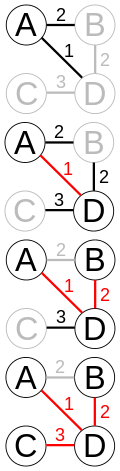
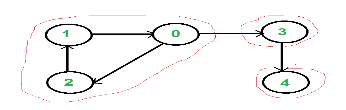
* Topological Ordering: vertices may rep. tasks to be done; edges may rep. constraints
  + Only possible if graph has no directed cycles; Must be directed acyclic graph
  + A valid sequence for tasks
* Dijkstra’s: shortest path from one node to all others; doesn’t work w/neg edge weight
  + Wiki says: O(E + Vlog(V)); greedy; notes say O(n2 log n2)
* Floyd Warshall: all pairs shortest path; Who wants to go through node 7? Node 8? Etc.
  + O(V3)
* Spanning Trees: min amt edges of the graph so everything is connected; not unique; root can be any node of the graph
  + Depth First: recursive; only visit unvisited nodes
  + Breadth First: use a queue; only visit unvisited nodes
  + Min. Spanning Tree: minimize weight of edges selected (this!=singleshortestpath)
    - Use when overall cost must be minimized, but indiv costs are no matter
  + Kruskal: make min span tree; alg: sort edges by weight, choose edges one at a time starting from least expensive but omit any edge that would make a cycle; O(E log E); greedy; undirected graphs only;  
    Will find forests if unconnected graph
  + <-Prim: makes min span trees; alg: (image)  
    only connected, unweighted graphs (forest can be found on individual connected components if the graph is not connected); greedy, O(V2) adjmatrix, O(ELogV) adjlist
* Euler Path/Trail: path that visits each edge exactly once; poss w/2 vertices w/odd num edges
* Euler Tour/Circuit/Cycle: an Euler path/trail that starts and ends at the same vertex; poss only if each node has an even number of edges
* Hamiltonian Path/Trail or Traceable Path: visits each vertex exactly once; finding one is NP-complete
* Hamiltonian Tour/Circuit/Cycle: Hamiltonian path that ends at the same place it begins
* Strongly Connected: you can get from anywhere to anywhere (directed graph)
* Connected: ditto strongly connected but for an undirected graph
* Strongly Connected Components: a set of nodes in a graph that are strongly connected, though the graph itself may not be (the graph must be directed)
  + Kosaraju’s alg: finds SCC; O(V + E); **1)** Create an empty stack ‘S’ and do DFS traversal of a graph. In DFS traversal, after calling recursive DFS for adjacent vertices of a vertex, push the vertex to stack. In the below graph, if we start DFS from vertex 0, we get vertices in stack as 1, 2, 4, 3, 0.  
    **2)** Reverse directions of all arcs to obtain the transpose graph.  
    **3)** One by one pop a vertex from S while S is not empty. Let the popped vertex be ‘v’. Take v as source and do DFS. The DFS starting from v prints strongly connected component of v. In the above example, we process vertices in order 0, 3, 4, 2, 1 (One by one popped from stack).
* Network Flow: directed graph where each edge has capacity and receives flow; get from the source (only has outgoing flow, does not receive) to the sink (only receives flow, does not send) w/ max flow
  + Flow into a node must equal flow out of a node
  + Augmenting path: a network is at max flow if and only if no augmenting path exists (a path which would allow more flow through)
  + Residual Graph: Graph representing the capacity left over given the existing flow; if an edge is at max flow, its direction will be reversed in the residual graph, ex: --2/2--> would become <--0/2—
  + Ford-Fulkerson Alg: computes max flow; greedy, O(Ef) e is edges f is max flow; steps: **1)** find an augmenting path **2)** compute bottleneck capacity **3)** perform augmentation of each edge involved and overall flow **4)** repeat until no augmenting edges exist
    - Bottleneck capacity: capacity of the edge that bottlenecks the rest in a particular path
    - When adding flow to forward edges, take away flow from backward edges
    - A full edge may become a backwards edge to create an augmenting path
    - Edges can be non-full and forward, or non-empty and backward

-**Hash Tables:**

**General:** generally O(1) lookup, insertion, and deletion

Rehashing: when things get bad, re-hash into a new table of twice the size;Expensive and inefficient; do when a large table gets about half full

**Cuckoo Hashing:** O(1) lookup time; Inserting may push an older key to a different location, but in a second table (there are exactly 2 tables, each w/unique hash functions); A key is always in its first or second choice

**Linear probing:** if desired location is full, look at next spot; Can make deletion complicated – must use lazy deletion;

Primary clustering: things that didn’t want to go to the same location compete for space

**Chaining:** a container at each location in the table (array, LL, vector, etc) to chain things that hash to the same value; Solves primary clustering, results in secondary clustering; Secondary clustering:  two records do only have the same collision chain if their initial position is the same

**Double Hashing:** if a collision occurs, each key jumps by a personalized increment, usually key % tablesize; Tries all keys before making it back to original key if tablesize and increment are prime relative to each other; Quadratic Probing: if hashed to H and there is a collision, try H + 12; H + 22, H + 32, etc; Can cause secondary clustering

-**Priority Queues:**

**Heap as an Array:** can be min heap or max heap; **Insertion:** Elements are inserted to the furthest but soonest left spot available so all but the last level is full; When inserting, look at parent and swap if needed; repeat until done or new node is at the root O(1) best and O(log n) worst; i = subscript of a node; 2i+1 and 2i+2 = i’s children; (i-1)/2 = i’s parents; always round down; **Deletion:** only ever delete the root – swap hole at the root w/ bottom-leftmost element and let heap sort itself as you would when inserting; O(log n)

**Leftist Heap:** keep track of null path length (npl) for each node; npl is dist. to nearest nullptr; npl of left child must be >= npl of right child; On merge O(log n) compare npl on the way back up and swap children as needed; Skew Heaps: always swap children after recursively merging; amortized O(log n); don’t keep track of npl

**Binomial Queues:** have the heap property; have B0, B1, B2 trees, etc. subscript denotes #nodes off root; B1 is 2 B0 trees, B2 is 2 B1­ ­trees, etc.; rules: may only have 1 of each kind of tree, but it is not req’d to have one of each; merge by attaching one tree to the root of another and keeping the heap property (only ever merge 2 of the same trees); have as few trees as possible and follow these rules; O(1) insert; O(log n) find min/max, delete min/max, merge

-**Sorting:** it is impossible to sort in O(logn) time or better

**Terms: Internal:** sort done in main memory; **External:** uses auxiliary storage (disk); **Stable:** retains original order in the case of duplicates; **Adaptive (non-oblivious):** takes advantage of existing order; **Sort-by-Address:** uses indirect addressing so the structure doesn’t need to be moved; **Indirect Sort:** make an array of pointers and sort those; used for very large records; **Inversion:** a pair of elements that is out of order; **In-Place:** uses at most O(1) aux space beyond initial array;

**Selection Sort:** 1) Find largest element, put it at the end and place what was there in the hole 2) find next largest element, put it at second to last 3) repeat; O(n2), can be stable by pushing back rather than swapping, not adaptive, in place

**Bubble Sort:** Compare 2 elements at a time and swap if necessary; repeat but exclude the last value from the prev run; repeat until sorted (stop when no swaps are done); O(n2), O(n) if nearly sorted, stable, adaptive if hasSwapped flag is included, in place

**Insertion Sort:** Take from an unsorted list and insert it in its proper place in a sorted list; uses 2 groups of keys, sorted and unsorted; O(n2), O(n) if nearly sorted, stable, adaptive, in place

**Shell Sort:** Sorts in stages: look at 2 things that are some gap away, sort (insertion), then repeat using a smaller gap, eventually w/gap of 1 (usually divide gap by 2); Worst: O(n2), avg bet: O(n1.17) and (n1.25), nearly sorted: O(nlogn); adaptive, unstable, in place

**Merge Sort**: divide into single elements; merge adjacent lists 1 at a time, sorting as you go; repeat until list is reconstructed in order; O(nlogn), stable, not adaptive, not in place

**Quick Sort:** pick a pivot; move it to 1st position, look for things smaller/larger than pivot; swap pivot into its proper place; recurse w/new partitions on either side of the old pivot; Worst: O(n2), Best: O(nlogn), not stable, not adaptive, not in place

**Heap Sort:** O(nlogn), not stable, not adaptive, in place;

**Bucket/Bin Sort:** all vals bet. 1 and k all are unique; place each element in its proper index(bin) of size k; Insert: O(n), print: O(n+k)

**TimSort:** Hybrid insertion & adaptive mergesort; looks for existing order, merge ordered sections 2 at a time until done; galloping; O(nlogn), not in place, stable, adaptive

-**Union – Find:** find: O(1), union: O(n)