

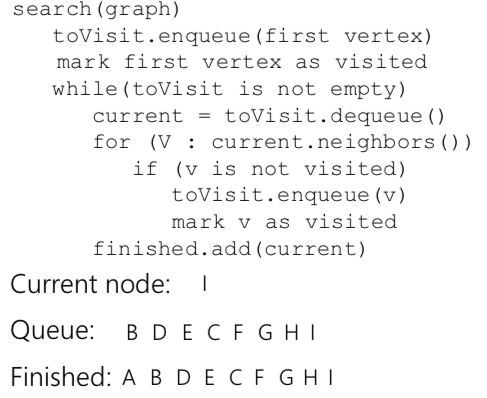
**Reduction: changing problem around so it fits an algorithm you already have!**

**BFS – Graph traversal** O(V+E)

Closer nodes from start reached first! 🡪 QUEUE

Shortest path for unweighted graphs!

Add v.dist = curr.dist + 1; v.pred = curr

  
**DFS – Graph traversal** O(V+E)

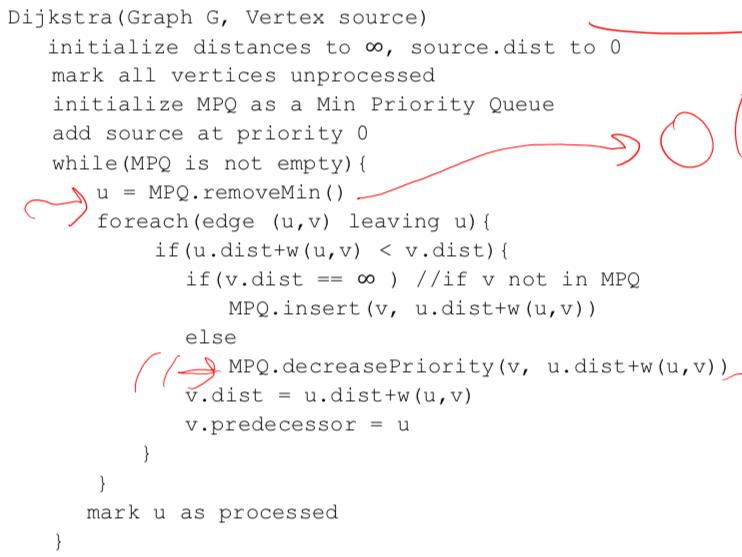
Searches farther nodes first STACK

But same as BFS besides order

Dijstrkas –Shortest Path O(log V (V + E))

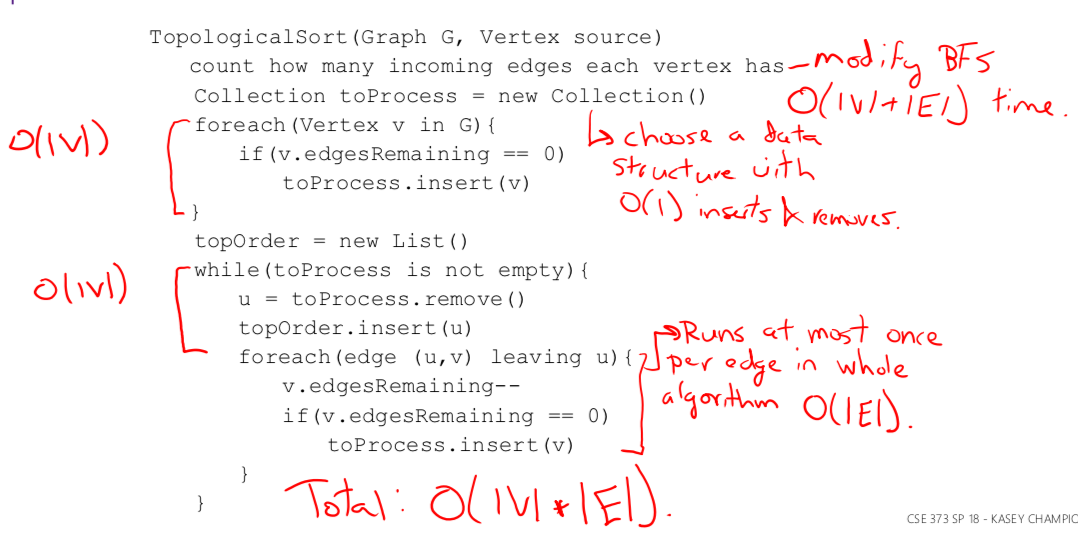
minPriorityQueue needs UPDATE priority feature

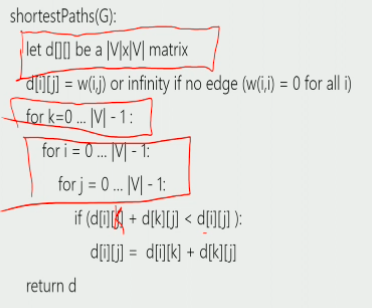
While loop/for each are pretty much graph traversal (O(V+E)) and we call a remove or change priority each team, a log V operation



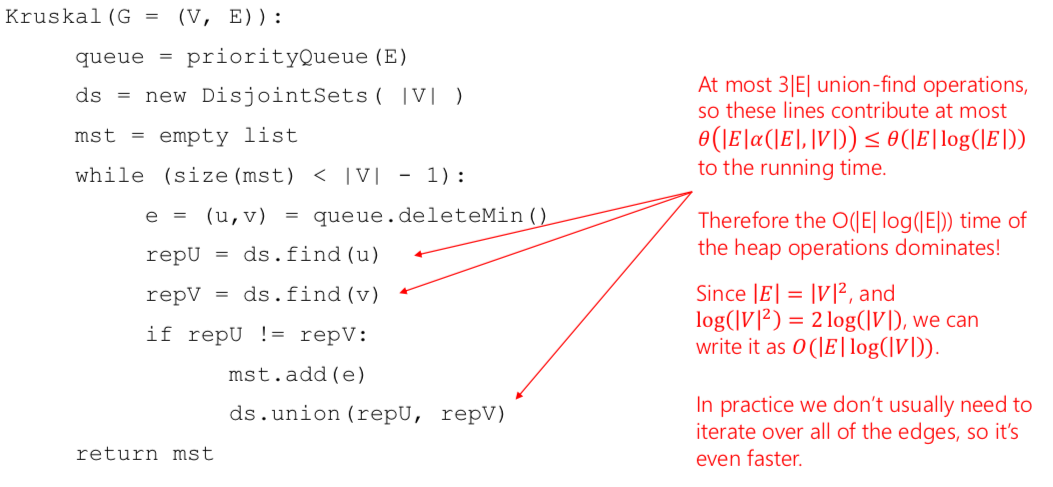
Topological sort O(V + E)

**DAG** 🡪 sort exists





Kruskals –MST O(E log V)

Sort edges; continuously choose smallest edge unless it creates a cycle until MST is formed

Graph Definitions/terminology

**Simple graph:** no parallel edges/self loops

**Dense:** edges ~ V2 ; **Sparse:** edges ~ V

**Adjacency matrix:** (i,j) = weight of eij or -1 if N/A; add/remove/check edge exists = O(1); get out/in (vi) = O(V); space inefficient = O(V2)

**Adjacency List:**  vertex 🡪 Edges/adj vertexes; add = O(1); remove edge/check exists; get out vi = O(min(V,E)); get in = O(V+E); space = O(V+E)

**Path:** seq of connecteted vertices (simple = no repeated vertex except beginning maybe)

**Cycle:** paththat starts and ends at same vertex

**Connected Graph:** path between any two vertices (defines reachability)

If directed: **strongly:** path between any two vertices **weakly:** connected if considered as undirected

**Tree:** Acyclic; one bath between each pair of vertices (n-1 edges)

**Spanning =** includes all vertices; **min = smallest edge sum**

**Subgraph:** delete edges and vertices (and remaining is valid graph)

**Clique:** complete subgraph

**Connected component:** maximalconnected subgraph (weak or strong)

**Maximal “”:** cannot add another vertex and maintain property “”

**Maximum “”:**  largest “” in graph overall (also maximal by def)

Memoization: store results of a deterministic function in a table for later use

Deterministic function: give same arguments returns same results; not merge sort as lists you give it to merge each time may be different

Trade off between space and time

Dynamic programing: define subproblems; what subproblems does OPT(x) rely on?

Tips: add extra constraints 🡪 what are the ways I can make change for W cents with for k coins

Ex make change with fewest coins: OPT(W) = min(OPT(W – w[1]), OPT(W-w[2]), …);

bestCoin 🡪 array: index = W, data = best coin to choose at that index (index in w; stores j for w[j])

countWays to make change: OPT(W,k): # make ways to make change for W with first k coins

OPT(W,k) sum(OPT(W – i\*kthCoinDenomination, k-1)) i goes 0 🡪 W/kthCoinDenomination

**Floyd warshall (algorithm below):** ALL SHORTEST PATHS 🡪 Dijstkas algorithm from each vertex O(V^3)

OPT(k,i,j) = shortest path from i to j, using only first k nodes as intermediate nodes.

OPT(k,i,j) = min( OPT(k-1,i,j), OPT(k-1,I,k) + OPT(k-1, k, j ) ) either we use kth vertex as intermediate node

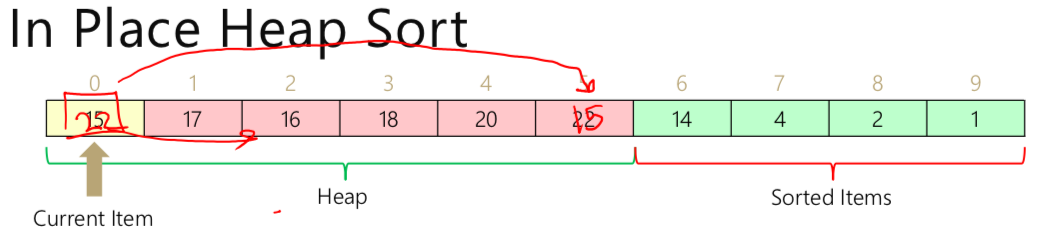
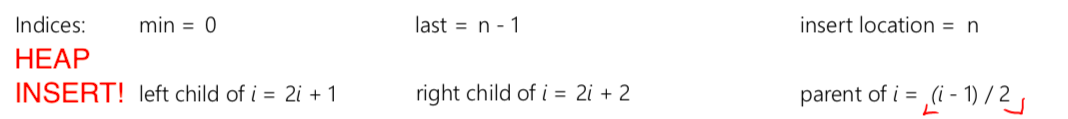
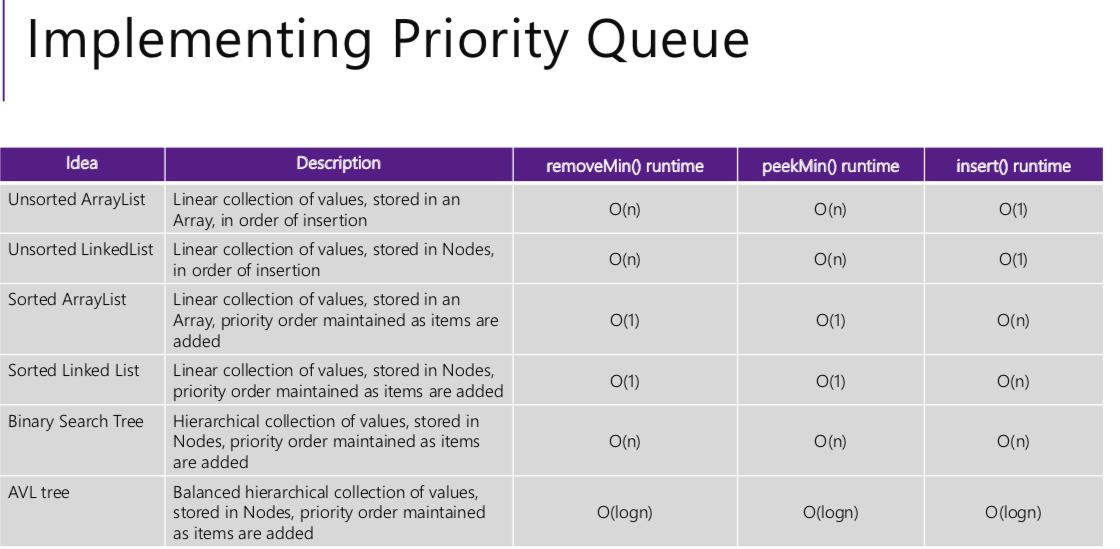
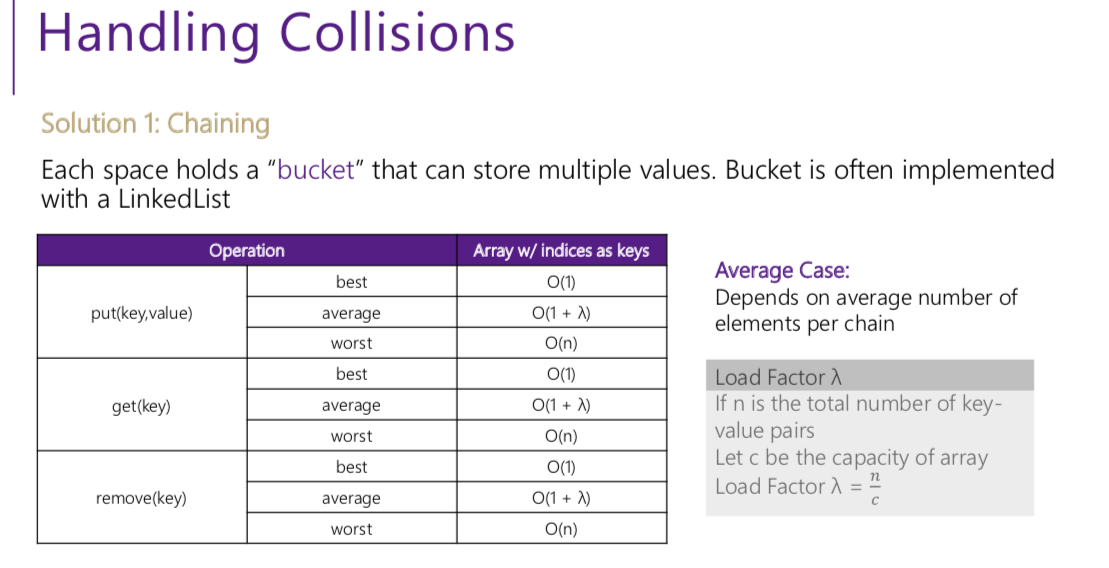
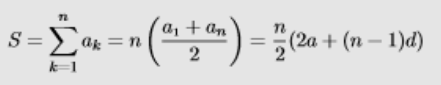
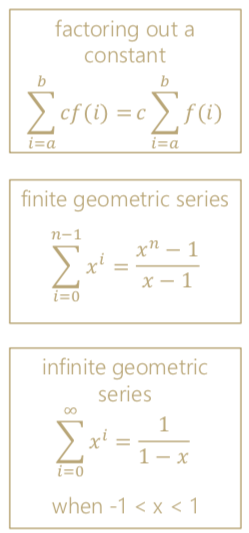
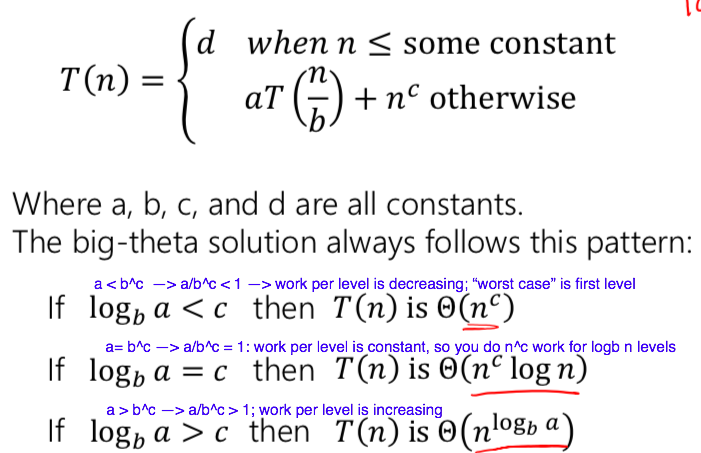
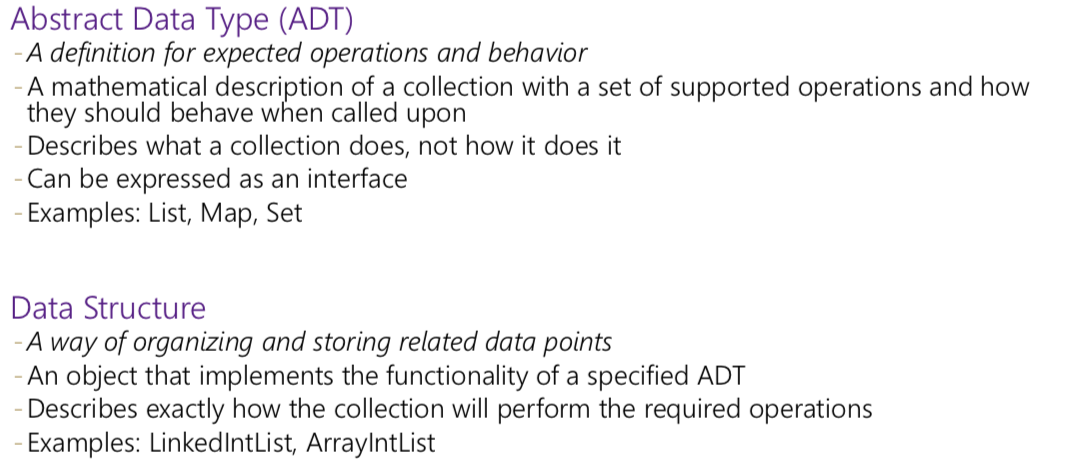
Or we don’t! (k is related to vertex label)

Prims – MST

\*\*Replace Distrkas u.dist + w(u,v) with just w(u,v)

1. Choose start; set dist = 0 and current = start;
2. Update adjacent V dists if edge curr🡪V < the current distance (mark curr as predecessor)
3. Choose min dist V to be new curr
4. Repeat

SCC algorithm – O(V + E)

­­­

LinkedList

* get = O (n)
* put = O(n) (at end; traverse list)

ArrayList

* get = O(1)
* put = O(n) (shift elements over)

Sequential Search (front to back): O(n)

Binary Search: O(log n)

Iterators traverse in O(n)! (get traverses in O(N^2) as it has to traverse to each element 1 by 1.

**Heap Sort**: O(n logn); can be in place (reverses order); unstable

**Selection Sort:** Scan through list repeatedly, move smallest element to front; best/worst O(N^2); in-place; stable

**Insertion Sort:** grab next element in unsorted, check where it belongs in sorted, place/shift stuff over. O(n^2) ave/worst, best O(n), stable and in place (works best on sorted/mostly sorted data)

**Merge Sort:** sort pairs of 2; sort pairs of 4; etc; two array optimization; good for large datasets that cannot fit on one machine; stable; not inplace

**Quick Sort:** pick next item as pivot; move smaller items to left of subarray; put pivot in middle. Choose new pivot in left subarray (repeat); same for right subarray (**O(n^2) WORST case = sorted in reverse/forward order 🡪 have to traverse entire list each time. As fividing into subarrays doesn’t divide work at all**

A function f(n) is **dominated** by g(n) if

f(n) <= c\*g(n) for all n > n0 > 0, c > 0.

O(f(n)) = all functions dominated (<) by f(n)

AVL Tree:

* Binary, BST, balanced;
* put/remove = O(logn); maintain order : O(1)

BST

* put/remove = O(height) = O(n); best = O(logn)

Min/Max Heap

* Binary, complete (and therefore balanced) NOT binary tree
* Build Heap O(n): put all elements in tree filling from top, down, left to right (complete). Start at bottom and percorlate down when needed.
* Insert (log(n)): put in next spot (bottom leftmost) and percorlate up
* removeMin(log(n)): remove top, replace with last element (bottom right) percorlate down if needed

Stack/Queue

* put/remove = O(1)
* For queue, assume back reference is saved

ArrayDisjointSet

* Path compression; during find set all intermediate nodes to point to representative element
* Union by height: if h1==h2 then h1-> h1 - 1
  + # at repr = - (height + 1)
* Union by size: newSize = s1 + s2
  + Max depth = log n

BST/AVL

* Full: all nodes have 2 children or leaf
* Complete: all levels filled except last; all nodes as far to left as possible
* Deletion: replace node with largest value in left subtree (if nonempty), or smallest in right (if nonempty) or remove (if empty); rotate if necessary (AVL)
* ROTATIONS: If longest path has a kink, rotate first to make longest path linear. Then rotate normally
  + NEVER >1 rotation/double rotation
* Height = # edges in longest branch BELOW; depth = # edges above

