один элемент справа.
- После этого укороченный массив может не быть пирамидой.
- Применим к нему процесс «просеивания» для элемента a_i.
- Преобразованная последовательность станет пирамидой.
- Повторяя этот процесс ***n-1*** раз отсортируем массив ***A*** по возрастанию.

Пирамидальная сортировка

Pascal:

```
right:=n;
while right>1 do
begin
x:=a[1];
a[1]:=a[right];
a[right]:=x;
right:=right-1;
Shift(1,right);
end;
```

C/C++:

Реализация пирамидальной сортировки (Pascal)

```
left:=n div 2; right:=n;
{определяем место элемента, у которого нет
потомков и номер последнего просматриваемого
элемента}
while left>1 do {пока не достигнем вершины}
   begin left:=left-1; Shift(left,right) end;
while right>1 do
   begin
      x:=a[1]; a[1]:=a[right]; a[right]:=x;
      right:=right-1;
      Shift(1,right)
end;
```

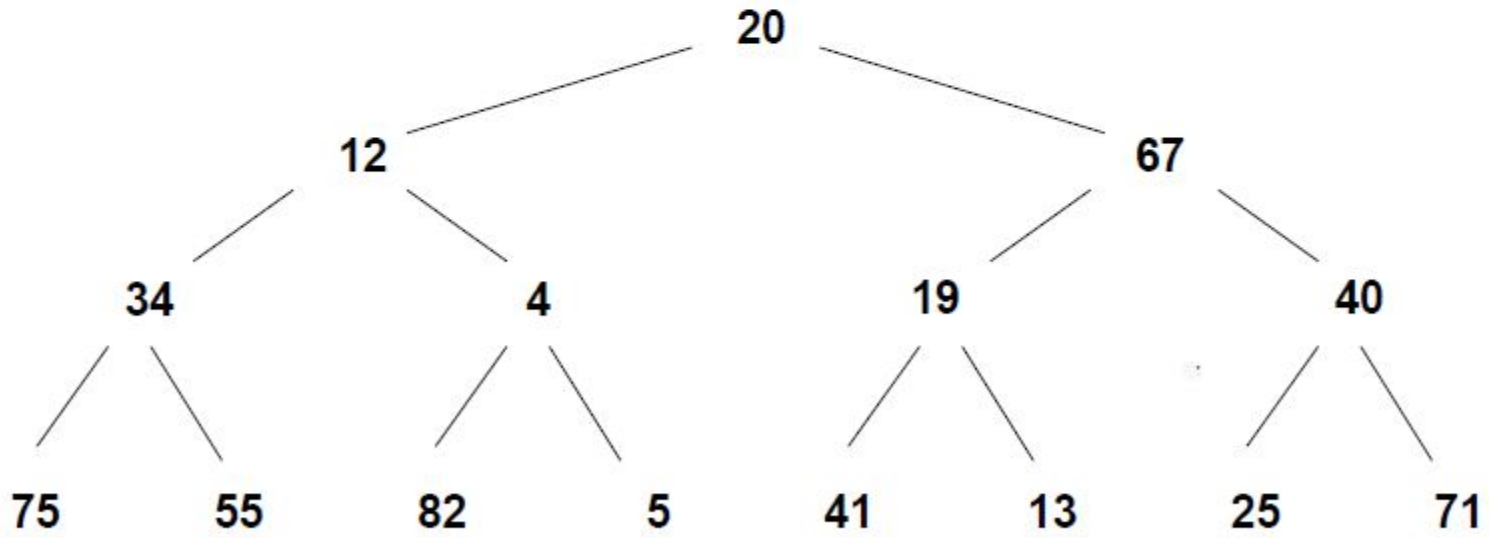
Реализация пирамидальной сортировки (C/C++)

```
left=n/2; right=n;
//определяем место элемента, у которого
нет потомков и номер последнего
просматриваемого элемента
while (left>1)//пока не достигнем
вершины
   { left--; shift(left,right); }
while (right>1) {
   x=a[1]; a[1]=a[right]; a[right]=x;
   right--;
   shift(1,right);
}
```

Пусть задан исходный массив вида:

20	12	67	34	4	19	40	75	55	82	5	41	13	25	71	Исходный массив

Этот массив можно представить в виде дерева:



- Данное дерево не является пирамидой, поэтому его надо изменить.
- Представим пирамиду, состоящую только из нижнего уровня, т.е. из 8 элементов

a[8], … , a[15]

75	55	82	5	41	13	25	71

- Далее будем добавлять в массив по одному элементу, вставляя его в нужное место.

Шаг 1

Добавляем элемент ***a[7]=40***.

У него два потомка ***a[14]=25*** и ***a[15]=71***.

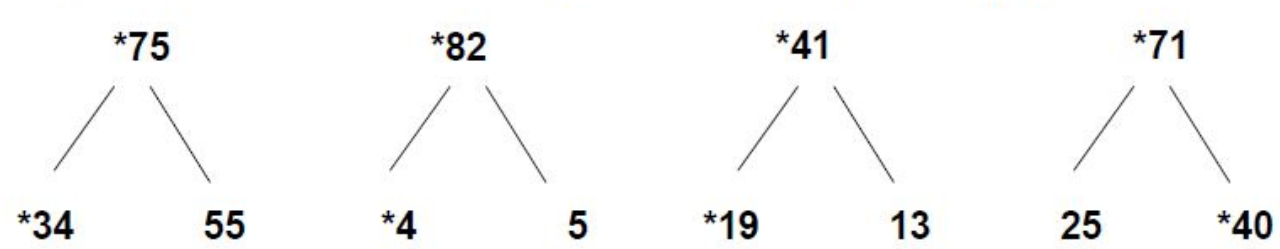
Т.к. **a[7]< max(a[14],a[15])**, то меняем местами ***a[7]*** и ***a[15]***.

Шаг 2

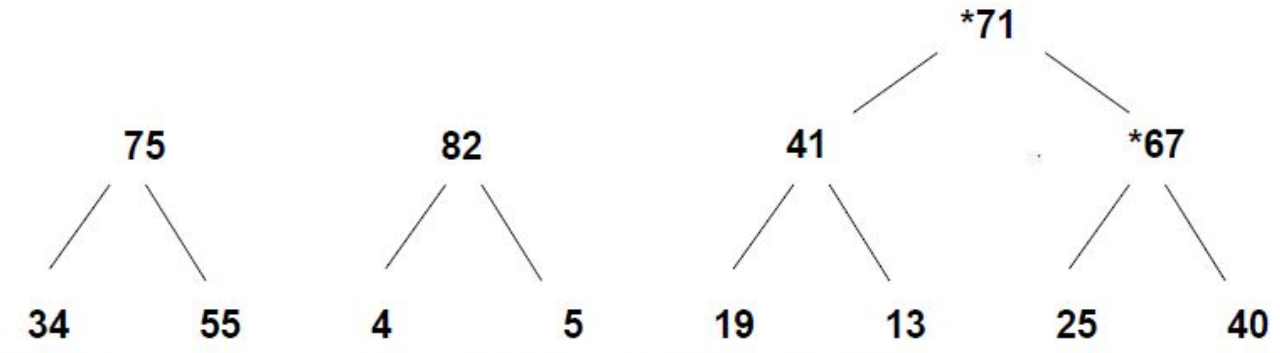
Добавляем элемент ***a[6]=19*** и меняем его местами с ***a[12]=41***.

Добавляем элемент ***a[5]=4*** и меняем его местами с ***a[10]=82***.

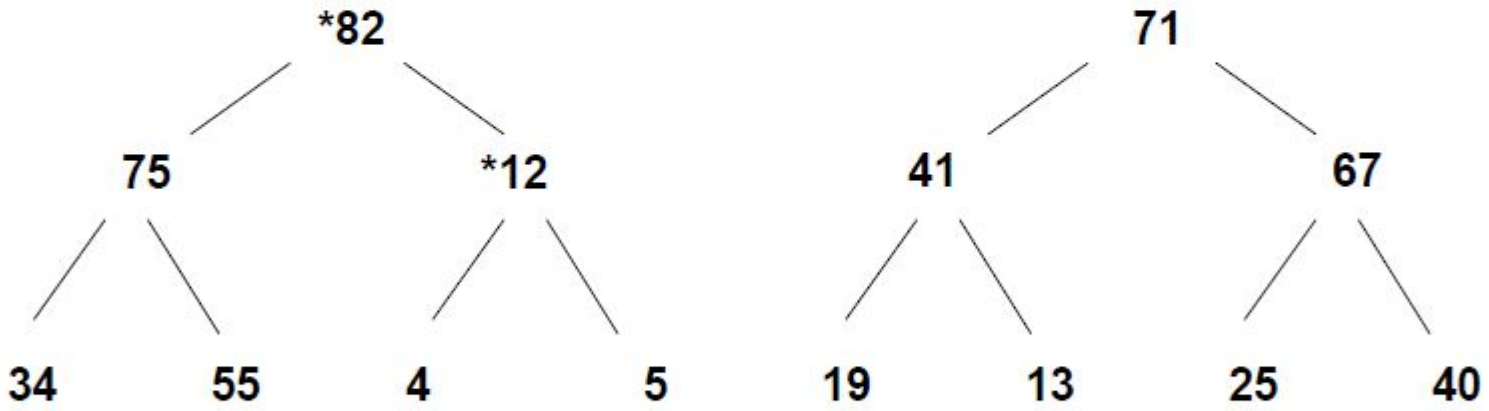
Добавляем элемент ***a[4]=34*** и меняем его местами с ***a[8]=75***.



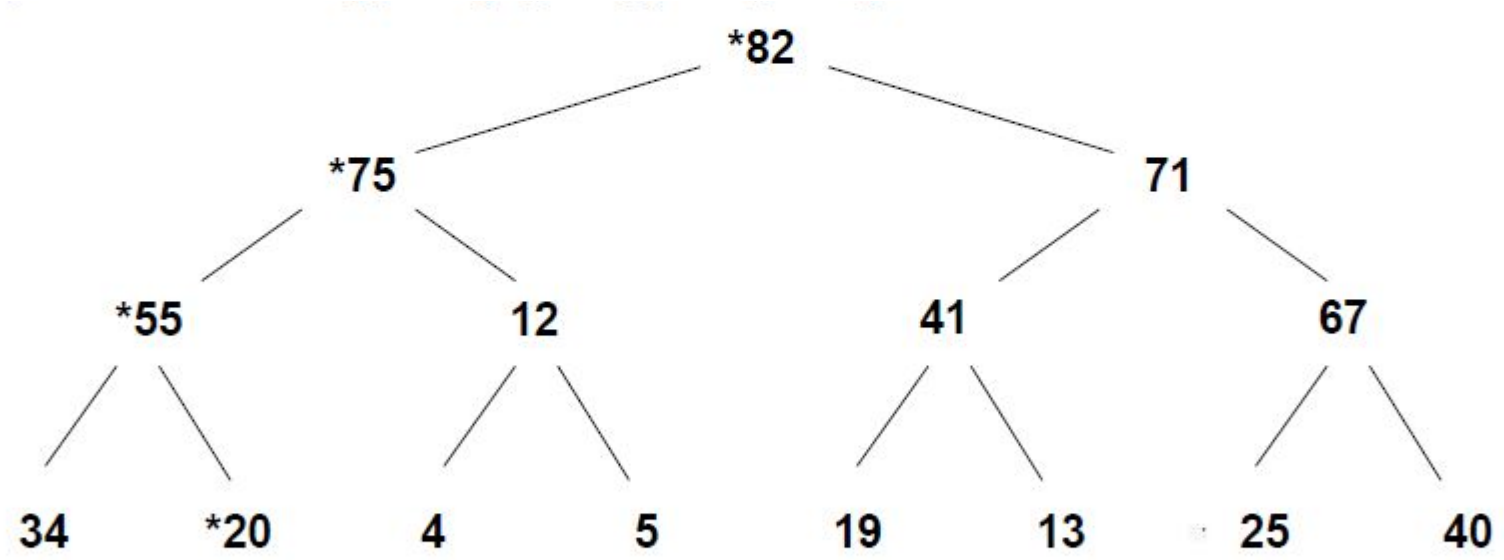
Шаг 3. Меняем местами ***a[3]=67*** и ***a[7]=71***.



Шаг 4. Меняем местами ***a[2]=12*** и ***a[5]=82***.

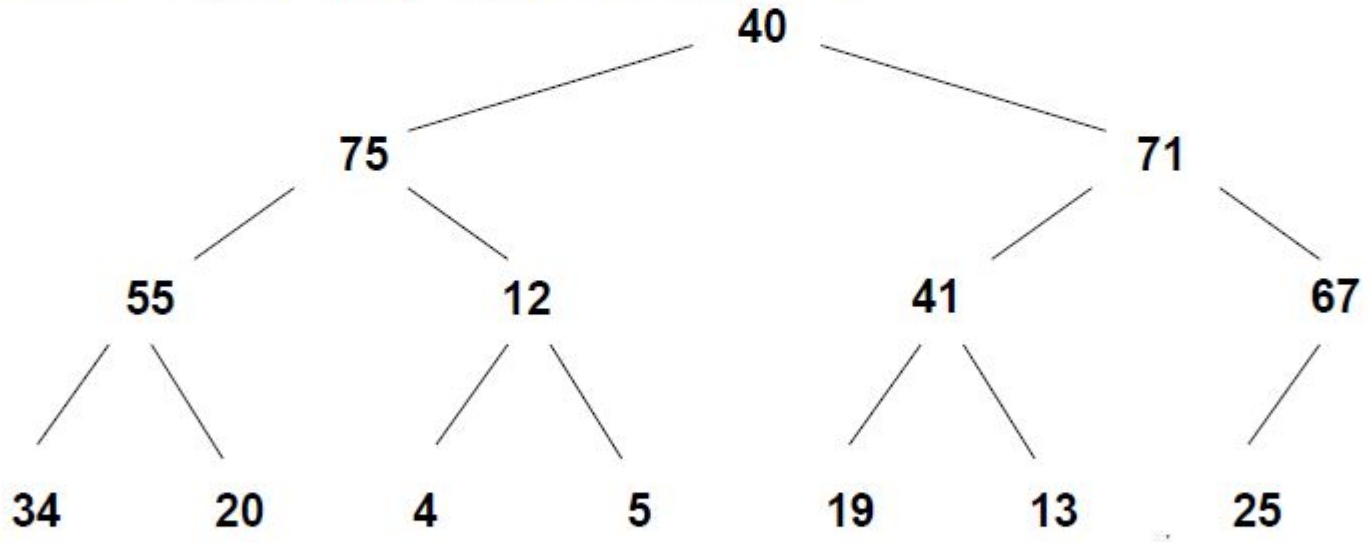


Шаг 4. Добавляем ***a[1]=20.*** Меняем местами ***a[1]*** и ***a[2], a[2]*** и ***a[4], a[4]*** и ***a[9]***. В итоге получаем пирамиду, которая удовлетворяет требованиям: a[i]>=a[2i] и a[i]>=a[2i+1].



В итоге максимальный элемент наверху. Построение пирамиды закончилось. Далее этап сортировки.

Меняем местами первый и последний элементы. В результате **в конце массива максимальный элемент** и на следующем этапе из рассмотрения и



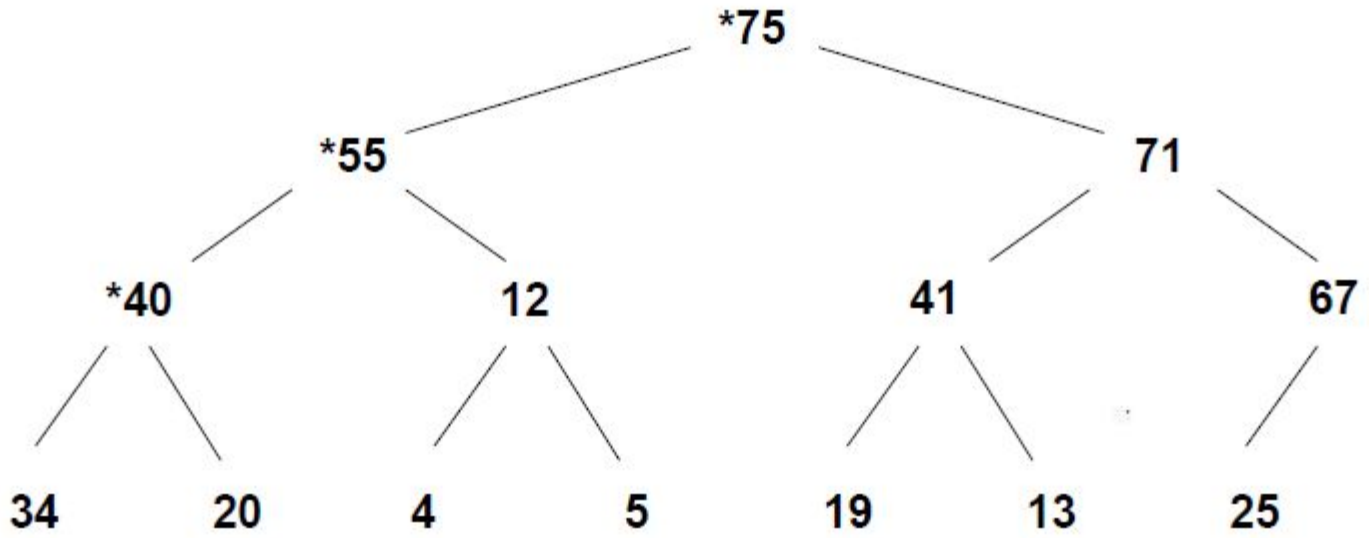
Массив A после перестановок:

40	75	71	55	12	41	67	34	20	4	5	19	13	25	82

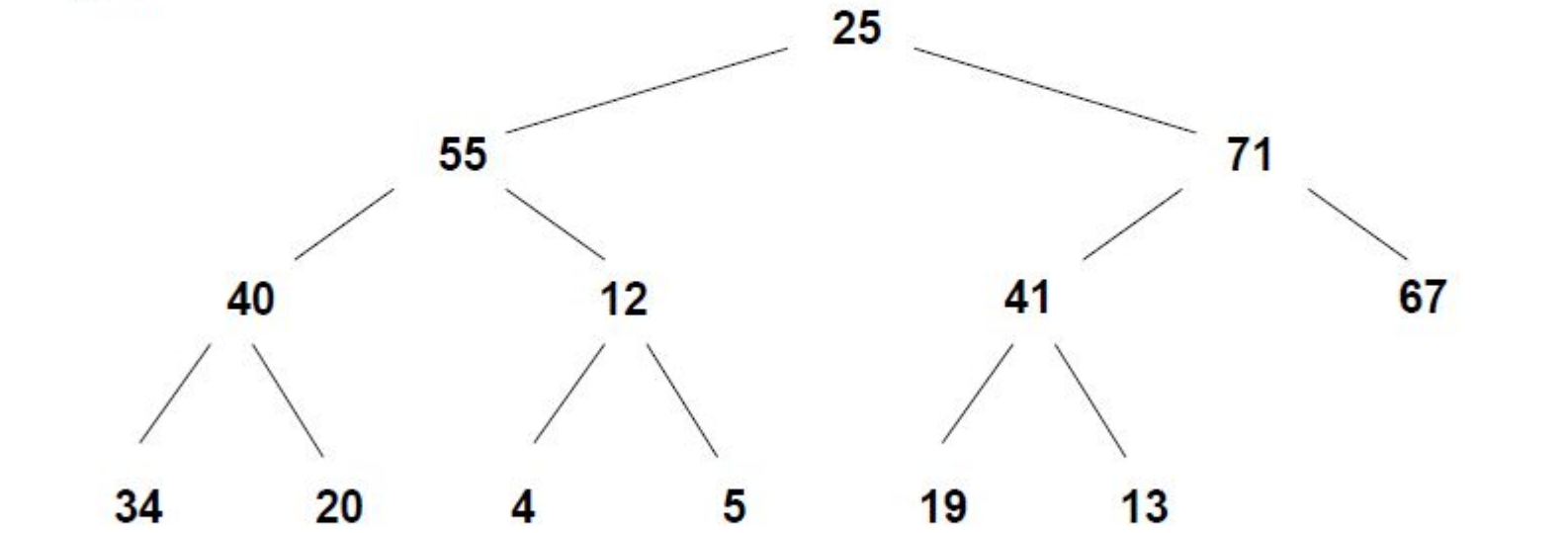
Полученное дерево – не пирамида.

Не на своём месте ***a[1]***, его надо «просеять».

Для этого меняем местами ***a[1]*** и ***a[2], a[2]*** и ***a[4].***



Меняем местами первый и последний элементы и получаем дерево и массив:



25	55	71	40	12	41	67	34	20	4	5	19	13	75	82

Действия повторяются пока массив не будет отсортирован.

Сравнительный анализ методов сортировки

- Сравнение быстродействия различных алгоритмов сортировки можно выполнить по ряду показателей:
  - **среднему** времени выполнения алгоритма;
  - **максимальному** времени выполнения алгоритма;
  - **минимальному** времени выполнения алгоритма;

- Достаточно часто анализ эффективности алгоритмов производится подсчётом **количества выполненных операций сравнения и пересылки**.

Формулы, определяющие количество операций, выполняемых с ключами при использовании простых методов сортировки

Наименование сортировки	Операция	Количество выполняемых операций		
		Минимальное	Максимальное	Среднее
Простая обменная	Сравнение	$(n^2 - n)/2$	$(n^2 - n)/2$	$(n^2 - n)/2$
	Пересылка	0	$1{,}5(n^2 - n)$	$0{,}75(n^2 - n)$
Простая вставкой	Сравнение	$n - 1$	$(n^2 + n - 2)/4$	$(n^2 - n)/2 - 1$
	Пересылка	$2(n - 1)$	$(n^2 - 9n - 10)/4$	$(n^2 + 3n - 4)/2$
Простая выбором	Сравнение	$(n^2 - n)/2$	$(n^2 - n)/2$	$(n^2 - n)/2$
	Пересылка	$3(n - 1)$	$n(\ln n + 0{,}75)$	$n^2/4 + 3(n - 1)$

Количество операций сравнения, выполняемых при использовании различных методов сортировки

Длина массива / Сортировка	Случайный массив		Упорядоченный массив		Массив, упорядоченный в обратном порядке	
	100	1000	100	1000	100	1000
Простая обменная (улучшенная)	10 098	1 000 998	200	2000	10 098	1 000 998
Простая вставкой	5 373	504 543	99	999	9 911	989 909
Простая выбором	9 999	999 999	9 999	999 999	9 999	999 999
Подсчётом	10 200	1 002 000	10 200	1 002 000	10 200	1 002 000
Двухпутевыми вставками	3 121	260 570	800	11 020	1 002	13 009
Шейкер-сортировка	7 947	738 634	198	1 998	9 949	999 219
Метод чётных и нечётных транспозиций	246	984 492	201	2 001	10 251	100 501
Метод квадратичного выбора	4 149	128 775	249	2 159	4 149	128 775
Быстрая сортировка вставкой (Шелла)	1 173	21 543	497	8 004	841	14 792
Быстрая обменная сортировка (Хоора)	995	15 609	729	11 020	738	11 031
Быстрая сортировка выбором (пирамида)	2 439	36 825	2 611	39 241	2 115	33 797

Сравнительный анализ методов сортировки

- **Для простых сортировок** при увеличении размера массива **в 10 раз** количество операций возрастает примерно **в 100 раз**.
- **Для быстрых сортировок** при увеличении размера массива **в 10 раз** количество операций возрастает примерно **в 15-19 раз**.

Временная сложность:

- простых сортировок - $O(n^2)$;
- быстрых сортировок - $O(n \times \log n)$.