Data Structures: store, access, and manage data
Algorithms: affect performance and scalability

Voting Algorithm: find the majority element
Intuition: the majority item must be the majority candidate at some point
- Constant Memory: 2 variables m = null (majority) and c = 0 (count)
- iterate through items x and
    - if c == 0, let m = x and c = 1
    - else if m == x, c += 1
    - else c -= 1
- once the algorithm finishes, item m must be the majority item if there exists a majority

Generalized Majority Algorithm: find k items appearing more than N/(k+1) times
- Use k (majority, count) tuples
- iterate through items x and
    - if some x == m_i, c_i += 1
    - else if some m_j == 0, let m_j = x and c_j = 1 (only one)
    - else decrement all counts
- once algorithm finishes, items m will be the k items identified

Big O (Upper Bound): T(N) = O(f(N)) if T(N) <= c * f(N) when N >= n
- bounded above by some c * f(N)

Big Omega (Lower Bound): T(N) = Omega(g(N)) if T(N) >= c * g(N) when N >= n
- bounded below by some c * g(N)

Big Theta (exact); T(N) = Theta(h(N)) if T(N) = O(h(N)) = Omega(h(N))
- T(N) is the growth complexity of the algorithm

Max Subsequence Problem: find contiguous sequence with the largest sum
Intuition: prefix sum cannot be negative, otherwise we can omit it and start with a positive
- keep track of maxSum and currentSum (init = 0)
- loop through elements in array
    - add element to currentSum
    - if currentSum > maxSum, replace maxSum with currentSum
    - if currentSum becomes negative, reset to 0
- return final maxSum value

Metrics to Measure Algorithm Performance: based on bottlenecks
- Running time
- Memory required
- (not so much) ease of programming

Runtime Difference Factors:
- Hardware
- Programming Language
- Programmer

Average Case Runtime: distributions hard to predict
Best Case Runtime: unrealistic
Worst Case Runtime: absolute guarantee

Sets: collection of unique members
- **Abstract Data Type:** set and operations

Dictionary: key value pairs with unique keys
- order is not supported, only equality

Hashing: each item uniquely identified by a key
- **Birthday Paradox:** with a very number of keys, there will be a hash collision
- Hash functions should spread keys evenly on structured inputs and quick to compute
- **Chained hashing:** each key is a linked list bucket of values
    - Worst Case for Insert, Delete, Find: O(n) time = Linked List
- **Hash Code Map (for non integer keys):** key -> hash (integer)
- **Compression Map:** hash -> table entry
- **Polynomial Accumulation:** take ascii values of strings, and then view them as coefficient of a polynomial
- Good functions usually involve prime numbers not close to powers of 2 (for mod or polynomial)
- **Deterministic:** h(k) always maps to the same value
- **Linear Probing:** insert at next available slot
    - On remove, add tombstone to signify element is empty, but previously deleted
- **Representative Analysis:** Average Case Behavior
- **Quadratic Probing:** insert at H(x) + 1, H(x) + 4, H(x) + 9...
- **Double Hashing:** second hash function if first one has a collision
- Rehashing: build another table twice as big and prime to prevent operations from taking too long

Hashing Analysis:
- Load factor: L = alpha = n / m (m slots and n elements)
- Worst Case: n keys to a single slot is O(n) + time to compute h(k)
- Average Case: depends on h(k)
- Simple Uniform Hashing w/ Chaining: O(1 + L) where L is the average length of the Linked List
- Random Probing: (a = 0.5-0.7 ensures constant behavior)
    - Unsuccessful: O(1/(1-L)) to find an empty bucket
    - Successful: O(1/L ln(1/1-L))

Perfect Hashing

- O(1) worst case lookup
- Static set of N keys
- No collisions/insert/delete
- Built in O(n) space O(n) time

Binary Trees:

- Full: all the leaves are at the same level and non-leaf nodes have 2 children
- Height(Empty Tree) = -1; Height(Singleton) = 0
- Complete: Full binary tree + 1 layer that is full from the left (and empty on the right)
- Problem: height is unbounded

Priority Queue: set of elements with priorities

- No find operation, only insert and deleteMin or deleteMax

Heap:

- **Shape:** complete binary tree
- **Order:** parents > children (max heap); parents < children (min heap)
- **Insert:** insert at next available slot and bubble up to keep ordering property => O(log n)
- **Delete:** remove extreme element, move last element to top and trickle down => O(log n)
- **Build Heap:** arrange array as heap; then iterate through each non leaf node backwards and trickle down => O(n)

Leftist Heaps: merge operation in O(log n)

- Maintains order property
- **Null Path Length:** NPL(x) shortest path from x to a node without two children
- **Leftist Property:** for every node x, npl(left(x)) >= npl(right(x))
  - Right path is as short as any in the tree
- **Theorem:** a leftist tree with r nodes on the right path must have at least $2^r - 1$ nodes
- **Merge Algorithm:**
  - If one tree is empty, return the other tree
  - Else, recursively take right subtree of left tree and merge with the other tree
  - Check if NPL condition is satisfied, else swap right with left
- **Insert:** merge with singleton node
- **Delete:** remove first node, and merge left and right leftist trees

Balancing BSTs:

- Motivation: want search to be O(log n)
- Attempt: every node has same height for left and right subtree
  - Problem: insert and delete requires too much maintenance (may not be logarithmic for insert/delete)'

AVL Trees: binary search tree + height constraint

- **Constraint:** difference in heights of left and right subtrees for every node must be at most 1
- Empty Tree Height = 0; Singleton Node Height = 1
- Properties:
    - upper half is full, lower half has gaps
    - height < 2 log(n)
    - $2^{(h-1)/2} < n < 2^h$
- Insert: same as BST; if violates balance, must rotate subtrees
    - Single Rotation: linear
        - rotate root of imbalance with child (child's subtree becomes root's subtree)
    - Double Rotation: nonlinear
        - rotate child with grandchild (grandchild's subtree becomes child's subtree)
        - rotate root of imbalance with new child (new child's subtree become root's subtree)
- Delete: same as BST with 3 cases:
    - A node with one child: replace parent with child and delete old child
    - A leaf: trivial in some cases, however, if unbalanced:
        - find root of imbalance (Z)
        - choose (Y) subtree with larger height
        - choose (X) subtree be subtree of (Y) of larger height
        - rotate X Y Z (linear or nonlinear case)
            - may need to recursive process up the tree until height is balanced for entire tree
    - An interior node

Sterling's Property: n! => (n/2)^(n/2) … thus, log(n!) => n log(n)

Red-Black Tree:

Properties:

- Every node is either red or black
- The root is always black; every leaf node is black
- A red node cannot have a red parent or red child
- Every path from a node to any descendant null node has the same number of black nodes

Theorem 1: In a red-black tree of height h, at least half the nodes of any path from the root to null leaf must be black

Theorem 2: In a red-black tree, no path from any node X to a null leaf is more than twice as long as any other path.

Multi-way Search Tree: generalization of binary search tree

- Has at least two children
- stores collection of items in form of <key, value>
- Contains d - 1 <key, value> pairs where d is the number of children

Graphs:

- A graph (G) is composed of a set of vertices (V) and a set of edges (E)
- An edge (e) connects a pair of vertices (u, v)
- Degree of a vertex = number of edges connected to that vertex
- Sum of degree of vertices = 2 * edges
- Directed graphs: edges have direction
- In-degree: edge point inward toward vertex
- Out-degree: edge point outward away from vertex
- Note: no self loops nor multiple edges
- Path: sequence of vertices such that consecutive vertices are adjacent
- Simple Path: no repeated vertices
- Cycle: simple path with the last vertex being the same as the first
- Connected Graph: any two vertices are connected by some path
- Subgraph: subset of vertices and edges that make up a new graph
- Connected component: maximal connected subgraphs
- Tree: connected graph without cycles
- Forest: collection of trees
- Complete graph: all pairs of vertices are adjacent
- Adjacency Matrix: 1 in matrix if there's an edge that connects row and column => O(n^2)
- Adjacency List: linked list of vertices to edges => N + 2M where O(n) is vertices and O(m) is edges

2-4 Trees: height balanced search trees (worst case complexity is bounded)

- multi-way search trees with at least 2 children and at most 4 children
- all leaf nodes are at the same level
- $\log_4(N)$ < height < $\log_2(N)$ => height = O(log n)
- Search: O(log n)
- Insertion: O(log n) splits (constant) worst case
    - insert at leaf node (may reshuffle order of keys in leaf node)
    - if leaf node is full and parent node is not full, move middle key up and split node to create two new leaf nodes
    - if parent node is full, split and move middle key up until no conflict
- Deletion: O(log n)
    - trivial case: delete key with other keys in the leaf (may need to shift)
    - delete parent node: replace key with predecessor (leaf key)
    - delete only key in leaf: borrow key from sibling (must rotate with parent)
        - if sibling is empty, must merge the siblings and reduce a parent key into the sibling
        - if parent becomes empty, must merge and recursively go up to reduce height of tree

Breadth-First Search: traverses a connected component of a graph and defines a spanning tree

- add root to queue
- remove element from queue and add all neighbors with value 1
- remove from queue and add neighbors to end of queue with value += 1
- continue until all neighbors are explored
- color the elements depending on whether they have been included, in the queue, or not included
- Property: BFS finds all nodes reachable (also the shortest path) from a source S
- Property: if an edge connects two nodes, those two nodes can differ by level numbers by at most one

BFS Application: Connected Components

- Algorithm: Count total number of nodes that enter and exit queue: O(V + E)

BFS Application: Bi-Partite Graphs

- Disjoint subsets of vertices X and Y
- Edges connect node of X to node of Y
- Algorithm: each level must alternate in colors to verify bi-partite graph

Depth-First Search: traverses a connected component of a graph going deep as possible first, defining a tree

- mark element as visited
- iterate through each adjacent element of source
    - for each element, if not traversed, perform DFS on that element
- continue until all elements are explored
- Time Complexity: O(V + E) since traverses V vertices + 2E edges

Depth First Search Application: Two-Edge Connectivity

- Two-edge connected: A graph that can remove any one of its edges and still stay connected
- If a bridge to a subtree (the sibling is non null) is removed and the graph remains connected, then the graph is two-edge connected

Directed Graph Edges: for edge from u to v

- Tree edge: edge that is part of the tree
- Back edge: edge that connects a node to its ancestor
    - arrival(u) > arrival(v); departure (u) < departure (v)
- Cross edge: edge that connects siblings
    - arrival(v) < departure(v) < arrival (u) < departure (u)
- Forward edge: edge that satisfies below property but not used to create the tree
    - arrival(u) < arrival(v); departure (u) > departure (v)

Connected Graphs: path between every pair of vertices for undirected graphs
Strongly Connected Graphs: path between every ordered pair of vertices for directed graphs

DFS Application: Cycle

- If there's a back edge, there's a cycle

Minimum Spanning Tree

- Spanning Tree has n-1 edges (n vertices) that connects all the vertices
- Minimum spanning tree: spanning tree of minimum height
- Edge has associated weight
- length of spanning tree = sum of weights of edges

Kruskal's Algorithm (Greedy Algorithm): MST on undirected graphs

- **Greedy Algorithm:** choose best choice available at each level (may not be optimal solution in some contexts)
- Sort edges in increasing order
- Choose smallest edge on entire tree
- Subsequently, choose the smallest edge without creating a cycle
- Once the spanning tree has been constructed, end of program

Proof:

- Simplify by assuming all weights are unique
- Let the algorithm produce a tree with n-1 edges.
- Assume that this is not the minimal tree. Thus, there exists an optimal tree with n-1 edges smaller than the sum of the previous edges. Since these two are not identical, find the first smallest mismatch in tree weights.
    - Case I: new < old, then new edge must be not any of the other edges in old tree and create a cycle; it is also greater than all of the old edges in the cycle (old edge will have to be smaller, because we select the one with lower cost) => new is the bigger tree, this scenario not possible
    - Case II old < new: skipped over old edge because it created a cycle in the optimal tree, or it was not the optimal tree, thus contradiction

Prim's Algorithm for MST:

- Cut in an undirected graph: partition of the vertex set into two parts
- Max # of cuts = 2^{n-1} for n vertices (power set, ignore permutations)
- Minimum edge in the cut must be contained in spanning tree
- Contradiction: add any other edge and minimum edge creates cycle; must use minimum edge
- Algorithm: start with one vertex on one side, and start including smallest edges and vertices into the cut

Dijkstra's Algorithm: Single Source Shortest Paths

- Given a graph (directed and undirected)
- Each edge has length that is nonnegative
- Given source and destination, find shortest path
- If there is a common midpoint, can break into smaller subparts
- Subproblem: find shortest path from s to every vertex
- For all vertices, initialize shortest path = infinity in an array and insert into heap
- set source destination = 0 and decrease priority of source to 0
- while heap not empty:
    - remove minimum vertex
    - for all vertices adjacent to to the vertex, if direct path is longer, update direct path to new shortest path (length from min vertex to new vertex, plus length to min vertex) and decrease priority of that vertex to the new path length
- Runtime: O(m * log(n)) where m is edges and n vertices

Proof by Induction:

3 Cases:

- direct path => base case
- indirect path => inductive case
- indirect path + new point (inc earlier) => source -> new point < source -> indirected -> new point (already accounted for)

Continuation:

- Assume not shortest path (induction by contradiction)

Negative case: negative cycles