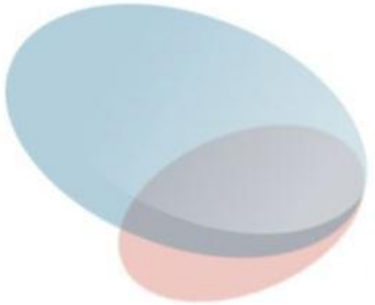


Алгоритмы и структуры данных
Лекция 2
Алгоритмы сортировки

1. Пирамидальная сортировка

Пирамидальная сортировка (heap sort (heap – куча)) – алгоритм сортировки, требующий при сортировке n элементов в худшем, среднем и в лучшем случае O(n log n) операций. Количество применяемой служебной памяти не зависит от размера массива (то есть, O(1)).

Пирамидальная сортировка значительно **улучшает базовый алгоритм (сортировку выбором**), используя структуру данных «**куча**» для ускорения поиска и удаления из рассмотрения максимального (минимального) элемента.

Пирамидальная сортировка может также рассматриваться как **усовершенствованная Bubblesort**, в которой элемент всплывает (max-heap) / тонет (min-heap) по многим путям.

1. Пирамидальная сортировка

В компьютерных науках **куча** – это специализированная **структура данных типа дерево**, которая удовлетворяет свойству (кучи): **если узел A является узлом-родителем узла B, то ключ(A) ≥ ключ(B).**

Из этого следует, что элемент с наибольшим ключом всегда является корневым узлом кучи, поэтому иногда такие кучи называют max-кучами.

В альтернативном случае, если сравнение перевернуть, то наименьший элемент будет всегда корневым узлом. Такие кучи называют **min-кучами**.

Не существует никаких ограничений относительно того, сколько дочерних узлов имеет каждый узел кучи, хотя на практике их число обычно не более двух.

Замечание. Структуру данных **куча** не следует путать с понятием куча в динамическом распределении памяти.

1. Пирамидальная сортировка

Сортировка пирамидой использует ***сортирующее дерево***, которое называется **пирамидой** и является частным случаем кучи.

- **Пирамида** — это полная *бинарная* куча, то есть для нее выполнены условия:

- каждый лист имеет глубину либо d, либо d − 1, где d — максимальная глубина дерева;
- значение в любой вершине больше (не меньше), чем значения её дочерних узлов.

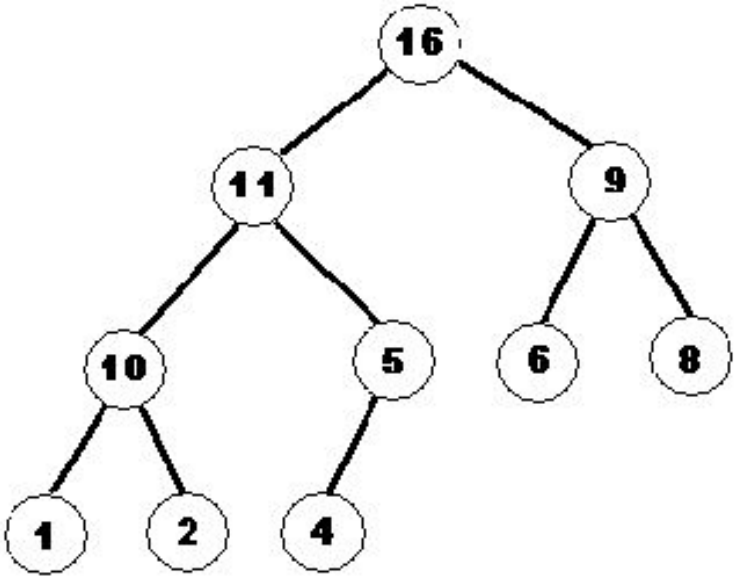
Пирамиды интересны сами по себе и полезны при реализации **очередей с приоритетом**.

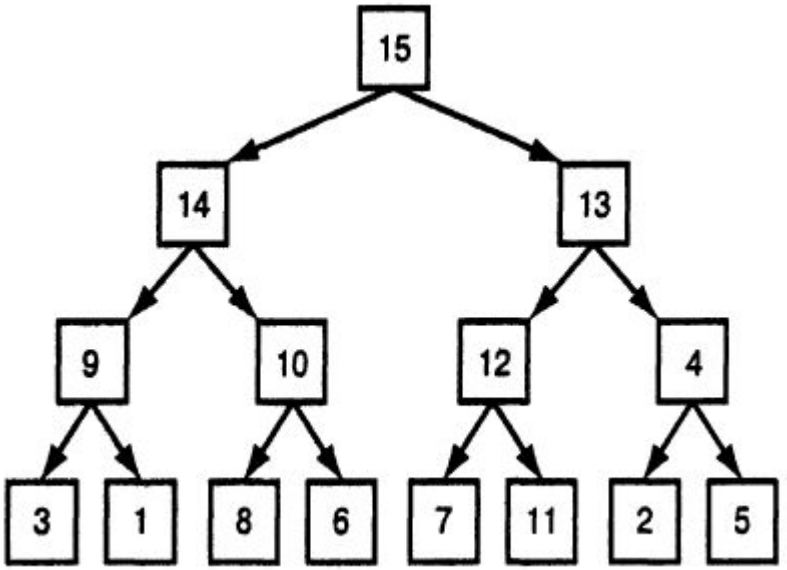
- Удобная структура данных для пирамиды — такой массив Array, что Array[1] — элемент в корне, а потомками элемента Array[i] являются элементы Array[2i] и Array[2i+1].

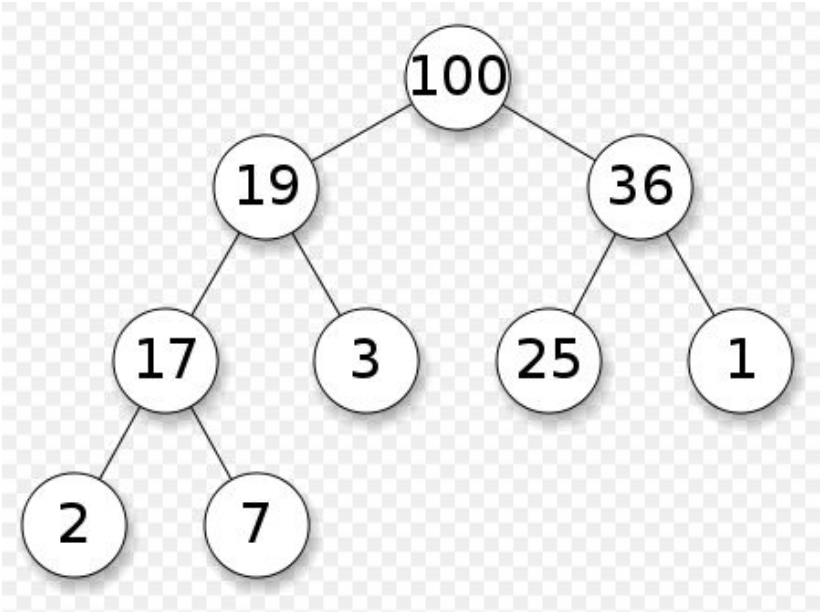
Это известное правило расположения полного двоичного дерева в массиве.

1. Пирамидальная сортировка

Примеры пирамид







Индекс	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Значение	15	14	13	9	10	12	4	3	1	8	6	7	11	2	5

1. Пирамидальная сортировка

Алгоритм подготовительного этапа пирамидальной сортировки – построение пирамиды

Можно построить пирамиду **снизу вверх**.

1. Вначале **разместим элементы исходного массива в виде двоичного дерева**, согласно известному правилу (по ширине).
2. Затем **сформируем пирамиды из небольших поддеревьев** (не более, чем с тремя узлами), вершины которых расположены на **предпоследнем** уровне дерева. Для этого сравним вершину маленького дерева с каждым из дочерних узлов. Если один из дочерних узлов больше, он меняется местами с родителем. Если оба дочерних узла больше, **больший дочерний узел меняется местами с родителем.**

Этот шаг повторяется до тех пор, пока все поддеревья, имеющие по 3 узла (не более 3-х узлов), не будут преобразованы в пирамиды.

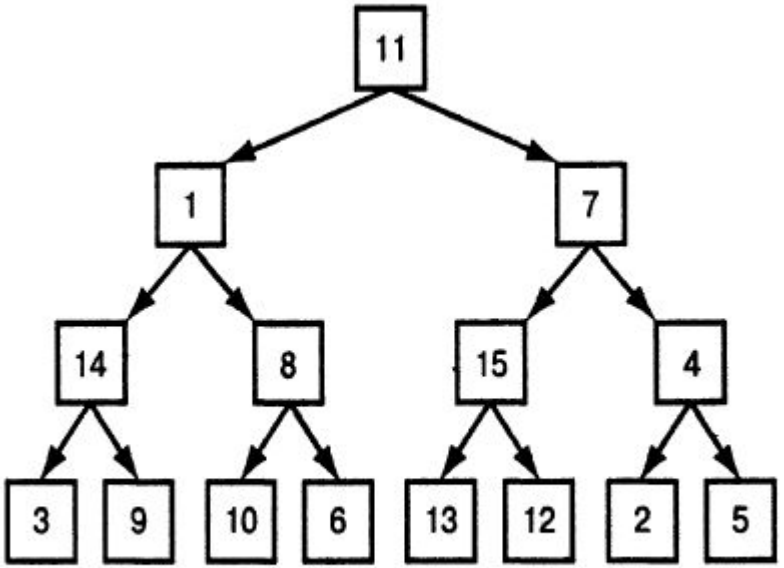
3. Теперь **объединим** (не более, чем две) **меньшие пирамиды и создадим более крупные пирамиды**. Сравним новую вершину с каждым из дочерних узлов. Если один из дочерних узлов больше, поменяем его местами с новой вершиной.

Если одно поддерево изменилось, проверяем, является ли оно все еще пирамидой. Для этого сравниваем его новую вершину с дочерними узлами и, если один из дочерних узлов больше, меняем его местами с новой вершиной.

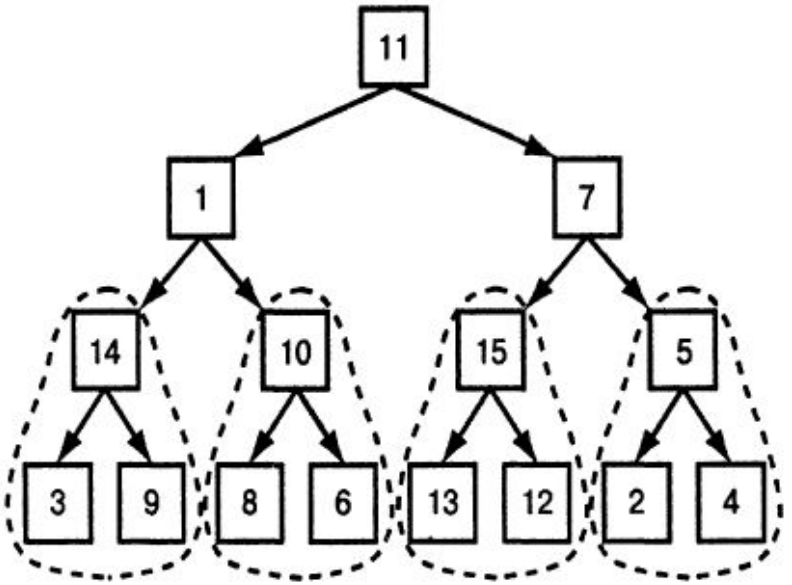
Продолжаем **перемещать новую вершину вниз**. В итоге, либо будет достигнута точка, в которой перемещаемый вниз узел больше обоих своих дочерних узлов, либо алгоритм достигнет основания дерева.

4. Продолжим объединение пирамид, образуя пирамиды большего размера до тех пор, пока все элементы не образуют одну большую пирамиду.

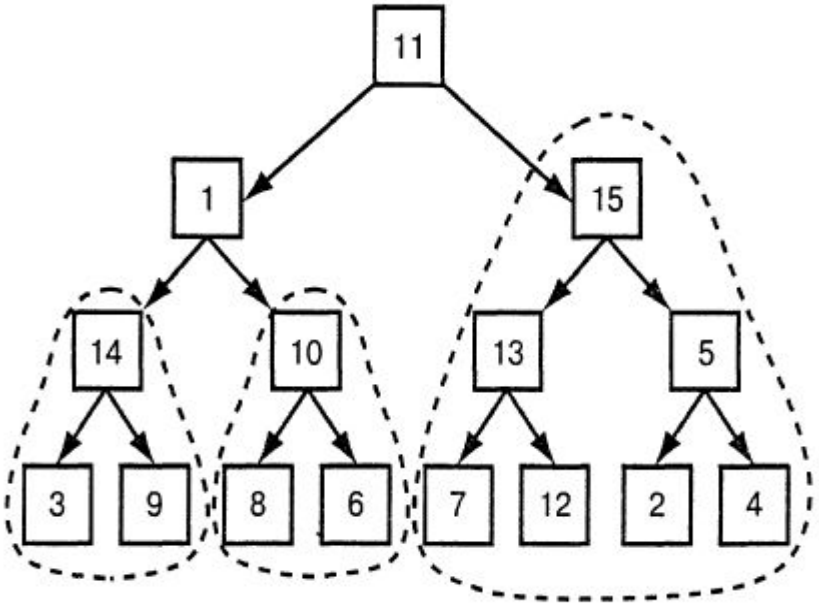
1. Пирамидальная сортировка



Несортированный список в полном дереве



Поддеревья второго уровня являются пирамидами



Объединение пирамид в пирамиду большего размера

1. Пирамидальная сортировка

Анализ пирамид

При первоначальном превращении списка в пирамиду осуществляется создание множества пирамид меньшего размера. **Для каждого внутреннего узла дерева строится пирамида с корнем в этом узле**. Если дерево содержит n элементов, то в дереве O(n) внутренних узлов, и в итоге **приходится создать O(n) пирамид**.

При создании каждой пирамиды может потребоваться продвигать элемент вниз по пирамиде, возможно до тех пор, пока он не достигнет концевого узла. Самые высокие из построенных пирамид будут иметь высоту порядка O(log(n)). Так как создается O(n) пирамид, и для построения самой высокой из них требуется O(log(n)) шагов, то **все пирамиды можно построить за время порядка O(n log(n)).**

Это грубая оценка. На самом деле времени потребуется меньше – порядка O(n).

1. Пирамидальная сортировка

Время для построения пирамиды

Пусть h — высота дерева. Это полное двоичное дерево, следовательно, h=log(n).

Для каждого узла, который находится на расстоянии h-i уровней от корня дерева, строится пирамида с высотой i. Всего таких узлов 2^(h-i) – количество узлов на (h-i)-м уровне, и всего **создается 2^(h-i) пирамид с высотой i**.

Перемещение элемента вниз по пирамиде с высотой i требует до i шагов. **Для пирамид с высотой i полное число шагов**, которое потребуется для построения 2^(h-i) пирамид, равно **i*2^(h-i)**.

Сложив все шаги, затрачиваемые на построение пирамид разного размера, получаем 1*2^(h-1)+2*2^(h-2)+3*2^(h-3) +…+(h-1)* 2^1. Вынеся за скобки 2^h, получим 2^h*(1/2+2/2^2+3/2^3+…+(h-1)/2^(h-1))<2^h*2=**n*2**.

Это означает, что **для первоначального построения пирамиды требуется порядка O(n) шагов.**

1. Пирамидальная сортировка

После того, как подготовительный этап завершен и пирамида построена, начинается непосредственно сортировка.

Алгоритм основного этапа пирамидальной сортировки

1. **Элемент, стоящий в вершине (в корне) пирамиды, меняем местами с последним элементом массива и уменьшаем счетчик количества элементов массива на единицу**, чтобы исключить из рассмотрения последний элемент.
2. Новый элемент на вершине может оказаться меньше, чем его потомки. Поэтому **алгоритм продвигает новый элемент *вниз* на его место**, используя алгоритм HeapPushDown.
3. Алгоритм **продолжает повторять шаги 1 и 2, то есть менять элементы местами и восстанавливать пирамиду до тех пор, пока в пирамиде не останется элементов.**

1. Пирамидальная сортировка

Полный алгоритм пирамидальной сортировки состоит из двух основных этапов:

1.Выстраиваем элементы массива в виде пирамиды:

Array[i] >= Array[2i]

Array[i] >= Array[2i+1]

при 1<=i<n/2.

Этот этап требует O(n) операций.

2.Будем удалять элементы из корня сортируемого дерева по одному за раз и *перестраивать дерево*. То есть на первом шаге обмениваем Array[1] и Array[n], преобразовываем Array[1], Array[2], … , Array[n-1] в пирамиду.

Затем переставляем местами Array[1] и Array[n-1], преобразовываем Array[1], Array[2], … , Array[n-2] в пирамиду.

Процесс продолжается до тех пор, пока в сортируемом дереве не останется один элемент. Тогда Array[1], Array[2],…,Array[n] — упорядоченная последовательность.

1. Пирамидальная сортировка

Время выполнения алгоритма пирамидальной сортировки

Первоначальное построение пирамиды требует **O(n log(n))** шагов.

После этого требуется **O(log(n)) шагов для восстановления пирамиды, когда один элемент продвигается вниз на свое место**.

Это **действие выполняется n раз**, поэтому **требуется всего порядка** O(n)*O(log(n))=**O(n*log(n)) шагов, чтобы получить из пирамиды упорядоченный список.**

Полное время выполнения для алгоритма пирамидальной сортировки составляет порядка O(n log(n))+O(n log(n))= **O(n log(n)).**

1. Пирамидальная сортировка

ПРИМЕР ПИРАМИДАЛЬНОЙ СОРТИРОВКИ

K_1	K_2	K_3	K_4	K_5	K_6	K_7	K_8	K_9	K_{10}	K_{11}	K_{12}	K_{13}	K_{14}	K_{15}	K_{16}
503	087	512	061	908	170	897	275	[653	426	154	509	612	677	765	703]
503	087	512	061	908	170	897	[703	653	426	154	509	612	677	765	275]
503	087	512	061	908	170	[897	703	653	426	154	509	612	677	765	275]
503	087	512	061	908	[612	897	703	653	426	154	509	170	677	765	275]
503	087	512	061	[908	612	897	703	653	426	154	509	170	677	765	275]
503	087	512	[703	908	612	897	275	653	426	154	509	170	677	765	061]
503	087	[897	703	908	612	765	275	653	426	154	509	170	677	512	061]
503	[908	897	703	426	612	765	275	653	087	154	509	170	677	512	061]
[908	703	897	653	426	612	765	275	503	087	154	509	170	677	512	061]
[897	703	765	653	426	612	677	275	503	087	154	509	170	061	512]	908
[765	703	677	653	426	612	512	275	503	087	154	509	170	061]	897	908
[703	653	677	503	426	612	512	275	061	087	154	509	170]	765	897	908
[677	653	612	503	426	509	512	275	061	087	154	170]	703	765	897	908
[653	503	612	275	426	509	512	170	061	087	154]	677	703	765	897	908
[612	503	512	275	426	509	154	170	061	087]	653	677	703	765	897	908
[512	503	509	275	426	087	154	170	061]	612	653	677	703	765	897	908
[509	503	154	275	426	087	061	170]	512	612	653	677	703	765	897	908
[503	426	154	275	170	087	061]	509	512	612	653	677	703	765	897	908
[426	275	154	061	170	087]	503	509	512	612	653	677	703	765	897	908
[275	170	154	061	087]	426	503	509	512	612	653	677	703	765	897	908
[170	087	154	061]	275	426	503	509	512	612	653	677	703	765	897	908
[154	087	061]	170	275	426	503	509	512	612	653	677	703	765	897	908
[087	061]	154	170	275	426	503	509	512	612	653	677	703	765	897	908

1. Пирамидальная сортировка

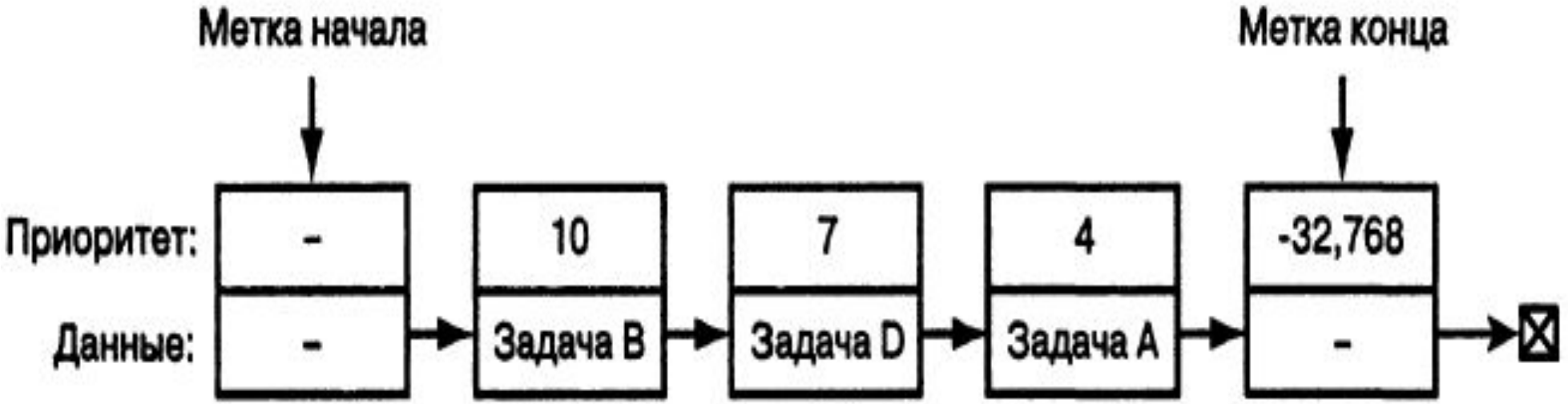
```
// Основная функция, выполняющая пирамидальную сортировку
void heapSort(int arr[], int n)
{
  // Построение кучи (перегруппируем массив)
    for (int i = n / 2 - 1; i >= 0; i--)
        heappushdown(arr, n, i);

   // Один за другим извлекаем элементы из кучи
    for (int i=n-1; i>=0; i--)
    {
        // Перемещаем текущий корень в конец
        swap(arr[0], arr[i]);

        // вызываем процедуру heappushdown на уменьшенной куче
        heappushdown(arr, i, 0);
    }
}
```

1. Пирамидальная сортировка

```
void heappushdown(int arr[], int n, int i)
{
    int largest = i;
// Инициализируем наибольший элемент как корень
    int l = 2*i + 1; // левый = 2*i + 1
    int r = 2*i + 2; // правый = 2*i + 2
 // Если левый дочерний элемент больше корня
    if (l < n && arr[l] > arr[largest])
        largest = l;
 // Если правый дочерний элемент больше, чем самый большой элемент на
     // данный момент
    if (r < n && arr[r] > arr[largest])
        largest = r;
 // Если самый большой элемент не корень
    if (largest != i)
    {
        swap(arr[i], arr[largest]);
 // Рекурсивно преобразуем в двоичную кучу затронутое поддерево
        heappushdown(arr, n, largest);
    }
}
```

Очереди с приоритетом

Куча является максимально **эффективной реализацией** абстрактного типа данных, который называется **очередью с приоритетом**.



Очередь с приоритетами на основе связанного списка

Очереди с приоритетом

Удаление элемента из очереди с приоритетом

Если в качестве очереди с приоритетом используется **пирамида**, легко найти **элемент с самым высоким приоритетом —** он всегда **находится на вершине пирамиды**.

Но если его удалить, получившееся дерево без корня уже не будет пирамидой.

Чтобы снова превратить дерево без корня в пирамиду, **поместим последний элемент** (самый правый элемент на нижнем уровне) **в вершину пирамиды**. Затем при помощи алгоритма HeapPushDown **продвинем новый корневой узел *вниз* по дереву** до тех пор, пока дерево снова не станет пирамидой. В этот момент, можно получить **на выходе очереди с приоритетом следующий элемент с наивысшим приоритетом.**

Очереди с приоритетом

Добавление элемента к очереди с приоритетом

Поместим новый элемент на свободное место в *конце* массива. Полученное дерево может не быть пирамидой.

Чтобы снова преобразовать его в пирамиду, сравним новый элемент с его родителем. **Если новый элемент больше родителя, поменяем их местами**. Если элемент больше родителя, то он также больше и второго потомка.

Продолжим сравнение нового элемента с родителем и перемещение его по дереву *вверх* к корню, пока не найдется родитель, больший, чем новый элемент (или пока не достигнем корня). В этот момент, дерево снова представляет собой пирамиду.

Очереди с приоритетом

Время для удаления максимального элемента

Для удаления элемента из очереди с приоритетом, **последний элемент перемещается на вершину дерева.** Затем **продвигается *вниз***, пока не займет свое окончательное положение, и дерево снова не станет пирамидой. Так как дерево имеет высоту log(n), процесс может занять не более log(n) шагов. Это означает, что удаление **элемента из очереди с приоритетом на основе пирамиды осуществляется за O(log(n)) шагов.**

Время добавления элемента к очереди с приоритетом

При добавлении в пирамиду **новый элемент помещается внизу дерева и передвигается к вершине**, пока не займет нужное место (максимум за log(n) шагов). То есть, **новый элемент добавляется к очереди с приоритетом на основе пирамиды тоже за время порядка O(log(n)).**

2. Сортировка подсчетом

Сортировка подсчетом (counting sort) — специализированный алгоритм, который очень хорошо работает, если **элементы данных — целые числа, значения которых находятся в относительно узком диапазоне**, например, если значения находятся между 1 и 1000.

Выдающаяся скорость сортировки подсчетом, значительно быстрее быстрой сортировки, достигается за счет того, что при этом **не используются операции сравнения элементов**.

Без использования операций сравнения элементов, алгоритм сортировки подсчетом позволяет упорядочивать элементы за время порядка O(n).

Для сравнения: время выполнения любого алгоритма сортировки, использующего операции сравнения, порядка O(n log(n)).

2. Сортировка подсчетом

Алгоритм сортировки подсчетом

1 шаг. **Создается массив для подсчета числа элементов, имеющих определенное значение**. Если значения элементов сортируемого массива List находятся в диапазоне между min_value и max_value, алгоритм создает массив Counts с нижней границей индекса min_value и верхней границей индекса max_value. **Если существует m значений элементов,** массив Counts содержит m записей, и **время выполнения этого шага порядка O(m).**

2 шаг. **Вычисляется, сколько раз в списке встречается каждое значение.** Для каждого значения i, i=min_value, …,max_value, сортируемого массива, в массиве Counts увеличивается значение соответствующей записи Counts(List(i)). Так как **этот шаг просматривает все записи в исходном массиве, он требует порядка O(n) шагов.**

3 шаг. Обходится массив счетчиков Counts и **помещается соответствующее число элементов в отсортированный массив**. Для каждого значения i, i=min_value, … ,max_value, в исходный массив помещается Counts(i) элементов со значением i. Так как **этот шаг** помещает по одной записи в каждую позицию в **сортируемом** массиве, он **требует порядка O(n) шагов**.

2. Сортировка подсчетом

Время работы алгоритма

Алгоритм целиком **требует порядка** O(m)+O(n)+O(n)=**O(m+n) шагов**.

Если m<<n, то **O(m+n)=O(n),** что довольно быстро.

Пример. Если n=100000 и m=1000, то m+n=101000, тогда как n ln(n) > 1миллиона (для алгоритмов, использующих сравнения).

Особенности алгоритма

- **Шаги**, выполняемые алгоритмом сортировки подсчетом, **относительно просты** по сравнению с шагами быстрой сортировки.
- Сортировка подсчетом опирается на тот факт, что значения данных — целые числа, поэтому этот **алгоритм не может сортировать данные других типов.**

2. Сортировка подсчетом

```
type
   TCountArray = array [1..10000000] of Longint;
   PCountArray = ^TCountArray;

procedure Countingsort(list : PLongintArray; counts : PCountArray;
                       min, max, min_value, max_value : Longint);
var
   i, j, new_index : Longint;
```

2. Сортировка подсчетом

```
begin
    // Установка счетчиков в 0.
    for i := min_value to max_value do
        counts^[i] := 0;

    // Подсчет значений.
    for i := min to max do
        counts^[list^[i]] := counts^[list^[i]] + 1;

    // Помещение значений в правильную позицию.
    new_index := min;
    for i := min_value to max_value do
        for j := 1 to counts^[i] do
        begin
            list^[new_index] := i;
            new_index := new_index + 1;
        end;
end;
```

3. Блочная сортировка

Блочная сортировка (карманная сортировка, корзинная сортировка, англ. Bucket sort) — алгоритм сортировки, в котором **сортируемые элементы распределяются между конечным числом отдельных блоков** (карманов, корзин) **так, чтобы все элементы в каждом следующем по порядку блоке были всегда больше** при сортировке по твозрастанию, **чем в предыдущем**. Каждый блок затем сортируется отдельно, либо рекурсивно тем же методом, либо другим. Затем элементы помещаются обратно в массив.

Особенности алгоритма

- Этот тип сортировки может обладать **линейным временем исполнения.**
- Не использует операций **сравнения элементов** (в основной части алгоритма).
- Данный алгоритм **требует знаний о природе сортируемых данных**, выходящих за рамки функций "сравнить" и "поменять местами", достаточных для сортировки слиянием, сортировки пирамидой, быстрой сортировки, сортировки Шелла, сортировки вставкой.

  Например, **знание значений максимального и минимального элементов**.

3. Блочная сортировка

Алгоритм

- **Алгоритм использует значения элементов для разбиения их на множество блоков,** и затем последовательно **рекурсивно сортирует полученные блоки**.
- **Когда блоки становятся достаточно малыми, алгоритм останавливается** и использует более простой алгоритм типа сортировки выбором для завершения процесса.
- **Отсортированный массив получается путем последовательного перечисления элементов каждого блока.**

Предположения

- Для деления массива на блоки алгоритм предполагает, что **значения данных распределены равномерно**, и распределяет элементы по блокам тоже равномерно.
- Поскольку входные числа распределены равномерно, предполагается, что в каждый блок попадет приблизительно одинаковое количество чисел.

3. Блочная сортировка

Сложность алгоритма

Если в списке n элементов, и алгоритм использует n блоков, в каждый блок попадает всего один или два элемента. Программа может отсортировать их за конечное число шагов, поэтому **время выполнения алгоритма в целом порядка O(n).**

Преимущества алгоритма

- Относится к классу быстрых алгоритмов с **линейным временем выполнения O(n)** (на удачных входных данных).

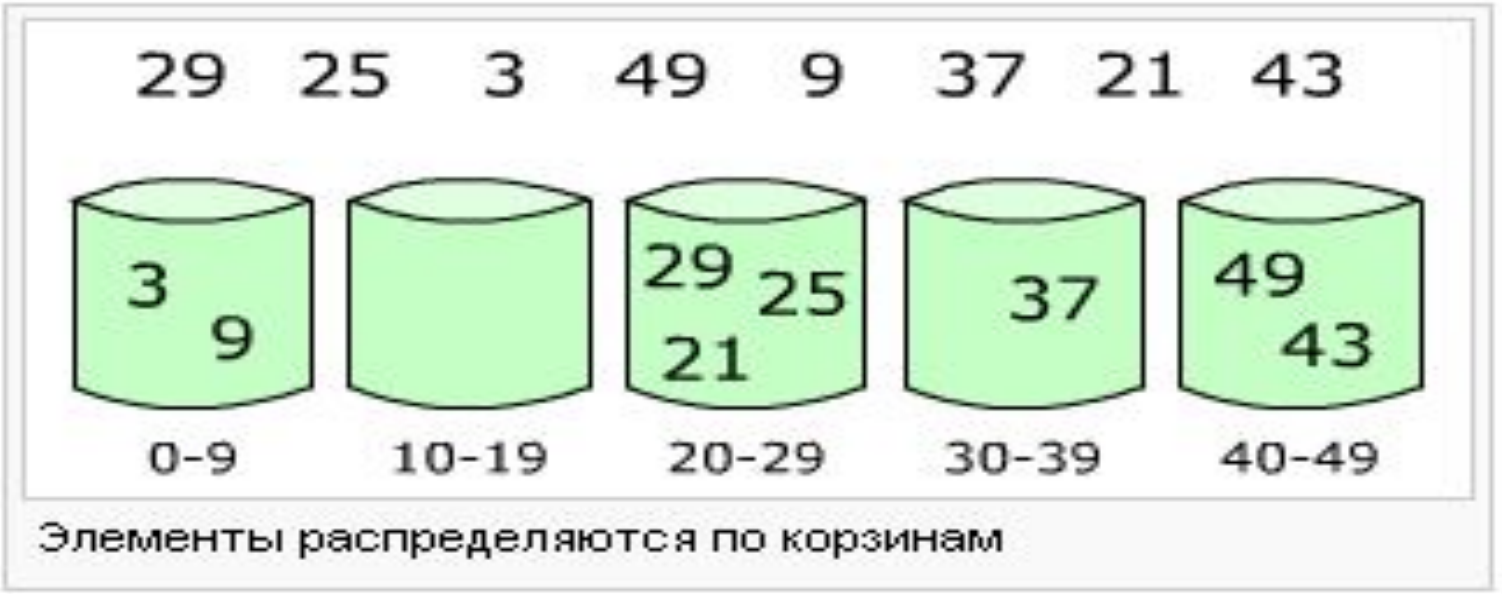
Недостатки алгоритма

- Сильно **деградирует при** большом количестве мало отличных **(одинаковых) элементов**, или на неудачной функции получения номера блока по содержимому элемента.
- Проблемы могут возникать, если массив содержит небольшое число различных значений.

  Например, если все элементы имеют одно и то же значение, они все будут помещены в один блок. Если алгоритм не обнаружит это, он снова и снова будет помещать все элементы в один и тот же блок, вызвав бесконечную рекурсию и исчерпав все стековое пространство.

3. Блочная сортировка

Пример. Число блоков (корзин) меньше числа элементов.



Элементы распределяются по корзинам

Затем элементы в каждой корзине сортируются

3. Блочная сортировка

Пример. Число блоков равно числу элементов

Неупорядоченный список	1	74	38	72	63	100	89	57	7	31

Номер блока	1	2	3	4	5	6	7	8	9	10
Блок	1 7		38 31		57	63	74 72	89		100

Помещение элементов в блоки

Упорядоченный массив 1 7 31 38 57 63 72 74 89 100

3. Блочная сортировка

Реализовать **алгоритм блочной сортировки** можно различными способами.

Блочная сортировка на основе одномерного массива

Блочную сортировку можно реализовать в массиве, используя идеи подобные тем, которые используются при сортировке подсчетом.

1. При каждом вызове алгоритма, **вначале подсчитывается число элементов, которые относятся к каждому блоку**.
2. Далее на основе этих данных **во временном массиве рассчитываются смещения**, которые затем используются для правильного расположения элементов в массиве.
3. В конце **блоки рекурсивно сортируются**, и **отсортированные данные перемещаются** обратно **в исходный массив**.

3. Блочная сортировка

Блочная сортировка с использованием связанных списков

Можно использовать в качестве блоков связанные списки. Это облегчает перемещение элементов из одного блока в другой в процессе работы алгоритма.

Этот метод может быть более сложным, если элементы изначально расположены в массиве. В этом случае, необходимо перемещать элементы из массива в связанный список и обратно в массив после завершения сортировки. **Для создания связанного списка также требуется дополнительная память**.

3. Блочная сортировка

```
type
   PCell = ^TCell;
   TCell = record
      Value : Longint;          // Данные.
      NextCell : PCell;         // Следующая ячейка.
   end;

   TCellArray = array [1..1000000] of TCell;
   PCellArray = ^TCellArray;

procedure LLBucketsort(top : PCell);
var
   count, min_value, max_value : Longint;
   i, value, bucket_num : Longint;
   cell, nxt : PCell;
   bucket : PCellArray;
   scale : Double;
begin
   cell := top^.NextCell;
   if (cell = nil) then exit;
```

3. Блочная сортировка

```
// Подсчет ячеек и поиск минимального и максимального значений.
count := 1;
min_value := cell^.Value;
max_value := min_value;
cell := cell^.NextCell;
while (cell<>nil) do
begin
    count := count + 1;
    value := cell^.value;
    if (min_value > value) then min_value := value;
    if (max_value < value) then max_value := value;
    cell := cell^.NextCell;
end;

// Если min_value = max_value, то имеется только одно значение,
// поэтому список отсортирован.
if (min_value = max_value) then exit;

// Если список содержит не более Cutoff ячеек, заканчиваем
// сортировку с помощью LLInsertionsort.
if (count <= Cutoff) then
begin
    LLInsertionsort(top);
    exit;
end;
```

3. Блочная сортировка

```
// Размещение пустых блоков.
GetMem(bucket,count * SizeOf(TCell));
for i := 1 to count do bucket^[i].NextCell := nil;

// Перемещение ячеек в блоки.
Scale := (count - 1) / (max_value - min_value);
Cell := top^.NextCell;
while (cell<>nil) do
begin
    nxt := cell^.NextCell;
    value := cell^.value;
    if (value = max_value) then
        bucket_num := count
    else
        bucket_num := Trunc((value - min_value) * scale) + 1;
    cell^.NextCell := bucket^[bucket_num].NextCell;
    bucket^[bucket_num].NextCell := cell;
    cell := nxt;
end;
```

3. Блочная сортировка

```
    // Объединение сортированных списков.
    top^.NextCell := bucket^[count].NextCell;
    for i := count - 1 downto 1 do
    begin
        cell := bucket^[i].NextCell;
        if (cell<>nil) then
        begin
            nxt := cell^.NextCell;
            while   nxt<>nil   do
            begin
                cell := nxt;
                nxt := cell^.NextCell;
            end;
            cell^.NextCell := top^.NextCell;
            top^.NextCell := bucket^[i].NextCell;

            // Освобождаем метку конца, если она есть.
            if (nxt<>nil) then Dispose(nxt);
        end;
    end;

    // Освобождаем память, выделенную для блоков.
    FreeMem(bucket);
end;
```

4. Распределяющая сортировка

Распределяющая (поразрядная) сортировка относится **к быстрым алгоритмам, *не использующим операции сравнения***, с **временем выполнения порядка O(n)** и является разновидностью блочной сортировки.

Пусть **элементы** массива B есть **T-разрядные положительные десятичные числа**.

D(j, N) — j-я справа цифра в десятичной записи числа N>=0,

т.е. D(j, N) = trunc (N/m) mod 10, где m=10^(j-1),

или D(j, N)= floor (N/m)%10, где m=10^(j-1) .

Пусть B0,B1,...,B9 — вспомогательные массивы или списки (карманы), вначале пустые.

Алгоритм

Для реализации распределяющей сортировки для каждого шага j=1, 2, ...,T выполняется алгоритм, состоящий из двух этапов, называемых **распределение и сборка.**

Распределение заключается в том, что **элемент Ki** (i=1,…,n) из B добавляется как *последний* в список Bm, где m = D(j, Ki), и таким образом **получаются десять списков, в каждом из которых j-тые разряды чисел одинаковы и равны m.**

Сборка объединяет списки B0, B1, ... ,B9 в *том же порядке*, образуя **один массив / список B**.

4. Распределяющая сортировка

Пример

Рассмотрим реализацию распределяющей сортировки при T=2 для списка:

B={09, 07, 18, 03, 52, 04, 06, 08, 05, 13, 42, 30, 35, 26} .

J=1.

Распределение 1:

B0={30}, B1={}, B2={52, 42}, B3={03, 13}, B4={04},
B5={05, 35}, B6={06, 26}, B7={07}, B8={18, 08}, B9={09}.

Сборка 1:

B={30, 52, 42, 03, 13, 04, 05, 35, 06, 26, 07, 18, 08, 09}

J=2.

Распределение 2:

B0={03, 04, 05, 06, 07, 08, 09}, B1={13, 18}, B2={26},
B3={30, 35}, B4={42}, B5={52}, B6={}, B7={}, B8={}, B9={}.

Сборка 2:

B={03, 04, 05, 06, 07, 08, 09, 13, 18, 26, 30, 35, 42, 52}.

4. Распределяющая сортировка

Время выполнения

Количество действий, необходимых для сортировки списка из n T-значных чисел, определяется как O(n*T). Если T<<n, то **время выполнения** алгоритма распределяющей сортировки **порядка O(n).**

Недостатки

- **Сортирует только целые числа.**
- **Требует использования дополнительной памяти** под карманы.

Однако можно исходный список представить как связанный и сортировку организовать так, чтобы для карманов B0,B1,...,B9 не использовать дополнительной памяти, элементы списка не перемещать, а с помощью перестановки указателей присоединять их к тому или иному карману.

5. Битовая сортировка. Разновидность распределяющей сортировки

Элементы списка интерпретируются как двоичные числа, и D(j, N) обозначает j-ю справа *двоичную* цифру числа N, j=1,2,...,T.

В процессе **распределения** требуются только два вспомогательных кармана: B0 и B1; их можно разместить в одном массиве, двигая списки B0 и B1 навстречу друг другу и отмечая точку встречи.

Для осуществления **сборки** нужно за списком B0 написать инвертированный список B1.

Выделение j-го бита в числе требует только операций сдвига.

6. Бинарная сортировка. Разновидность блочной сортировки

Из всех элементов списка B выделяются его минимальный и максимальный элементы и находится их среднее арифметическое m =(MIN+MAX)/2.

Список B разбивается на подсписки B1 и B2, причем **в B1 попадают элементы, не большие m, а в список B2 - элементы, большие m.**

Для непустых подсписков B1 и B2 сортировка продолжается **рекурсивно, пока длина списка не достигнет 1.**