Master Theorem: If $T(n) = aT(\frac{n}{b}) + f(n)$ and

- $f(n) = O(n^{\log_b a-\epsilon}) \Rightarrow T(n) = \Theta(n^{\log_b a})$
- $f(n) = \Theta(n^{\log_b a} \lg^k n) \Rightarrow T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$
- $f(n) = \Omega(n^{\log_b a+\epsilon})$ and $af(\frac{n}{b}) \leq (1-\delta)f(n) \Rightarrow T(n) = \Theta(f(n))$

Priority Queue: Implements a set of elements each associated with a key. Operations are insert, max, extract_max which pops, and Increase_key.
Heap: Implements a priority queue, takes an array and visualizes it as a nearly complete binary tree. In particular, a max heap will be a useful data structure.

- max_heapify: If left(i) and right(i) are max heaps while $A[i]$ is a violation, then just switch $A[i]$ with the larger continuously so that it trickles down the tree. For a single violation the complexity is $O(\lg n)$ as we may have to trickle down the whole tree.
- build_max_heap: We can do a similar thing to build, just heapify all the leavers level by level starting from the leaves. Suppose we're i levels above the leaves. We would need i time to trickle down for each node and since there is $\Theta(\frac{n}{2})$ leaves, there are $\frac{n}{2^{i+1}}$ nodes at that level. Thus the total time is $n \sum \frac{i}{2^{i+1}} = \Theta(n)$.

Heap Sort: Build a max heap, extract the max value by switching with last leaf, then max heapify. Complexity is $O(n \lg n)$.
Binary Search Tree: Tree with pointers towards parent and child(s). All elements in the left subtree of a node must be smaller than the node element, and in the right greater.

- Insertion is simple, just follow the pointers until a proper placement is found. We can also check if the item is within "k" if some element during insertion. We are basically following a binary search algorithm. You can do the same for finding a value. Minimum/sorting just requires you to follow left pointers.
- Computations are centered around the height of the tree $O(h)$

Balanced BSTs i.e. AVL: Makes the trees balanced so $h = O(\lg n)$. The invariant is that subtree heights differ by at most 1. Inserting an element can restore the invariant by a constant number of rotations.
Counting Sort: Sorting n elements into keys $0, 1, ..., k-1$ takes $O(n+k)$ and is STABLE (order of elements don't change).

- Making an array of k empty lists takes $O(k)$
- Appending each key to the corresponding list in the array takes $O(n)$ time
- Output the items into an array takes $O(n+k)$

Radix Sort: Do things in base b so there are $d = \log_b k$ digits in $0, 1, ..., b-1$ then count sort on each digit from least significant to most. Each digit takes $O(n+b)$ time so total is $O((n+b)\log_b k)$ which is minimized when $b = n$ to be $O(n \log_n k) = O(n)$ if $k \leq n^c$.
Hashing: ADT is dictionaries which are basically sets of key-value pairs. Hashing basically narrows a large set of n keys to a smaller set of m keys i.e. 0 to $m-1$ using a hash function.

- Good hash functions have a random output in $0, 1, ..., m-1$
- Since collisions can occur, values should be linked lists i.e. resolving collusion via chaining. However if all collide we can have $\Theta(n)$ time for search. Instead we look at the average cost: Let $\alpha = \frac{n}{m}$ be the "load factor" we expect the lists to have a length of α so the running time is $\Theta(1+\alpha) = \Theta(1)$ if $m = \Theta(n)$.
- Good hash function ex: $h(k) = [(ak+b) \mod p] \mod m$, a, b random, p large prime so that prob. of collision is less than $\frac{1}{m}$ (UNIVERSAL HASHING, $h_{a,b}$ forms a universal family)

Table Doubling: We need to make sure $m = \Theta(n)$ when inserting so table needs to be resized.

- Idea is to double whenever $\alpha = 1$ but elements have to be rehashed as well
- Cost of inserts = cost of inserting + cost of table doubling; the former is n while the latter is $1+2+4+...+n \leq 2n$ so the total cost is less than $3n$ but the amortized cost (per insert) is $\Theta(1)$.

Rolling Hash: If we want to pattern match a desired word of length m in a document of length n, matching the characters of the word through the document naively takes $\Theta((n-m)m) = \Theta(nm)$.

- Have a sliding window and cleverly hash as the window slides using the characters as numbers in base b into q keys
- m time to initialize the hash of the desired word, window has to slide $n-m$ times, $\frac{n-m}{q}$ expected hits taking m time to verify, and mv time to return v hits, totalling to $\Theta(n+mv) = \Theta(n)$ when $q \geq m$ and $m << n$.

Open Addressing: What if we want there to be only 1 element per key? Table doubling is obviously a necessity and typically $\alpha < \frac{1}{2}$. A probing sequence $h(k,i)$ is used

- Prove sequence $h(k,0), h(k,1), h(k,2)...h(k,m-1)$ must be a random permutation of the $m-1$ keys. Uniform hashing is usually assumed so that all permutations are easily likely
- A good approximation is double hashing i.e. $h_1(k) + ih_2(k) \mod m$, where h_2 doesn't map to 0. Linear probing i.e. $h_2(k) = 1$ is bad since it leads to clusters.
- Search is done by using the probe sequence keys. Insertion is done by using first empty slot in probe sequence. Deletion requires a DEL flag so that search doesn't interpret an empty slot.
- Unsuccessful search takes $\sum \alpha^k = \frac{1}{1-\alpha}$ probes on average while successful is around $\frac{1}{n}\sum_{i=0}^{n-1} \frac{m}{m-i} = \sum \frac{\alpha^k}{k+1} = 1.4$ for $\alpha = \frac{1}{2}$

Graphs: V :=set of vertices, E :=set of edges as pairs i.e. (v, w), if directed means $v \to w$, Adjacency is list of neighbors of each vertex $u \in V$ i.e. $A[u]$ is $u's$ neighbors.

- BFS: Focuses on searching everything at each level i.e. level i consists of vertices reachable in i steps but not fewer. Idea is to store the level of each vertex (so starting vertex initialized to 0 and the rest to ∞), also parents initially point to null, then expand the frontier level by level updating the attributes. Frontier should keep vertices in order visited so that vertices of level i are visited before those of level $i+1$. Complexity is $O(V+E)$ since each vertex is visited at most once and the Adjacency of each vertex is exhausted. BFS finds shortest path as well using parent pointers.
- DFS: Focuses on going deep as possible first. Parents initialized to $s : None$
  - DFS-visit: starts by visiting the Adjacency of s and if $v \in Adj[s]$ is not in the parent dictionary, then parent$[v]$=s and then v gets visited (remember Adj is forlooped).
  - DFS: Forloops vertices in V, and if it isn't in the parent dictionary then perform DFS-visit on it. DFS also visits every Adj and vertex so it's $O(V+E)$ time.
- Edge Classification (DFS): Forward edges points to descendants, back edges point towards ancestors,. and cross edges connect two separate DFS sections. Edges can be classified based on DFS-visit timing. Undirected graphs can only have tree and back edges.
- Cycle Detection: Cycle $\Leftrightarrow$ DFS has back edge. Backwards direction easy but forward direction relies on looking at first vertex visited by DFS.
- Topological Sort: Given a Directed Acyclic Graph, perform DFS starting at source vertices (no incoming edges) and record finishing times. Reverse the order i.e. finishing first should come last to get the sort.

Shortest Paths: Each path now has an associated weight $w(p)$, we want the minimum $w(p)$ for each pair of vertices i.e. find $\delta[u,v]$. Note $\delta[u,v] + \delta[v,w] \geq \delta[u,w]$ by Triangle Inequality and we "relax" edges by assigning $d[v] \leftarrow d[u] + w(u,v)$ if $d[v] > d[u] + w(u,v)$. The following algorithms will rely on relaxation.

- DAGs: No negative cycles possible, topologically sort the DAG, then relax each edge coming out of it. Complexity is $\Theta(V+E)$ since each edge and vertex is visited once.
- Djikstra: Based on the fact that when using gravity with balls and string length, vertex processing can be seen. No negative cycles. Maintains a set S whose final shortest path weights are determined. Select $u \in V - S$ with shortest path estimate (priority queue), adds it to S and relaxes all edges out of it. In other words if Q is a priority queue, $S \leftarrow \phi$, $Q \leftarrow V[G]$, then while $Q \neq \phi$, $u \leftarrow$ EXTRACT-MIN(Q). $S \leftarrow S \cup u$, RELAX (decreasing keys) (u,v,w) for all edges $(u,v) \in$ Adj$[u]$. Array has $\Theta(v)$ to find min and $\Theta(1)$ to decrease key so in total we have $\Theta(V^2+E) = \Theta(V^2)$. Min-heap $\Theta(\lg V)$ for extracting min and decreasing key so $\Theta(V \lg V + E \lg V)$. For a Fibonacci Heap same but $\Theta(1)$ amortized for decrease key so optimal is $\Theta(V \lg V + E)$.
- Bellman-Ford: Detects negative weight cycles. Makes $|V|-1$ passes and relaxes every edge. This would guarantee shortest paths since every vertex is at max $|V|-1$ segments away from each other unless there is a negative weight cycle. If that were to occur we would be able to still relax an edge so check for relaxation at $|V|$th step and if yes then report negative weight cycle. Complexity is obviously $O(VE)$.
- Single-source, single-target: Stop Djikstra once target has been found
- Bi-Directional Search: Alternate searching forward from source and backwards from target. Terminates when a vertex is deleted from queue of both searches then find node x such that sum of forward and backward weights is minimized.