CS2020: DS&A (Acc)

Java

- **Primitive types:** byte short int long float double boolean char
- **Access Modifier**

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
none	Y	Y	N	N
private	Y	N	N	N

- abstract - abstract classes without implementation
- **compareTo(T a)** -1 ($x < y$), 1 ($x > y$), 0 ($x == y$) (Total Ordering)
- **equals(Object o)** - Note, not T!

Basic Math

- **AP:** $a_n = a_1 + (n-1)d$
- **$\sum$ of AP:** $\frac{n}{2}[2a_1 + (n-1)d]$
- **GP:** $a_n = a_1 r^{n-1}$
- **$\sum$ of GP:** $\frac{a(1-r^n)}{1-r}$
- $\sum_{k=0}^{\infty} ar^k = \frac{a}{1-r}$ for $|r| < 1$

Masters' Theorem

- $T(n) = aT(\frac{n}{b}) + f(n)$ where $a \geq 1, b > 1$
- Case 1: $f(n) \in O(n^c), c < \log_b a$
  $T(n) \in \Theta(n^{log_b a})$
- Case 2: $f(n) \in \theta(n^c \log^k n), c = \log_b a$
  $T(n) \in \Theta(n^c log^{k+1} n)$
- Case 3: $f(n) \in \Omega(n^c), c > \log_b a$ & $af(\frac{n}{b}) \leq kf(n), k < 1$ & n sufficiently large
  $T(n) \in \Theta(f(n))$

Theory

- **O(n):** Upper bound
- **Θ(n):** Tightbound
- **Ω(n):** Lower bound
- **Amortized cost** Algo has amortized cost $T(n)$ if $\forall k \in \mathbb{Z}$, cost of k operations is $\leq kT(n)$

Basic Data Structures

- **Stack** (LIFO) - $O(1)$
- **Queue** (FIFO) - $O(1)$
- **Dequeue** - Both Stack & Queue (Can access FIFO or LIFO way)

Searching

- **Binary Search** $O(\log n)$: Keep track of a range, look at middle value and recurse appropriate half.
- **One sided** Binary Search: Suppose 1 side is bounded (i.e. answer between $[1, \infty)$). We try using the following sequence: $[1, 2, 4, 8, 16, 32, ..., 2^{k-1}, 2^k, ...]$ If it works for 2^k, then search on $[2^{k-1}, 2^k]$
- **Peak finding:** A[j] in array A is peak if (i) A[j] > A[j-1] & (ii) A[j] > A[j+1]. If only one item in array, it's vacuously true.
- **1D Peak Finding:** $O(\log n)$ Binary search on indices of A.
- **2D Peak Finding:** $T(m,n) = O(m+n)$
  Approach: Divide and conquer. Look at enclosure, cross and 4 quadrants.

Sorting (W/A/B/S)

- **BubbleSort:** Stable, In-Place W&A $O(n^2)$, B $O(n)$, S $O(1)$
- **SelectionSort:** In-Place ~~Stable~~ find minimum element and swap. W,A&B $O(n^2)$, S $O(n/1)$
- **InsertionSort:** In-Place & Stable Each item, move leftwards till in correct place for subarray. W $O(n^2)$, B $O(n)$, S $O(n/1)$
- **MergeSort:** Stable & ~~In-Place~~ W/B $O(n \log n)$, S $O(n)$
- **QuickSort:** In-place & ~~Stable~~ Partition, items less than left, more than right of pivot and recurse two parts. W $O(n^2)$, A/B $O(n \log n)$, S $O(n/n \log n)$
  - Good choice of pivot: partitions array into fractional halves per step: $\frac{k}{n}, \frac{n-k}{n}$
  - Bad choices of pivot partition the array by shaving off a constant amount (e.g. k and $n-k$). This will incur worst case running time of $O(n^2)$
- **Heap Sort:** In-place & ~~Stable~~ $O(n \log n)$, S $O(1)$ Using He
- **Counting Sort:** Only for numbers and the like. Basically: calculate frequencies, calculate cumulative frequency and output! $O(n)$

Linked Structure

- **SkipList:** Search/Insert/Delete: $O(\log n)$ Worse: $O(n)$. Space: $O(n \log n)$. Element appears with probability of p

Binary Trees

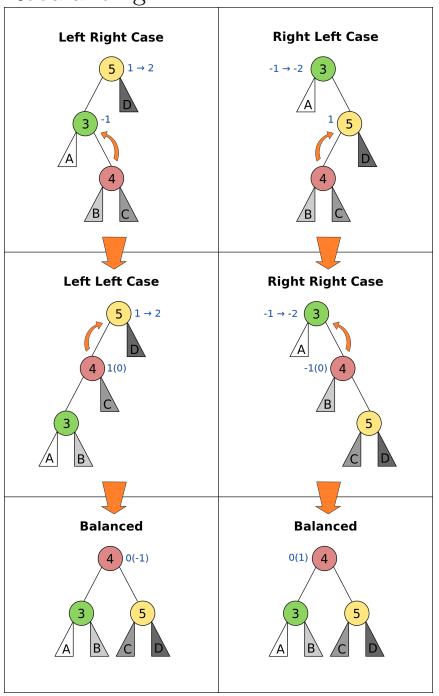
- H: `max(h(v.left), h(v.right)) + 1`
- **Traversal:** $O(n)$

Binary Search Trees

- Left Child < parent < Right Child
- **Running Time:** O(h) for almost all operations
- **Successor:** Search key (res ≤ key)
  - Case 1: Has Right Child – successor is right child min
  - Case 2: No right – go up till it's the left child – parent is successor
- **Delete:** 3 cases $O(h)$
  - Case 1: No children – just delete
  - Case 2: 1 child – link child to parent
  - Case 3: 2 child – replace deleted with sucessor
- **Trie:** Tree of words

AVL Trees

- **Height-balance:** `|v.l.height - v.r.height|` ≤ 1
- Rebalancing



- **Insertion:** BST insertion and walk up (only two rotations at most)

- **Deletion:** Do BST deletion and after deletion, for every ancestor, check if height-balanced (up to $\log n$ rotations)

Augmented Trees

- **Rank Tree:** store weight (i.e. no of child nodes) for each node!
- **Select** kth item in Rank Tree: r is the left tree weight. If $k < r$, choose left tree, else $k > r$, choose right (but $k = k - r$). If $k == r$, the node is the rank!
- **Determine** rank of item: r = left.weight + 1; node = this; while (node != null) { if node is right child { r += node.parent.left.weight + 1; } node = node.parent; } return r;
- **Interval Trees**: Sort by left end point of current node, store maximum right endpoint of children on each node.
- **Range Trees**: Store items such that range queries can be done (e.g. find ppl between 22 and 27 age)
- **Range Tree Implementation**: BST, store all points in leaves, internal nodes store MAX of leaf in the LEFT sub-tree.
- **Range Searching** $O(k + \log n)$ (k points of output):
  - Find split node: $O(\log n)$
  - Left Traversal: At every step either: (i) Output all right subtree and recurse left or (ii) recurse right.
  - Right Traversal: At every step either: (i) Output all left subtree and recurse right or (ii) recurse left.
- **2D Range Tree**: Each node has a y tree linked to it
- **Analysis of 2D Range Tree**: Query Time: $O(\log^2 n + k)$.
  - $O(\log n)$ to find split node, recurse steps
  - $O(\log n)$ y-tree searches of cost $O(\log n)$ each
  - $O(k)$ to enumerate
- **Space of 2D Range Tree**: $O(n \log n)$

Splay Trees

Tree that has most used item in root

- **Search/Insert/Delete:** Move to root (not rotate-to-root) and balance along the way. Two level at a time
- Three cases
  - Zig: Parent is root – rotate element to root
  - Zig-Zag: Parent is left, grandparent is right – two rotations
  - Zig-Zig: Parent is left, grandparent is left – rotate parent and rotate self

Hash Tables

- findMin, findMax, successor & predecessor: $O(n)$
- **Fingerprint Hash Table:** Don't store item, only store whether it exists! (0 or 1s)
- **Bloom Filter:** Multiple fingerprint hashes to reduce false positive. False negative only possible if deletion is allowed.

Hash Tables: Collisions

- **Chaining:** LinkedList in HashTable for collisions. Good if linked list is short
- **Open Addressing:** If collide, find another bucket through probing. Probing function **MUST** step through every bucket
- **Linear probing** Problem: clusters, if 1/4 full, clusters of size $\Theta(\log n)$
- Simple Uniform Hashing Assumption: every key is equally likely to map to every bucket
- Define $\alpha = n/m$ (avg # items / bucket) and assume $\alpha < 1$
- **Claim**: for n items in table size m, assume uniform hashing, expected cost of an operation is: $\leq \frac{1}{1-\alpha}$
- Double Hashing: $h(k, i) = f(k) + i \cdot g(k) \mod m$. Pick a $g(k)$ relatively prime to m

Hash Functions

- Division Method: $h(k) = k \mod m$
- Multiplication: $h(k) = (Ak) \mod 2^w >> (w - r)$

Graphs

- **Representations:** Adjacency List (array of nodes with a linked list) or adjacency matrix
- **Complete Graph:** n vertices with $\frac{n(n-1)}{2}$ edges. Approximate E to $O(V^2)$
- **Tree:** v nodes with $v - 1$ edges
- **Strongly Connected Component:** Graph component is strongly connected if every vertex is reachable from every other vertex in the component.
- **Triangle Inequality** dist(u, v) + dist(v, w) $\geq$ dist(u, w)
- **BFS:** Use a queue. $O(V + E)$
- **DFS:** Use a stack. $O(V + E)$ or $O(V^2)$ if adj. matrix
- BFS & DFS visit every node and edge

Digraphs

- In-degree – incoming edges. Outdegree – outgoing edges.
- Strongly connected: Two-way linkage
- **Topological Ordering:**
  - Sequential total ordering of all nodes and edges point forward ONLY
  - Topological Ordering is not unique! If two things have same dependency, either one can be done first.
  - **TopoSort: Post-order** Depth-First Search – print out node after visiting other nodes! $O(|V| + |E|)$ (List), $O(|V|^2)$ (Matrix).
  - **Alternative:** set of nodes with no incoming edges. Loop through.

Single-Source Shortest-Path Algorithms

- BFS finds minimum number of HOPS, not distance. (unless all edges have same weight!)
- **Bellman-Ford** Adj List: $O(VE)$ Adj Mat: $O(V^3)$
  - Relax all edges V times! (or till one relax all edges operation has no effect)
  - *Invariant:* After k iterations of relax, the k hop estimate for any node is correct
  - Run $|V| + 1$ to detect negative weight cycles
  - Bellman-Ford is able to find SSSP for graphs with negative

weights

- **Relaxation** means updating each node's estimate value
- For **trees**, relax edges in DFS or BFS order $O(V)$. Assume: Weighted (pos & neg) and undirected tree
- For DAG, use topological order! (DFS post-order) $O(E)$, but $O(V+E)$ to run Toposort

Dijkstras

- **Idea:** Running Dijkstra's creates a shortest path tree
- **Worst Running Time:**
  $O(|E|\log|V|)$ (Binary Heap, List)
  $O(|V|^2\log|V|)$ (Bin Heap, Matrix)
  $O(|E| + |V|\log|V|)$ (Fib Heap, List)
  $O(|V|^2)$ (Fibby Heap, Matrix)
  - Maintain distance **estimate** for every node
  - Begin with empty shortest-path-tree
  - Repeat: Consider vertex with minimum **estimate**, add vertex to shortest-path-tree, relax all outgoing edges from that vertex! $O(E)$ relax
- Don't use for negative weight edges!

Binary Heap

- **Properties:** It is a binary tree.
- **Max heap:** Parents are larger than children. For **Min heap**, it is the reverse!
- Used to implement **Priority Queue**
- **Insertion:** Add to end of array and bubbleUp!
- **Increase/Decrease Priority:** Change priority and bubbleUp/bubbleDown
- **DeleteNode:** Swap the item with the last item in heap, delete the last node and bubble down the item that is taking over
- **Extract Maximum:** Just take the first item and delete it (using the delete operation)!
- **Storing Heap:** Use an array with level order.
- **Finding:** $\text{left}(x) = 2x+1$
  $\text{right}(x) = 2x+2$
  $\text{parent}(x) = \text{floor}((x-1)/2)$
- All heap operations are $O(\log n)$, except createHeap/pred/sucessor, which is $O(n)$
- **Heap Merging:** $O(m+n)$ – Basically, manually readd all items!
- **Unsorted list to Heap:** (Heapify) $O(n)$ – recursively build the heap, start from right, bubbleDown each element!
- **Heap to Sorted Array** $O(n\log n)$ – extractMax and put into array! n extracts taking $O(\log n)$ each

UnionFind

- **QuickFind:** Find, Union: $O(1), O(n)$ – naive implementation
- **QuickUnion:** F&U: $O(n)$ – find parent for each set and union them together!
- **Weighted Union:** Find, Union: $O(\log n)$ – QuickUnion item with most children becomes parent during soyuz!
  Path Compression Find, Union: $O(\log n)$ – compress to grandparent or to root
- **Weighted Union with Path Compression** Find, Union: $O(\alpha(n)), O(\alpha(n))$

Minimum Spanning Trees

Spanning tree that has minimum overall weight!

- **Properties:**
  - Cutting an edge in MST results in two MSTs
  - No cycles in MST
  - Cycle – maximum weight edge is **NOT** in MST
  - Minimum weight of cycle is **not guaranteed** to be in MST
  - **Cut Property:** For every partition of nodes in the original graph (not MST), the minimum weight edge across the cut is in MST
  - **Corollary:** For every vertex, minimum outgoing edge is always part of MST
- **Prim's** $O(E\log V)$ – Using Priority Queue: Identify cut between rest of graph and the current MST, find minimum weight edge and add to MST (Basically, Dijkstra's modified)
- **Kruskal's** $O(E\log V)$ – UnionFind. List all possible nodes/edges by weight and take all of the cheapest, if they are not inside (or cause a cycle)!
- **Boruvka's** $O(E\log V)$ – for every node, add minimum edge outgoing. After that, consider the connected component as a 'node' and recurse.
- **BFS/DFS** $O(E)$ – only for same weight graph

Applications of MST

- Network Design
- Approximation for NP-hard problems, such as Steiner trees and TSP
- Max bottleneck paths (using DFS)

MST Variants

- **Directed Minimum Spanning Tree** - for every node except root, add minimum weight *incoming* edge. For DAG only! $O(E)$
- **MaxST** - negate all weights and run any MST algorithm

Steiner

(find MST for subset of points, but can use other points too)

- Let O be OPT tree. Let T be SteinerMST tree.
- Let D = DFS on O. cost(D) = 2*OPT.
- $D = \{H, A, G, D, G, E, G, A, H\}$
- Skip Steiner Nodes: $D' = H, A, D, E, A, H$
- $cost(D')$ = cost of traversing shortest paths $< cost(D) < 2 * OPT$.
- $cost(T) < cost(D')$

Steiner Tree Algorithm (Approx)

- For every pair of required vertices (v,w), calculate the shortest path from (v to w)
  - Dijkstra V times
  - Floyd-Warshall (APSP)
- Construct new graph on required nodes
  - V = required nodes
  - E = shortest path distances
- Run MST on new graph
  - Use Prim's or Kruskal's

– MST gives edges on new graph

All Pairs Shortest Path

- **Floyd-Warshall:** For negative weight edges. Only Matrix data struct $O(|V|^3)$.
- **Dijkstra (Binary Heap)** $O(|V||E|\log|V|)$ (List) $O(|V|^3\log|V|)$ (Matrix)
- **Dijkstra (Fib Heap)** $O(|V||E| + |V|^2\log|V|)$ (List) $O(|V|^3)$ (Matrix)
- **BFS** (Only for Unweighted) $O(|V||E| + |V|^2)$ (List) $O(|V|^3)$ (Matrix)

Dynamic Programming

- **Top-Down Approach (Memoization)**: Store all intermediate results and if there's repetition, retrieve the previous result
- **Bottom-Up Approach (Dynamic Programming)**: Start from base cases, work upwards towards actual problem state and keep results of states that might be repeated
- **Floyd-Warshall**: $O(V^3)$.
  Idea: Update Bellman-Ford equation in the adjacency matrix.
  Invariant: At step k, shortest path via nodes 0 to k are correct
  Storing APSP paths can be done in $O(V^2)$ by pointing to next node on shortest path!

Network Flows

- Source and target are **k-edge-connected** if there are k edge-disjoint paths (i.e. paths that don't use any paths) from the source to target
- **Maximum flow** is defined as the st-cut with minimum capacity (minimum outgoing from s-area, but don't consider incoming)
- **Minimum Cut** (For graphs with no augmenting paths): Let S be the nodes reachable from the source in the residual graph. $T =$ all other nodes. $S \to T$ is minimum cut.
- **Ford-Fulkerson**
  – Start with 0 flow
  – While there exists an augmenting path:
  – Find an augmenting path
  – Compute bottleneck
  – Increase flow on the path by the bottleneck capacity
- **Residual Graph** – amount that flow can be increased
- **Augmenting Path** – the path in the residual graph from s to t that has NO 0 weight edges
- **Time Complexity Based on Augmenting Path algo**
  – **DFS** $O(F \cdot E)$
  – **BFS** $O(VE^2)$
    * Shortest augmenting path
    * Edmonds-Karp
  – **Dinitz:** $O(V^2E)$
  – **Fattest-Path Search:** $O(E^2\log V\log F)$