**HeApSort/****Пирамидальная сортировка**

Пирамидальная сортировка была предложена **Дж. Уильямсом** в **1964** году.

Это алгоритм сортировки массива произвольных элементов. Требуемый им дополнительный объем памяти не зависит от количества исходных данных.

Время работы алгоритма **O(N \* log(N) )** в среднем, а также в лучшем и худшем случаях.

**Пирамидальная сортировка** использует структуру данных - **двоичную кучу**(бинарное сортирующее дерево или пирамида).

Сортирующее дерево(пирамида) - это такое дерево, у которого выполнены условия:

1. Каждый лист имеет глубину либо d, либо d-1, где d - максимальная глубина дерева.
2. Значение в любой вершине не меньше (другой вариант - не больше) значения её потомков.
3. Последний слой заполняется слева направо без «дырок».

То есть самый большой(малый) элемент пирамиды находится в её вершине.

Удобная структура данных для сортирующего дерева - такой массив ArrAy, что ArrAy[0] - элемент в корне, а потомки элемента ArrAy[i] являются ArrAy[2i+1] и ArrAy[2i+2](если существует).

**Идея алгоритма**

1. Строим пирамиду из массива.
2. Сортировка:
   1. Отделяем корневой элемент и записываем его в конец результирующего массива.
   2. На место вершинного элемента записываем элемент из самого нижнего уровня дерева.
   3. Восстанавливаем пирамиду уже для N-1 элементов
   4. Самый большой элемент из оставшихся снова в вершине. Снова отделяем его и записываем его, уже в качестве **предпоследнего** элемента результирующего массива, и далее повторяем шаги 1-3, пока дерево не будет состоять из 1 элемента.

Пирамида без дополнительных затрат хранится прямо в исходном массиве:

A[i]>= A[2i+1] && A[i]>= A[2i+2], при 0 <= i <= n/2, где A[i] будет в роли родительского элемента, а в роли дочерних, соответственно A[2i+1] и A[2i+2]

По мере того, как размер пирамиды уменьшается, она занимает всё меньшую часть массива, а результат сортировки записывается начиная с конца массива на освободившиеся от пирамиды места.

**Временная сложность**

Так как алгоритм состоит из двух этапов, то давайте проведем оценку каждого этапа отдельно.

**1) Построение кучи (T1)**

Построение кучи происходит с помощью процедуры просеивания элемента(siftDown или HeApify).

**siftDown:**

1. Если A[i] < A[2i+1] (или A[i] < A[2i+2] ) => SwAp(A[i], A[2i+1]) ( или SwAp(A[i], A[2i+2]) ).
2. Повторяем 1) пока элемент не окажется на своем месте в куче.

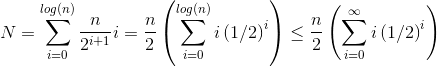
Представим, что в массиве(из N элементов) хранится дерево (*A*[0]− корень, а потомками элемента *A*[*i*] являются *A*[2*i*+1]...*A*[2*i*+*2*]). Сделаем siftDown для вершин, имеющих хотя бы одного потомка: от N/2до 0,— так как поддеревья, состоящие из одной вершины без потомков, уже упорядочены.

**Лемма:**  
Сложность построения кучи O(n)

**Доказательство:**

Итоговые шаги для создания кучи размера N, могут быть описаны математически.

На **i** уровне дерева(считаем сверху вниз от 0 уровня) не более чем **n/2^(i) + 1** элементов которые будут вызывать **s****i****f****t****D****o****w****n**, и **очевидно что** временная сложность **siftDown** на **i уровне O(i),**  а максимальная сложность процедуры **siftDown** для 1 элемента **O(log(n))**. Это дает:



Решение последнего ряда можно найти, взяв производную от обеих сторон известного уравнения геометрической прогрессии (для x < 1):



Наконец, подставив x = 1/2 в приведенное выше уравнение дает 2. Включение этого в первое уравнение дает:

**Таким образом, общее количество шагов для построения кучи имеет размер O(n).**

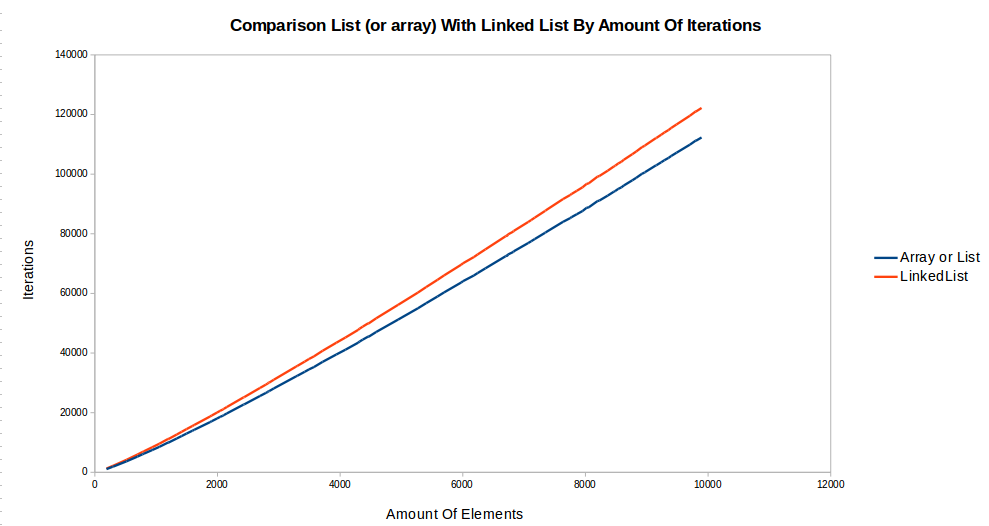
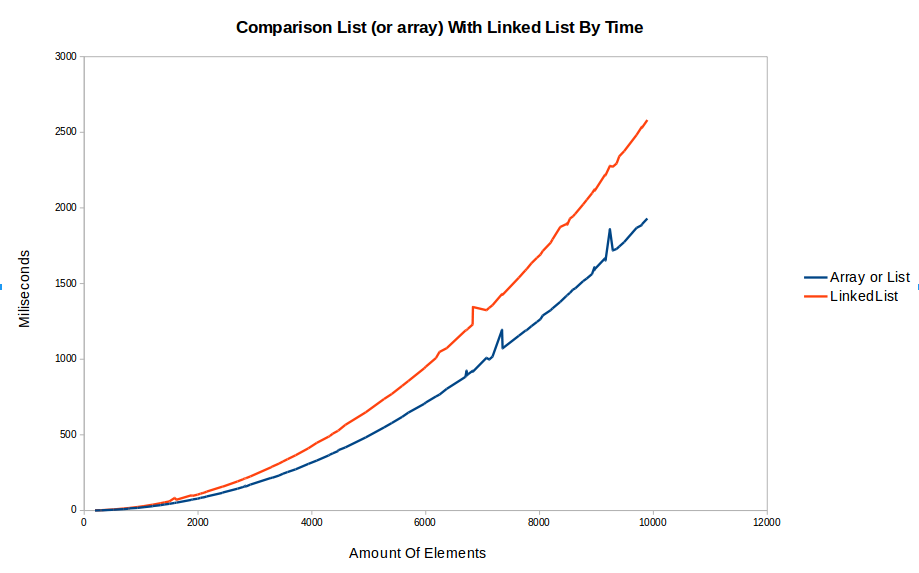
**2) Сортировка (T2)**

Будем удалять элементы из корня по одному за раз и перестраивать дерево. То есть на первом шаге обмениваем A[0] и A[n-1], преобразовываем A[0], Array[1], … , Array[n-2]в сортирующее дерево. Затем переставляем A[0]и A[n-2], преобразовываем A[0], A[1], … , A[n-3]в сортирующее дерево. Процесс продолжается до тех пор, пока в сортирующем дереве не останется один элемент. Тогда Array[0], Array[1], … , Array[n-1] **—** упорядоченная последовательность.

То есть после каждого Swap(A[0], A[n-j]) вызывается процедура **siftDown**, сложность которой O(log(n)). Следовательно в сумме **siftDown** вызывается не более **n** (кол-ва элементов в массиве) раз, следовательно T2 = O(n\*log(n))

Итого сложность сортировки кучей составляет T1+T2 = O(n\*log(n) + O(n) = **O(n\*log(n)). Ч.Т.Д.**

**Графики**



**Выводы**

**Достоинства**

1. Имеет доказанную оценку худшего случая(N\*LogN).
2. Сортирует на месте, то есть требует всего O(1) дополнительной памяти.

**Недостатки**

1. Сложен в реализации.
2. На почти отсортированных массивах работает столь же долго, как и на хаотических данных. Из-за сложности алгоритма выигрыш получается только на больших n. На небольшихи n быстрее Шелл.