1.Ω (омега большое) - нижняя граница оценки

Θ (Тета) - точная оценка

O (О большое) – верхняя граница оценки

2. Выбор – О(n^2)

Вставки – О(n) в лучшем, О (n^2) – в среднем

Пузырек – О(n) в лучшем, О (n^2) – в среднем

3. Merge – n log n

Quick - худшее – n^2, среднее n log n

4. В инверсии счетчик каждая инверсия +1

5. Сортируем массив и берем к элемент О(n log n) , быстрее – O(n)

6. Делим на половины и до победы О(log n)

7. Куча - Удобнее всего двоичную кучу хранить в виде массива a[0..n−1]a[0..n−1], у которого нулевой элемент, a[0]a[0] — элемент в корне, а потомками элемента a[i] являются a[2i+1] и a[2i+1].Высота кучи определяется как высота двоичного дерева. То есть она равна количеству рёбер в самом длинном простом пути, соединяющем корень кучи с одним из её листьев. Высота кучи есть O(logn)O(log⁡n), где n — количество узлов дерева.

Все siftup и siftdown за O(log n)

Нахождение мин – это корень дерева

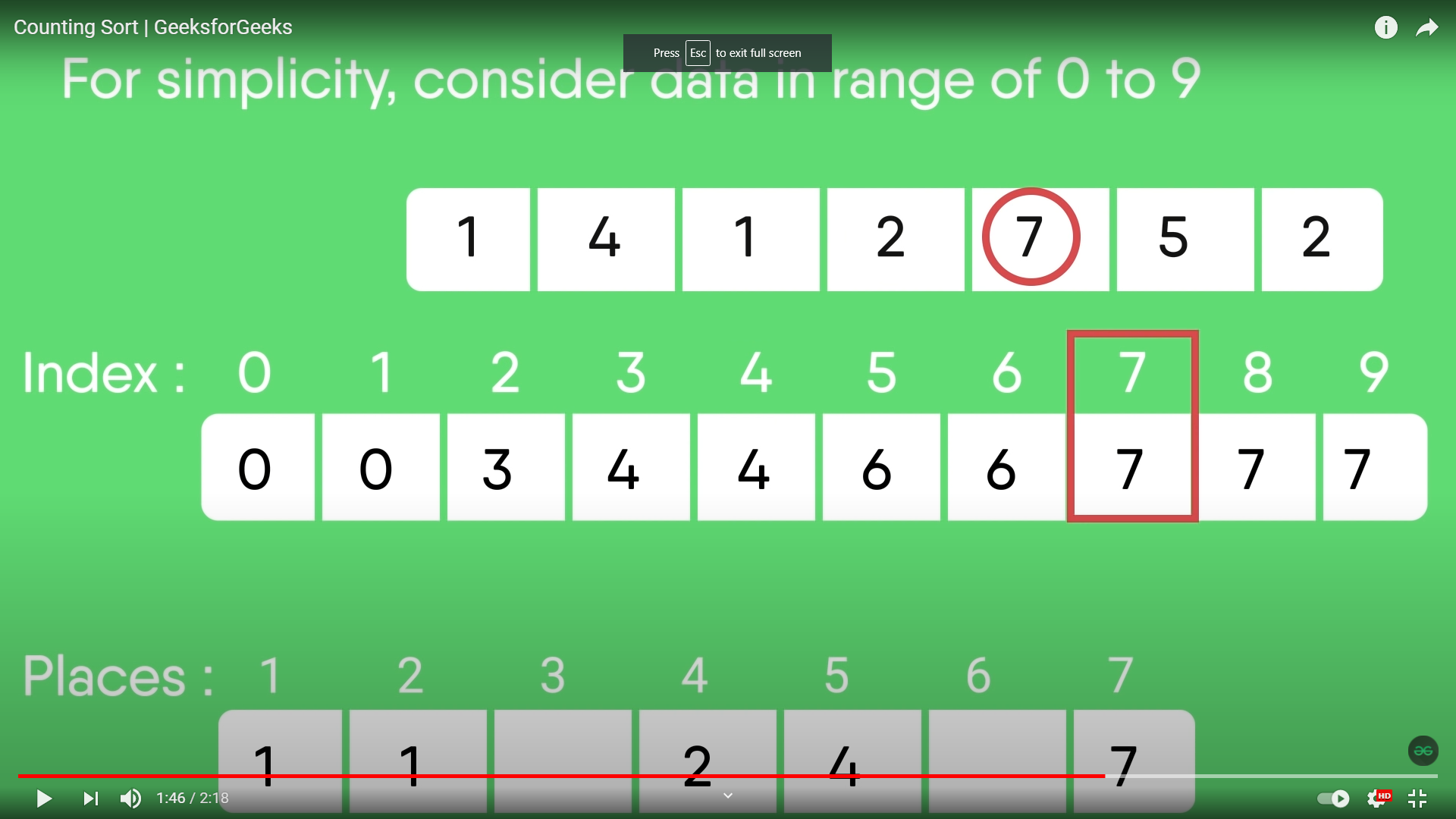
1. Извлечение мин элемента – за O(log n) Значение корневого элемента (он и является минимальным) сохраняется для последующего возврата.
2. Последний элемент копируется в корень, после чего удаляется из кучи.
3. Вызывается siftDown для корня.
4. Сохранённый элемент возвращается.

Выполняет добавление элемента в кучу за время O(logn). Добавление произвольного элемента в конец кучи, и восстановление свойства упорядоченности с помощью процедуры siftUp.

Изменение – свап корня и последнего элемента, удаляем корень и сифт ап

8. Хип сорт O(n log n) - Строим из массива кучу, по очереди извлекаем минимум кучи.  
Заметим, что можно отсортировать массив, сначала построив из него двоичную кучу, а потом последовательно извлекая максимальные элементы.

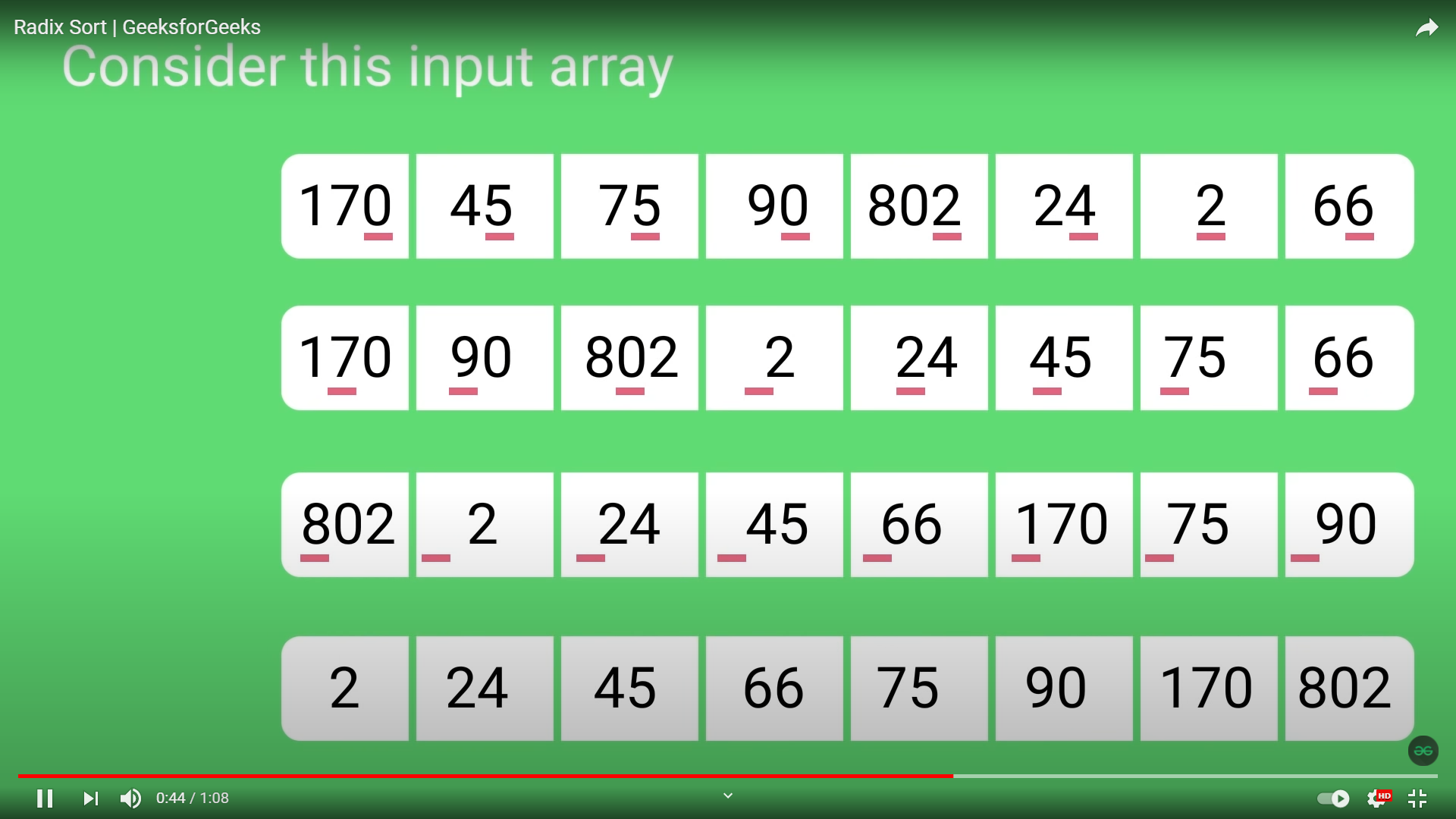
9. Подсчет Лучший – o(n), mid – o(n+k) , min o(k);

Делаем массив – счетчик, считаем по индексу сколько раз у нас есть число 

Потом прибавляем все ячейки индексов друг с другом

Потом по этому массиву индексов выставляем числа, выставив число, уменьшаем счетчик в индекс массиве на 1

10. цифровая сортировка – n log n



Сортируем по разрядам с единиц до сотен

11. Стэк – LIFO(LAST IN, FIRST OUT)(пирамидка) реализация через массив, индекс посл элем, STACK\_EMPTY, PUSH, POP О(1)

Очередь – FIFO (FIRST IN, FIRST OUT) реализация через массив, индекс 1 и посл элем. ENQUEUE, DEQUEUE О(1)

12. Хеш – закрытая адресация (метод цепочек) – Теперь из каждой ячейки таблицы ведет указатель на голову двусвязного списка из ключей, чьи значения хеш функции совпали.

Insert – вставка х в заголовок списка T[h.(x.key)] O(1)

Search – поиск элемента с ключом k в списке T[h(k)] O(1)

Delete – удаление x из списка T[h(x.key)] O(1)

Худший случай – все элементы оказались в одной ячейке тогда O(n)

Хеш – открытая адресация -все элементы хранятся в самой таблице, отказ от указателей, элементы с одинаковым хешем хранятся в виде последовательностей с определенным шагом,хеш стартовой ячейки вычисляется в зависимости от ключа, если эта заполненна, то проверяем след с опр шагом, шаг может быть const, а может и не быть

Insert – идут по ячейкам исследования вплоть до попадания в пустую ячейкуО(1)

Search – идут по ячейкам исследования до ячейки с иск значеним О(1)

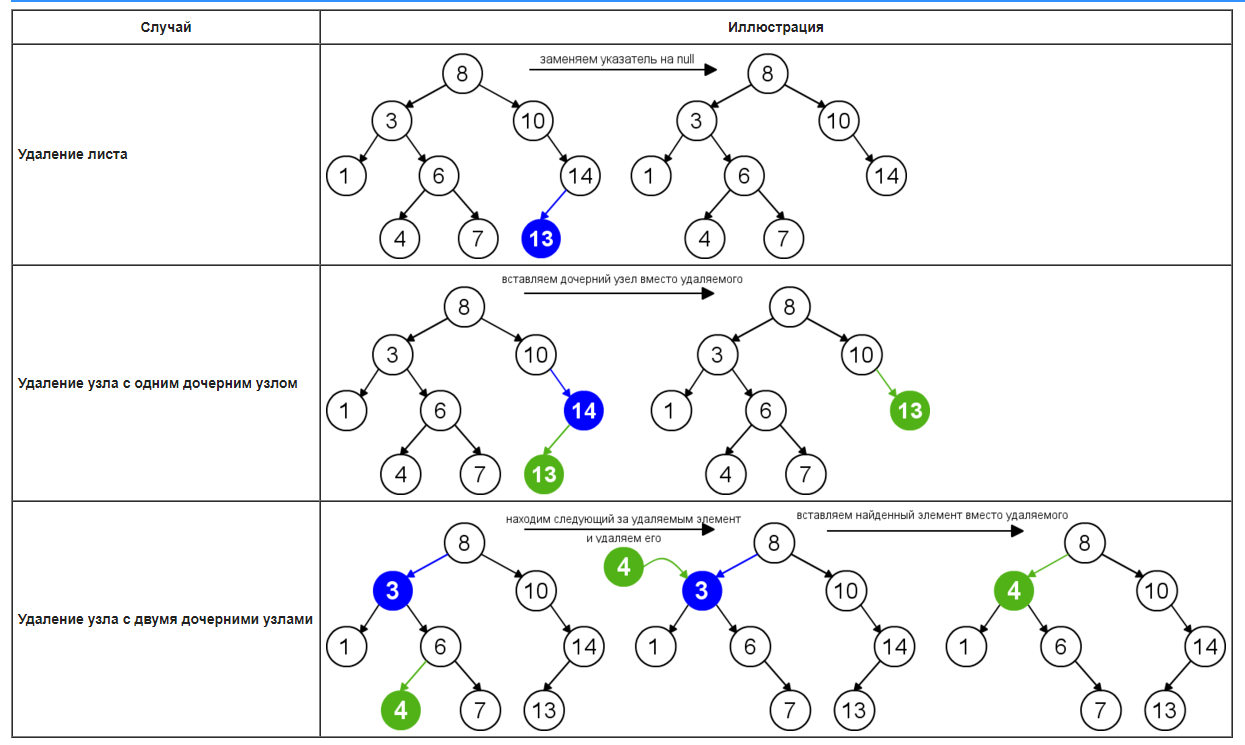
Delete – помечаем удаляему ячейку как deleted, тогда инсерт видит ее как пустую, а серч как заполненную О(1)

13. Бинарное дерево поиска обладает следующим свойством: если x — узел бинарного дерева с ключом k, то все узлы в левом поддереве должны иметь ключи, меньшие k, а в правом поддереве большие k.

Для поиска элемента в бинарном дереве поиска можно воспользоваться следующей функцией, которая принимает в качестве параметров корень дерева и искомый ключ. Для каждого узла функция сравнивает значение его ключа с искомым ключом. Если ключи одинаковы, то функция возвращает текущий узел, в противном случае функция вызывается рекурсивно для левого или правого поддерева. Узлы, которые посещает функция образуют нисходящий путь от корня, так что время ее работы O(h) где h — высота дерева.

Операция вставки работает аналогично поиску элемента, только при обнаружении у элемента отсутствия ребенка нужно подвесить на него вставляемый элемент.

Для удаления узла из бинарного дерева поиска нужно рассмотреть три возможные ситуации. Если у узла нет дочерних узлов, то у его родителя нужно просто заменить указатель на null. Если у узла есть только один дочерний узел, то нужно создать новую связь между родителем удаляемого узла и его дочерним узлом. Наконец, если у узла два дочерних узла, то нужно найти следующий за ним элемент (у этого элемента не будет левого потомка), его правого потомка подвесить на место найденного элемента, а удаляемый узел заменить найденным узлом. Таким образом, свойство бинарного дерева поиска не будет нарушено. Данная реализация удаления не увеличивает высоту дерева. Время работы алгоритма — O(h)



14. АВЛ - АВЛ-дерево — это прежде всего двоичное дерево поиска, ключи которого удовлетворяют стандартному свойству: ключ любого узла дерева не меньше любого ключа в левом поддереве данного узла и не больше любого ключа в правом поддереве этого узла. Особенностью АВЛ-дерева является то, что оно является сбалансированным в следующем смысле: для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу.

Поиск как в бин дереве О(h)

Вставка нового ключа в АВЛ-дерево выполняется, по большому счету, так же, как это делается в простых деревьях поиска: спускаемся вниз по дереву, выбирая правое или левое направление движения в зависимости от результата сравнения ключа в текущем узле и вставляемого ключа. Единственное отличие заключается в том, что при возвращении из рекурсии (т.е. после того, как ключ вставлен либо в правое, либо в левое поддерево, и это дерево сбалансировано) выполняется балансировка текущего узла. Строго доказывается, что возникающий при такой вставке дисбаланс в любом узле по пути движения не превышает двух, а значит применение вышеописанной функции балансировки является корректным. O(logn)

Удаление Для простоты опишем рекурсивный алгоритм удаления. Если вершина — лист, то [удалим](http://neerc.ifmo.ru/wiki/index.php?title=%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0,_%D0%BD%D0%B0%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D1%80%D0%B5%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F#.D0.A3.D0.B4.D0.B0.D0.BB.D0.B5.D0.BD.D0.B8.D0.B5) её, иначе найдём самую близкую по значению вершину aa, переместим её на место удаляемой вершины и [удалим](http://neerc.ifmo.ru/wiki/index.php?title=%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0,_%D0%BD%D0%B0%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D1%80%D0%B5%D0%B0%D0%BB%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F#.D0.A3.D0.B4.D0.B0.D0.BB.D0.B5.D0.BD.D0.B8.D0.B5) вершину aa. От удалённой вершины будем подниматься вверх к корню и пересчитывать баланс у вершин. Если мы поднялись в вершину ii из левого поддерева, то diff[i] уменьшается на единицу, если из правого, то увеличивается на единицу. Если пришли в вершину и её баланс стал равным 1 или −1, то это значит, что высота этого поддерева не изменилась и подъём можно остановить. Если баланс вершины стал равным нулю, то высота поддерева уменьшилась и подъём нужно продолжить. Если баланс стал равным 2 или −2, следует выполнить одно из четырёх вращений и, если после вращений баланс вершины стал равным нулю, то подъём продолжается, иначе останавливается.

В результате указанных действий на удаление вершины и балансировку суммарно тратится, как и ранее, O(h) операций. Таким образом, требуемое количество действий — O(logn).