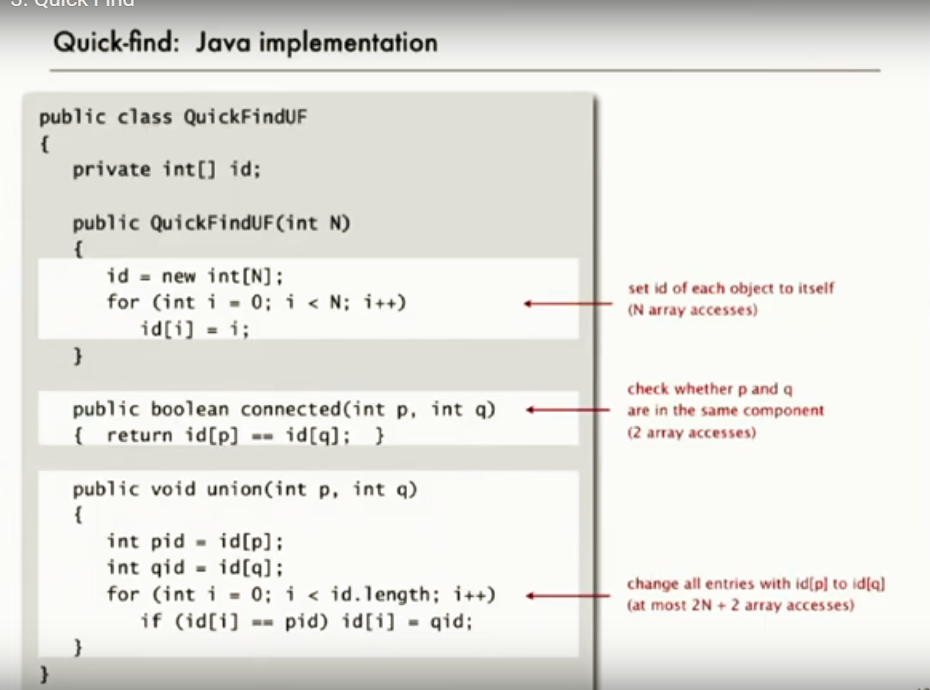
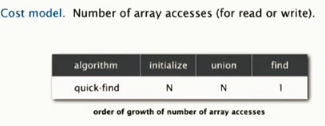
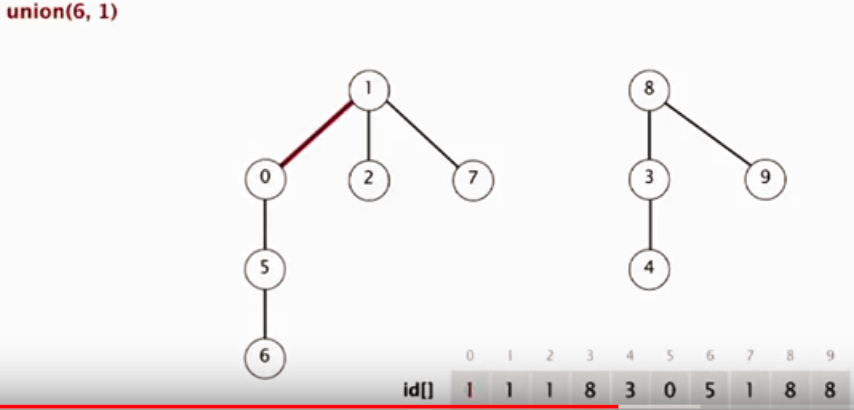
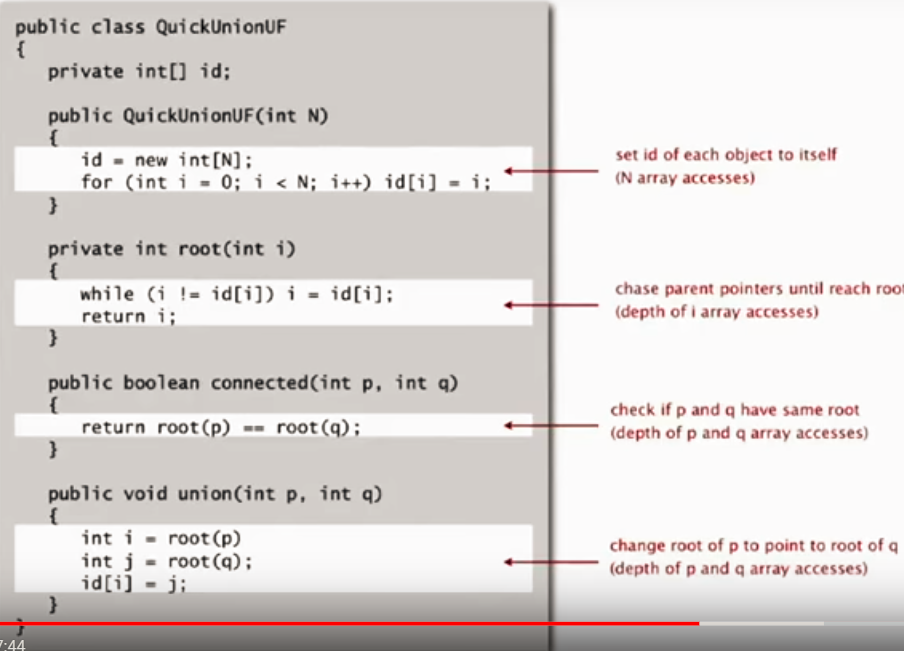
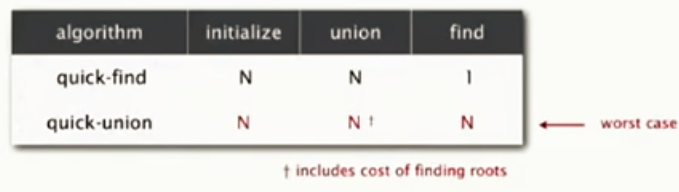
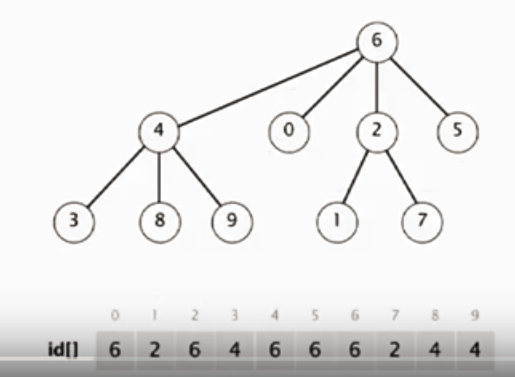
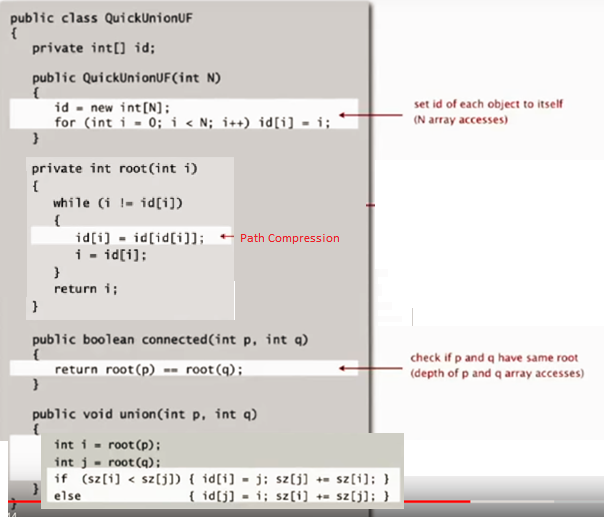
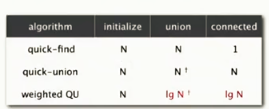
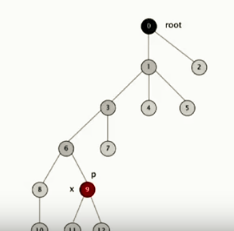
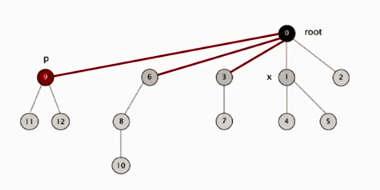
# **DYNAMIC CONNECTIVITY**

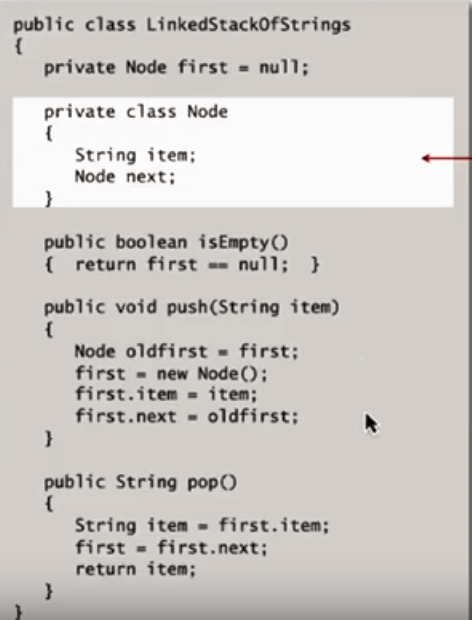
* **Quick Find**
* 
  + - *Do both indices have the same value? If so, connected.*
* 

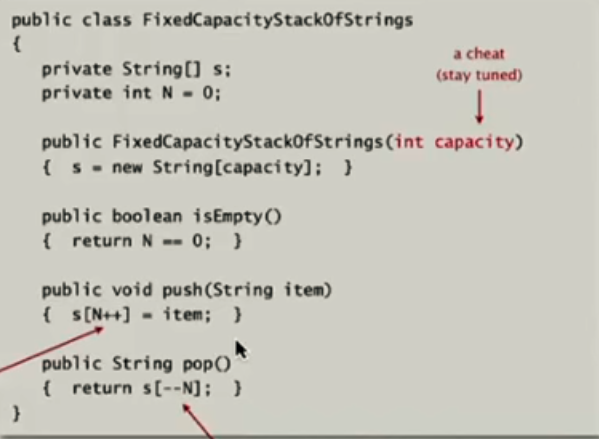
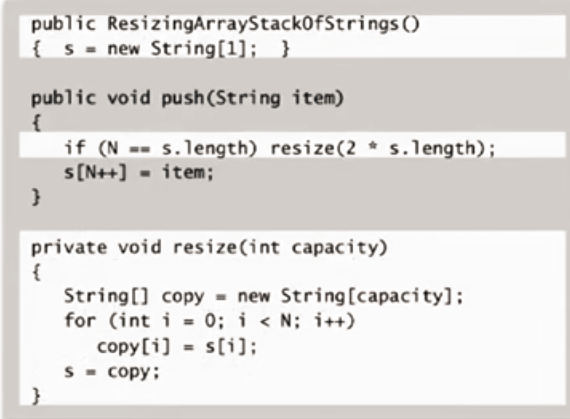
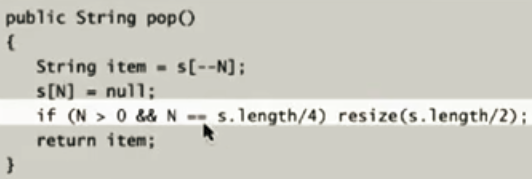
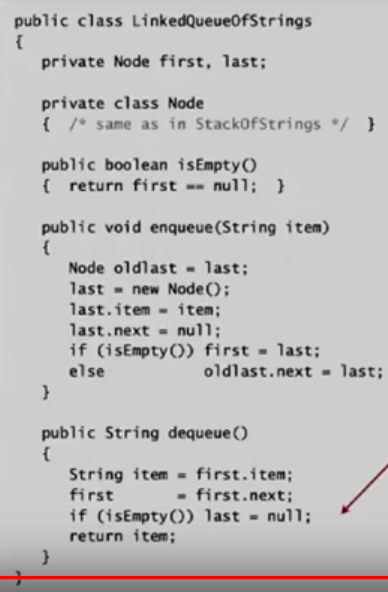


* + **Problems: union is N time. for N unions on N items: quadratic time.**
* **Quick Union**
  + When you follow roots, do both end up going to the same root? If so, connected
  + 
  + 
  + **Good:** Tree structure prevents N unions on N items getting to quadratic?
  + **Problems:** Trees get too tall. Find now takes N array accesses.
    - 
* **Weighted Quick Union (w/Path Compression)**
  + When combining trees – put the smaller one below the bigger one (creates shorter trees). When attaching: roots to roots.
    - 
    - 
  + 
  + 
* **Path Compression**

While you’re moving up paths to find root, add to root (every other root gets added to root)**** near goes to ****

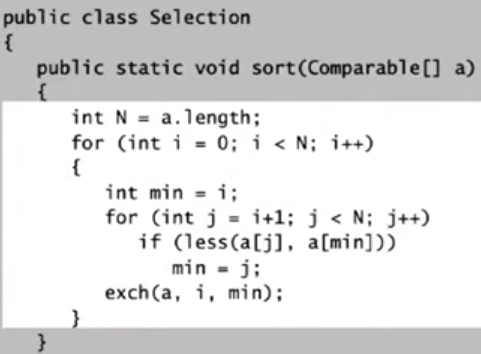
# **STACKS & QUEUES**

* **Stack**
  + **Linked List Stack**
    - 
    - N items = 40N bytes

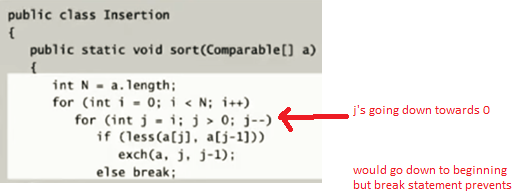
* + **Resizing Array Stack**
    - 
      * Resizing
      * 
      * 
  + Linked List vs. Array
    - LL is always constant time. More time/space due to links. Slower.
    - Array less wasted time/space. Faster.
      * Bad for sudden high-traffic tasks (slow-downs for resizing).
* **Queue**
  + **Linked List Queue**
    - ****
  + **Array Queue**
    - Use stack code, with a first and last pointer

# **SORTS**

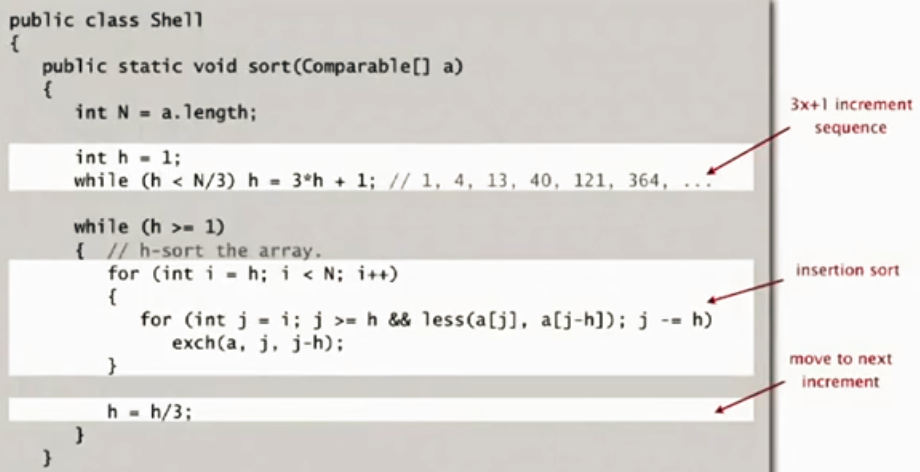
* **Selection Sort**
  + *select the best one – move it to the front. advance/repeat*

****

* + ***Performance***
    - *ALWAYS*
      * quadratic compares (1/2 n-squared) (even if list is already sorted)
      * linear exchanges (even if sorted)
* **Insertion Sort**
  + *move right. if an item is bigger/smaller than the ones before it, start moving it back until it is in place*

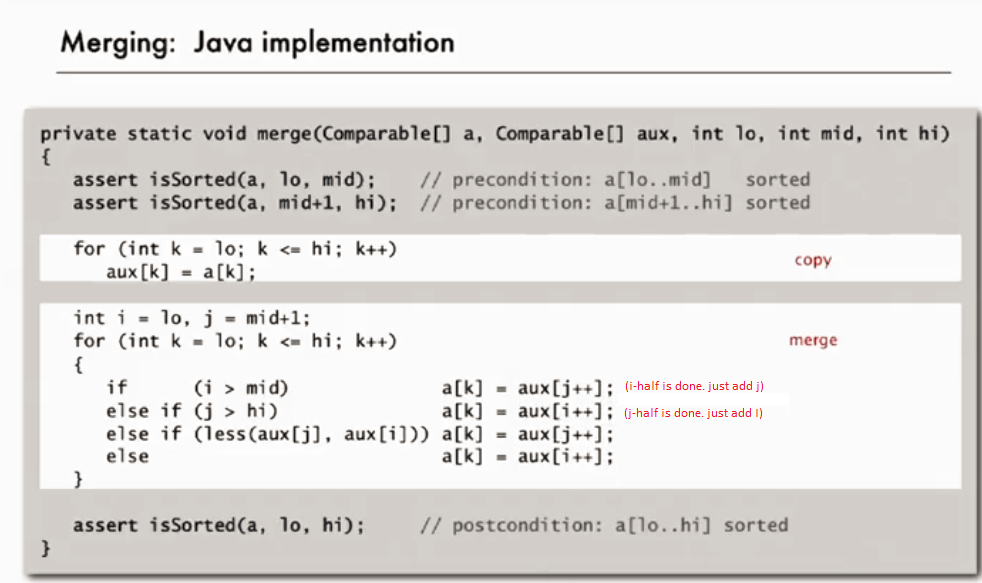


* + ***Performance***
    - **random-ordered array:**
      * Compares: quadratic (1/4 n-squared)
      * Exchanges: quadratic (1/4 n-squared)
    - **if sorted:**
      * Compares: linear (n-1) (just validating, essentially)
      * Exchanges: zero
    - **if partially-sorted:**
      * Compares: linear
      * Exchanges: linear
    - **if reverse-sorted** *(worse than Selection-Sort)*
      * Compares: quadratic (1/2 n-squared)
      * Exchanges: quadratic (1/2 n-squared)
* **Shell Sort**
  + *Insertion sorts with longer and then shorter strides (13, 5, 3, 1 or something)*
    - *rmbr: with insertion sort, you pull something ALL the way back when you can.*

******

* + ***Performance***
    - No known worst-case, but usually operates ~~n log n when using 3x+1
    - Fast unless array is huge
    - Little code – good for embedded
* **Merge Sort**

***MERGE***

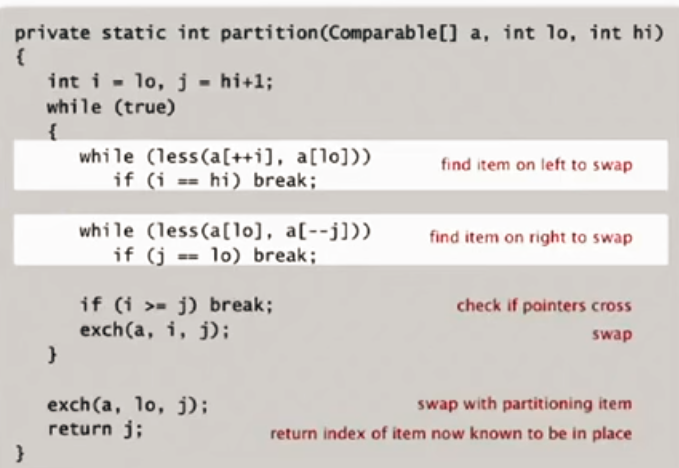
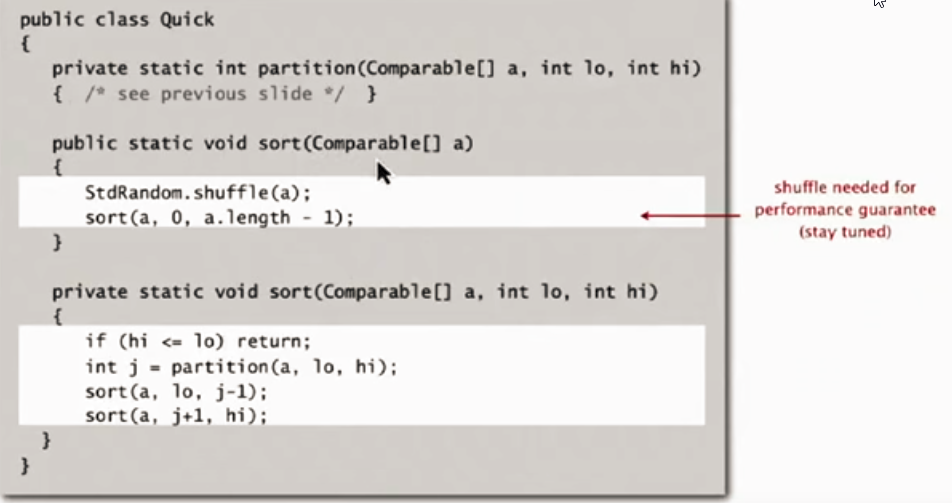
******

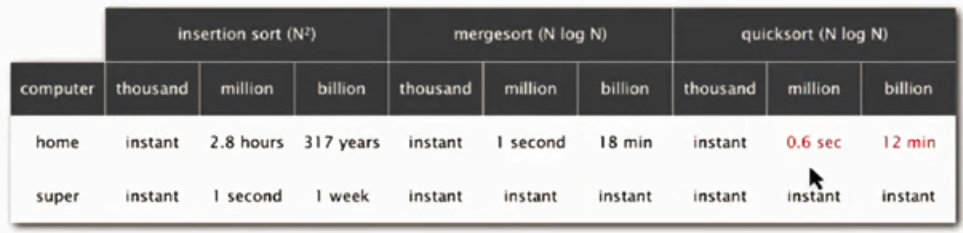
***mergeSORT/bottomUP***

**Need Pic**

* + ***Performance***
    - *n log n (random or reverse sorted)*
  + ***Practical Improvements***
    - *switch to Insertion sorts when subarray is less than 7 elements (lowers recursion overhead)*
    - *add test to see if array already sorted (becomes linear)*
    - *Consider non-recursive bottomUp mergeSort*
* **Quick Sort**
  + in-place sort
  + recursion after sort vs. mergeSort’s recursion before sort
  + *shuffle the array*
  + *two pointers [ i , j ] and the first item in array (k)*
  + *i-pointer on left. stop incrementing i-pointer when it is larger than k-val.*
  + *j-pointer on right. stop decrementing j-pointer when it is smaller than k-val.*
  + *exchange i-j values*

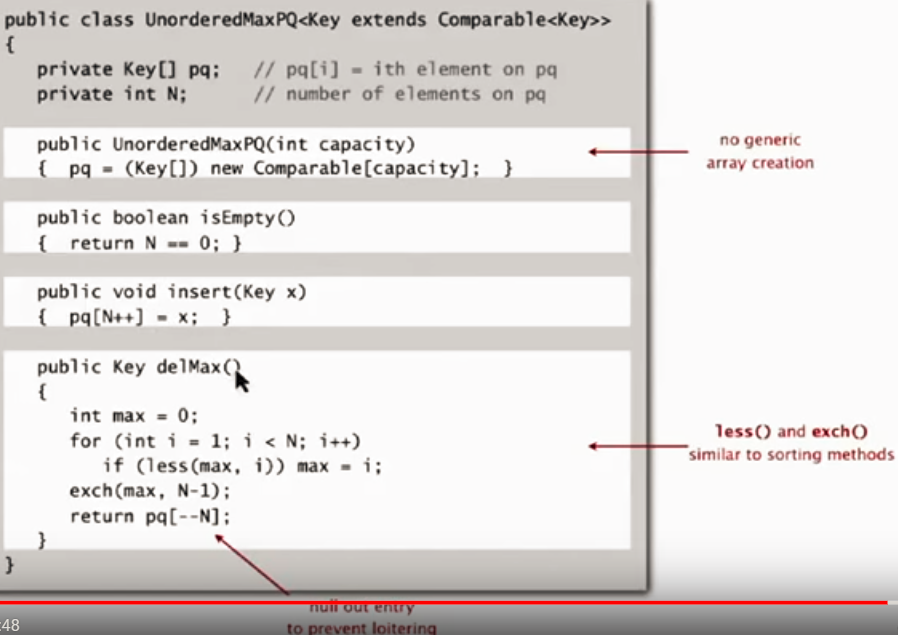


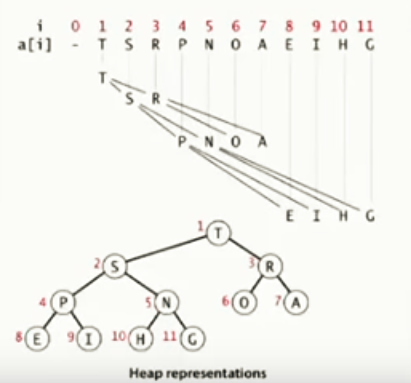
* + *continue this process until pointers cross. exchange k with j.*
  + ******
  + ******
  + ***Fastest sort in practical situations!***
  + can be n-squared in worst case. but shuffled – n log n
  + cutoff to insertion sort for ~10 item subarrays

****

# PRIORITY QUEUE AND HEAPS

* + **Unordered Array** (linear search for maximum)
    - Use stack code, with a first and last pointer

ssss

* **Binary Heap**
  + **Structure**
    - Actually an array (use math to get around)
    - Children must be smaller
      * no lateral ordering
      * certain items in a higher level could be smaller than those in a lower level. just need to respect parent-rule.
    - 
    - If child becomes larger than parent:
      * bubble the bigger child up
    - To delete max in heap:
      * exchange with a bottom node, delete. Bubble to replace top properly.
    - Insertion: bottom of heap, then swim up
    - 
  + **HeapSort**

1. Get the array Heap-Ordered.
   1. From bottom to top, going right to left: if elements are unordered, use swim/sink
2. Sort the heap
   1. Repeatedly exchange max at end with a bottom leaf (at end of array). Maintain heap order with sink/swim.
   2. All maxes eventually moved to the end of the array.
   * ***Performance***
     + *n log n*
     + *In place (n space)*
   * ***Practical Considerations***
     + *Not used very often (poor use of cache memory)*
     + *In theory – optimal for both time and space, but:*
       - *inner loop longer than quick sort’s*
       - *poor use of cache memory*
       - *not stable (long distance exchanges)*
         * *stability*

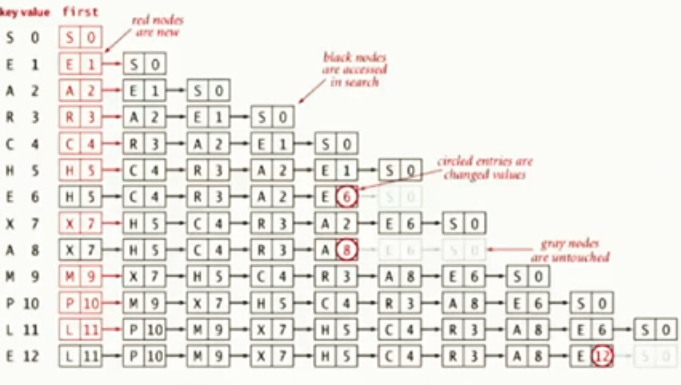
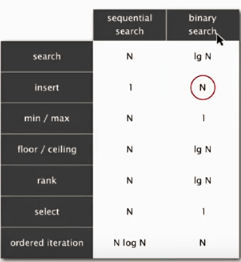
*relative order of near-identical items is retained*

*(1,4) (2,5) (6,7) (2, 3) -> (1,4) (2,5) (2,3) (6,7)*

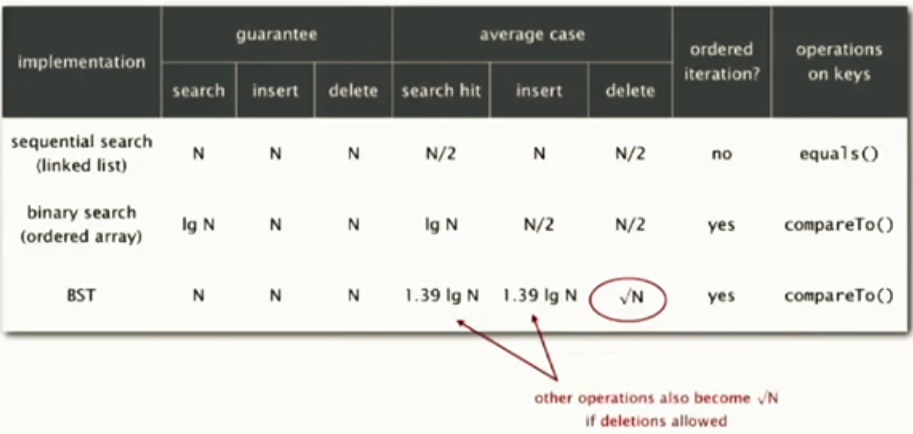
*red stays before yellow*

*stable sorts have elements move 1 element at a time back/forth. non-stable will have jumps that can ruin this.*

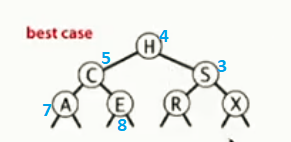
# **SYMBOL TABLE**

* + like a python dictionary (without hashing, yet)
  + **Linked List**
    - Search if value exists in LL. If so, update associated number value. If not, add key/val to LL.
      * **Problem:** Search and insert both linear (need to search full list).
      * ****
  + **Ordered Array**
    - ****
    - **Problem:** To maintain order, need to shift all the greater keys over
      * *(good for symbol tables that won’t be changed much)*
    - ****

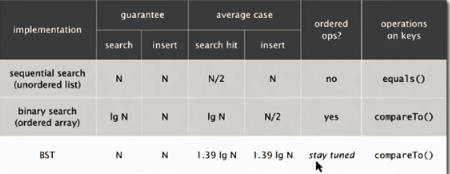
**Symbol Tables:**

****

* **Binary Search Tree**
  + - **Explanation**
      * Linked list with node, value AND left, right values
        + value field supports symbol table ops. looks like this?

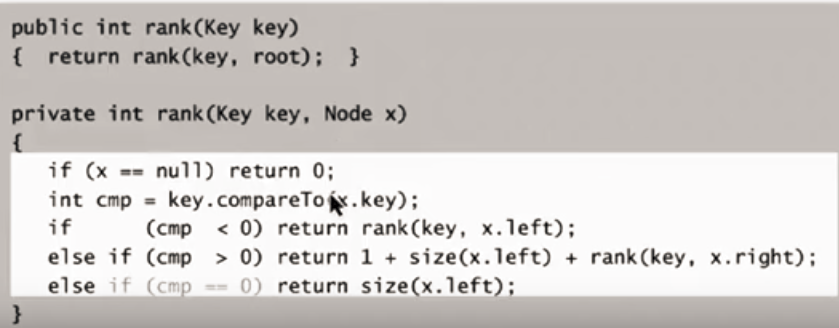
****

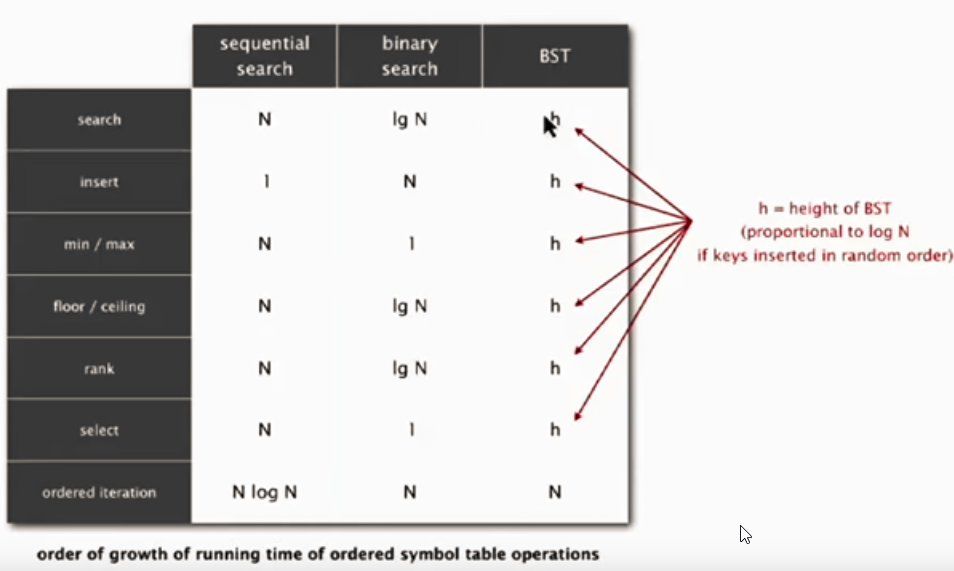
* + - * + move through *keys* in ordered BST fashion. then return value.
      * Left child smaller. right child bigger. also creates lateral order.
      * Search/Insert: log2

****

* + - * **Problems**
        + Tree order based on order in which elements inserted.

Worst Case: if in increasing order, it’s just a linked-list.

* + - **Ordered Operations**
      * Min / Max
        + left-most and right-most elements
      * Floor (biggest element less than a number) / Ceiling (smallest element bigger than a number)
        + think of floor and ceiling closing in
        + ****
      * Rank - Subtree Counts
        + add extra val (self.count) to Node-class: for how many nodes children to it
        + ****

****

* + - **Deletion**
      * Hibbard deletion:
        + **Node with 0 children:**

Easy – just remove reference

* + - * + **Node with 1 child:**

attach its parent to the node’s child

(effectively cut out the node from the list).

* + - * + **Node with two children:**

in deleted-node’s right child-tree: find minimum

minimum will have only 1 child at most

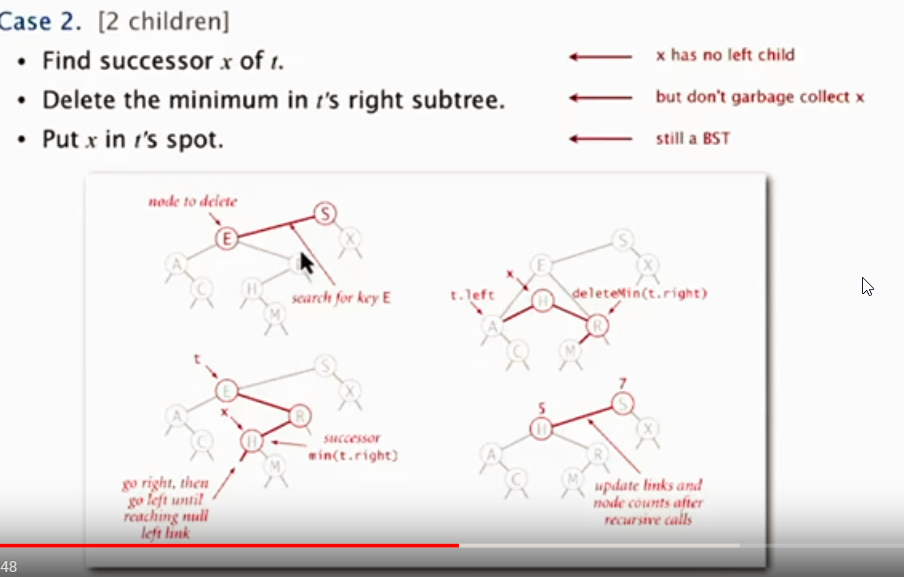
move small-node to del-node’s place.

attach small-node’s 1 child into place under right-child (essentially delete the small node - uses the 1 child method above)

* + - * + **Problem:**

Always taking right successor leads to unbalance.

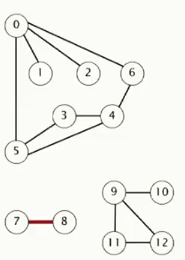
Height become sqrt(n) runtime.



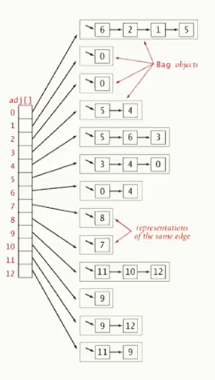
* + **Non-Binary Tree Structures (Red/Black, 2-3, etc.)**
    - red/black provides log(n) performance guarantees
* **Hashing**
  + *built in hash functions for standard types.*
  + *user-defined types may have issues.*
    - use all member variables in your custom type into hash function.
  + **use hashing when:**
    - ordered ops not needed
    - short keys (hash fx easy to compute)
  + Collision Resolution
    - Separate Chaining
      * Hash key array points to linked list.
      * Standard implementation: if you have N keys into M buckets: M = N/5
        + 100 keys to 20 buckets
    - Linear Probing
      * array size must be bigger than list of keys (ideally: keep the array half full!)
      * placement
        + put key where it hashes (position x).
        + if that’s taken, try x+1, x+2, etc. (wrap around if needed)
      * searches uses same method (where it hashes, then move)
        + if search value isn’t in array – you’ll probably hit an empty (null) spot while iterating.
        + ****

# **GRAPHS**

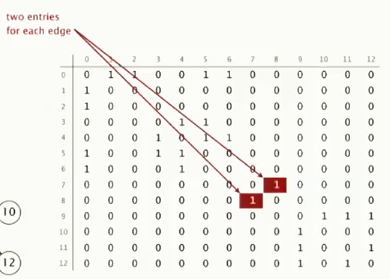
* **Undirected Graph**
  + - **Definitions:**
    - **Graph:** set of vertices and edges
    - **Path:** sequence of vertices connected by edges
    - **Cycle:** path whose first and last vertices are the same
      * **MST:** shortest set of edges that connect all vertices
  + **API**
    - addEdge(v, w)
    - findAdjacent(v)
  + **Representation**

****

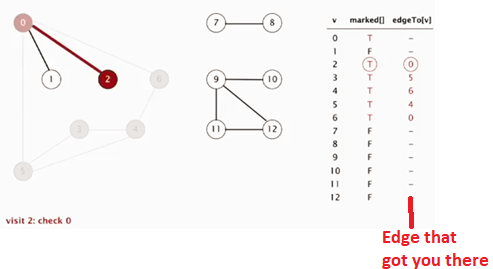
* + - **Adjacency List**

****

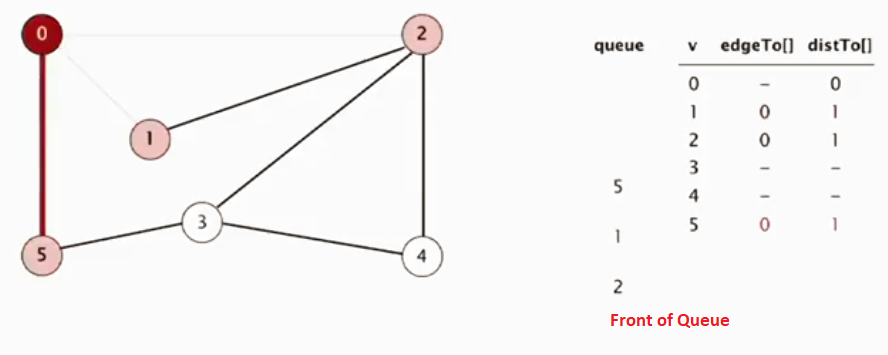
* vertex-indexed array, with linked list (or bag) for all connected vertices
  + vertex access constant
  + connection access proportional to degree of graphs (for sparse graphs, this is small)
    - **Adjacency Matrix**

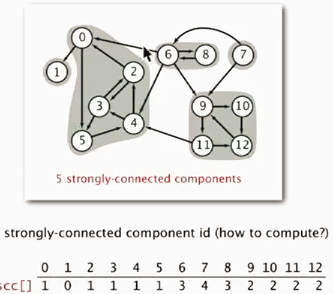
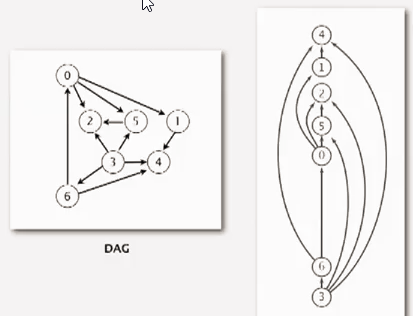


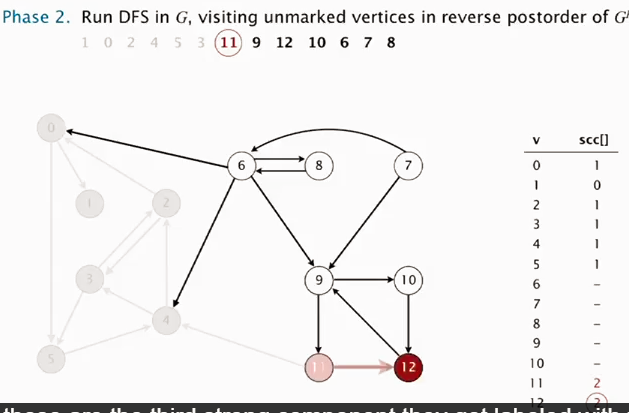
* inefficient for large, sparse graphs
  + **Processing**
    - **Depth First Search**
      * *recursive-stack based*
      * mark vertex-v as visited
      * recursively visit all unmarked vertices adjacent to v

******

* + - **Breadth First Search**
      * *queue based*
        + *assess* ***all*** *vertices connected to starting vertex before pursuing further down path*
      * **provides shortest path**
      * add starting vertex to queue
      * remove from queue
      * add all unmarked *vertices* adjacent to queue and mark them

******

* + - **Connected Components**
      * Use DFS to find all connected components
        + mark visited/which source node
      * go to next unmarked vertex. repeat DFS process.
      * ******
* **DiGraph**
  + **Representation**
    - Adjacency list (each will list all vertices that vertex v points OUT to)
      * if you need all vertexes that point IN to vertex v, add a second adjacency list to each vertex
  + **Processing**
    - Search - DFS
    - Shortest Path – BFS
  + **Topological Sort**
    - points all nodes upwards to create a process line
    - must be acylic (don’t want cyclic pre-requisites)
    - Use DFS
      * Pick a vertex and recurse.
      * When you reach an endpoint – whenever you need to go back from recursion, add the current element to post-order stack.
      * If you’ve reached all vertices you can and recursed back and there’s still vertices in graph – pick a random one and continue process.
      * Pop off stack til empty to get top-sort order
        + 
        + 
  + **Strong Components**
    - def: a path exists from v to w AND w to v
    - Data Structure
      * use same structure as connected components for UndiGraphs – if they have the same array value, they’re connected
    - process
      * reverse graph and computer top-sort order
      * run dfs on original graph, using vertices from Step 1’s top-sort order
        + for all vertices you can reach as you do this, they’re in the same strong CC. mark via array.



* **Minimum Spanning Trees**
  + - **Representation**
    - Edge objects (int v, int w, int weight)
    - EdgeWeightedGraph objects (array of Edge objects that contain specified vertex)
      * iterable: edges adjacent to a vertex
    - MST object (EdgeWeightedGraph G)
      * iterable: edges in MST
    - **MST:** If edges have weights: acyclic path that connects all vertices with minimum weight
    - **Greedy Algorithm**
    - take a random selection of nodes to be your cut (can’t have marked crossing edges)
    - mark the minimum cross black
    - repeat until you have V-1 edges mark black
  + **Kruskal’s Algorithm**
    - sort edges by weight
    - from minimum weight, keep adding edges unless they create a cycle
      * to finish: check if you have v-1 edges or keep cycle checks
      * to find cycles: use union-find data structure (log\* V runtime)
        + maintain a set for each connected component in graph
        + if v and w (vertices you’re considering connecting in MST) already in same set, you’d get a cycle
    - for list of edges E, computes MST in E log E time
  + **Prim’s Algorithm**
    - Start with a vertex
    - add the min weight edge that extends the tree
      * but only one end point in the tree – prevents cycles
    - To find min weight:
      * Use Priority Queue
        + Lazy: E log E time, E extra space
    - **Practical Considerations**
      * Array for dense graphs
      * binary heap for sparse graphs
      * Performance critical? use 4-way heap
* **Shortest Paths**
  + - **\_\_\_\_\_**
    - Used on Weighted Digraphs
    - DirectedEdge Object -> DiGraph Object -> ShortestPath Object
    - Use 2 parallel arrays (edgeTo and distTo)
      * *similar* to DFS – recurse from end point until you get to where you start

