Binary Search Tree (BST)

Properties: Left subtree < node < right subtree. Height $h = O(\log n)$ balanced, $O(n)$ worst.
Search: Time $O(h)$. Compare key, go left if smaller, right if larger, return if found.
Deletion (4 Cases):
1. *Node is leaf:* Simply remove it.
2. *Node has only left child:* Replace node with left child.
3. *Node has only right child:* Replace node with right child.
4. *Node has two children:* Find successor (min in right subtree) or predecessor (max in left subtree), replace node's value, delete successor/predecessor.

B-Trees

Properties: Minimum degree $t \geq 2$.
- Root: ≥ 1 key, ≥ 2 children (if not leaf)
- Non-root: $\geq t-1$ keys, $\geq t$ children
- All nodes: $\leq 2t-1$ keys, $\leq 2t$ children
- All leaves at same depth
- Height: $h \leq \log_t \frac{n+1}{2}$
- Min keys: $1 + (t-1)\sum_{i=1}^{h} 2t^{i-1} = 2t^h - 1$
- Max keys: $(2t-1)\sum_{i=0}^{h}(2t)^i = (2t)^{h+1} - 1$

Search: Time $O(th \log n)$. Binary search within node, recurse to child.
Insertion: Time $O(th \log n)$.
1. Search for position in leaf node
2. If leaf not full: insert key
3. If leaf full (has $2t-1$ keys): split node at median, promote median to parent
4. If parent also full: recursively split up to root if needed

Deletion Cases:
1. *Simple delete from leaf:* Remove if node has $\geq t$ keys
2. *Internal node, left child has $\geq t$ keys:* Replace with predecessor, recursively delete predecessor
3. *Internal node, right child has $\geq t$ keys:* Replace with successor, recursively delete successor
4. *Internal node, both children have $t-1$ keys:* Merge key with both children, recurse
5. *Node has $t-1$ keys, sibling has $\geq t$ keys:* Borrow from sibling through parent
6. *Node and all siblings have $t-1$ keys:* Merge with sibling and parent key

When to use: Databases, file systems, many keys, disk-based storage.

Red-Black Trees

Properties:
- Every node is red or black
- Root is black
- All leaves (NIL) are black
- Red node has black children (no consecutive reds)
- All paths from node to leaves have same # black nodes (black-height bh)
- Height: $h \leq 2\log_2(n+1)$
- Black-height: $bh \geq h/2$
- Min nodes: $n \geq 2^{bh} - 1$

Rotations:
- *Left-rotate(x):* x's right child y becomes parent of x; x becomes y's left child; y's left subtree becomes x's right subtree
- *Right-rotate(x):* Mirror of left-rotate
- Time: $O(1)$, preserve BST property and black-height

Insertion (4 Scenarios): Time $O(\log n)$. Insert as red node, then fix violations.
1. *z = root:* Color z black
2. *z.uncle = red:* Recolor parent, uncle to black; grandparent to red; recurse on grandparent
3. *z.uncle = black, triangle:* Rotate parent opposite direction to make line case
4. *z.uncle = black, line:* Rotate grandparent, recolor original parent black, original grandparent red

Deletion (3 Methods): Time $O(\log n)$.
1. *Transplant:* Replace subtree rooted at u with subtree rooted at v
2. *Delete:* Find node, use transplant, track color of removed node
3. *Delete-Fixup:* If removed was black, fix violations through 4 cases: sibling red, sibling black with black children, sibling black with red left child, sibling black with red right child

When to use: Guaranteed $O(\log n)$ operations, frequent insertions/deletions, balanced tree needed.

Left-Leaning Red-Black Trees (LLRB)

Additional Properties:
- Red links lean left (no right-leaning red links)
- No node has two red links
- Perfect black balance
- Corresponds to 2-3 tree: red link = 3-node
- 2-node in 2-3 tree = single black node in LLRB
- 3-node in 2-3 tree = black node with left red child in LLRB

Insertion: Time $O(\log n)$.
1. Insert as red node (like BST)
2. If right child red and left child black: left-rotate
3. If left child red and left-left grandchild red: right-rotate
4. If both children red: flip colors (node becomes red, children black)
5. Propagate fixes upward to root
6. Root always colored black at end

Deletion via 2-3 Tree Method (BEST APPROACH): Time $O(\log n)$.
1. **Conceptual approach:** Think in terms of 2-3 tree operations, then translate to LLRB
2. **Key insight:** Never let deletion path encounter a 2-node (would become empty)
3. **Going down (make 3-node or 4-node on path):**
   - If current node is 2-node: borrow from sibling or merge with sibling and parent key
   - Ensure current node becomes 3-node or 4-node before descending
4. **At bottom:** Delete from 3-node or 4-node (guaranteed safe)
5. **Going up:** Fix 4-nodes by splitting (color flip), restore LLRB properties
6. **LLRB translation:**
   - Move red left: ensure left path has red node (create temporary 4-node)
   - Move red right: ensure right path has red node
   - Fix-up on way back: rotate to eliminate right-leaning reds, split 4-nodes

Why 2-3 deletion is best for LLRB:
- Direct correspondence: LLRB is just 2-3 tree with specific encoding
- Fewer cases to handle than standard RB deletion
- More intuitive: work with 2-nodes and 3-nodes instead of complex color cases
- Guarantees we never delete from a 2-node (which would break tree)

Search: Same as BST, ignore colors. Time $O(\log n)$.

2-3 Trees

Properties:
- 2-node: 1 key, 2 children
- 3-node: 2 keys, 3 children
- All leaves at same level
- Perfect balance
- Height: $h = \lfloor \log_3 n \rfloor$ to $\lfloor \log_2 n \rfloor$

Relation to RB Trees:
- 2-node $\leftrightarrow$ black node
- 3-node $\leftrightarrow$ black node with red child
- LLRB enforces left-leaning representation
- Standard RB can have red on either side

Insertion: Time $O(\log n)$.
1. Search to find correct leaf position
2. Insert into leaf node
3. If creates 4-node (3 keys): split into two 2-nodes, promote middle key
4. Propagate splits up if parent also becomes 4-node
5. If root splits, tree height increases by 1

Deletion (THE KEY TO LLRB DELETION): Time $O(\log n)$.
1. **Case 1 - Delete from 3-node or 4-node leaf:** Simply remove key
2. **Case 2 - Delete from 2-node leaf:** Cannot directly remove (would be empty)
   - If sibling is 3-node: borrow key (transfer through parent)
   - If sibling is 2-node: merge with sibling and parent key to form 3-node
   - May propagate merge up to parent
3. **Case 3 - Delete from internal node:**
   - Replace with predecessor (max of left subtree) or successor
   - Recursively delete predecessor/successor from leaf
   - Handle as leaf deletion case
4. **Key invariant:** Never let a 2-node become empty during deletion
5. **Strategy going down:** Convert 2-nodes to 3-nodes or 4-nodes on path to deletion point

When to use: Theoretical understanding of balanced trees, basis for RB trees.

Graphs

Representation:
- *Adjacency List:* Array of lists. Space $O(V+E)$. Good for sparse graphs.
- *Adjacency Matrix:* $V \times V$ matrix. Space $O(V^2)$. Good for dense graphs, quick edge lookup $O(1)$, transitive closure, shortest paths (Floyd-Warshall).

BFS (Breadth-First Search): Time $O(V+E)$.
1. Start at source s, mark visited
2. Use queue: enqueue s
3. While queue not empty: dequeue v, enqueue all unvisited neighbors
4. Finds shortest path in unweighted graphs
5. Produces BFS tree with levels

DFS (Depth-First Search): Time $O(V+E)$.
1. Start at source, mark visited
2. Recursively visit unvisited neighbors
3. Use stack (explicit or recursion)
4. Produces DFS tree/forest
5. Edge classification: tree, back, forward, cross

When to use BFS: Shortest path (unweighted), level-order, min jumps.
When to use DFS: Topological sort, cycle detection, connectivity, SCCs.

Directed Graphs (Digraphs)

Strong Connectivity: Every vertex reachable from every other vertex.
Check Strong Connectivity: Time $O(V(V+E))$.
- Run DFS/BFS from each vertex v
- Check all V vertices visited in each traversal
- If all checks pass, graph is strongly connected

Topological Ordering (Kahn's Algorithm): Time $O(V+E)$. Only for DAGs.
1. Compute in-degree for all vertices
2. Add all 0 in-degree vertices to set S
3. While S not empty: remove vertex v, add to ordering, decrease in-degree of neighbors, add any that reach 0 to S
4. If all vertices processed: valid topological order. Else: cycle exists.

Transitive Closure: Find all pairs (u, v) where path exists from u to v.
Sedgewick & Wayne (Adjacency Matrix): Time $O(V(V+E))$.
- Initialize: Copy adjacency matrix, set diagonal to all 1s (every vertex reachable from itself)
- Run DFS from each vertex to find all reachable vertices
- $TC[i][j] = 1$ if path exists from i to j, else 0
- Diagonal always 1s because every vertex can reach itself

Goodrich & Tamassia (Adjacency Matrix):
- Initialize: Copy adjacency matrix, diagonal entries are 0s unless self-loop exists
- Only set $TC[i][i] = 1$ if explicit self-loop edge (i, i) exists in graph
- Otherwise same as Sedgewick & Wayne

Warshall's Algorithm: Dynamic programming on adjacency matrix. Time $O(V^3)$.
- Initialize: $TC^{(0)}[i][j] =$ adjacency matrix
- Triple nested loop: for $k = 1$ to V, for $i = 1$ to V, for $j = 1$ to V
- $TC^{(k)}[i][j] = TC^{(k-1)}[i][j] \vee (TC^{(k-1)}[i][k] \wedge TC^{(k-1)}[k][j])$
- Meaning: path $i \to j$ exists if already existed OR can go $i \to k \to j$
- Consider paths through intermediate vertices $\{1, \ldots, k\}$
- **Diagonal**: $TC[i][i] = 1$ if self-loop OR cycle containing vertex i

Floyd-Warshall (All-Pairs Shortest Path): Time $O(V^3)$. Allows negative edges.
Initialization:
- Set diagonal: $dist[i][i] = 0$ for all i (distance to self is 0)
- For each edge (i, j): $dist[i][j] = w(i, j)$ (direct edge weight)
- For non-adjacent vertices: $dist[i][j] = \infty$ (no direct path)

Algorithm (Triple Nested Loop):
1. For $k = 1$ to V: (consider vertex k as intermediate)
2. For $i = 1$ to V: (for each start vertex)
3. For $j = 1$ to V: (for each end vertex)
4. If $dist[i][j] > dist[i][k] + dist[k][j]$:
5. $dist[i][j] = dist[i][k] + dist[k][j]$
6. **Explanation:** For every start→end through intermediate k, check if route $i \to k \to j$ is shorter than direct $i \to j$

Key Points:
- After k iterations: $dist[i][j]$ = shortest path from i to j using vertices $\{1, \ldots, k\}$ as intermediates
- After all V iterations: $dist[i][j]$ = shortest path from i to j overall
- Detect negative cycles: if $dist[i][i] < 0$ for any i
- Can reconstruct paths by tracking predecessor matrix $prev[i][j]$

Strongly Connected Components (SCCs)

Kosaraju's Algorithm: Time $O(V + E)$.
1. **Phase 1:** Run DFS on G, push vertices to stack L in post-order (after visiting all neighbors)
2. **Phase 2:** Reverse all edges to get G^R
3. While stack L not empty: pop vertex v, if unvisited in G^R, run DFS from v in G^R. All visited vertices form one SCC

Why it works: Phase 1 orders by finish time. Phase 2 finds SCCs in reverse topological order of SCC DAG.
When to use: Find SCCs, simplify graph, analyze connectivity structure.

Shortest Path Algorithms

Dijkstra's Algorithm: Time $O((V + E) \log V)$ with min-heap. Non-negative edges only.
1. Initialize $dist[s] = 0$, $dist[v] = \infty$ for $v \neq s$
2. Add all vertices to priority queue (min-heap by distance)
3. While queue not empty: extract u with min $dist[u]$
4. For each neighbor v of u: if $dist[u] + w(u, v) < dist[v]$, update $dist[v]$ and $prev[v]$, decrease key in heap
5. **Greedy:** Always picks closest unvisited vertex

Restrictions: No negative edge weights. Will fail with negative edges.
When to use: Single-source shortest path, non-negative weights, GPS/routing.
Bellman-Ford Algorithm: Time $O(VE)$. Handles negative edges, detects negative cycles.
1. Initialize $dist[s] = 0$, $dist[v] = \infty$ for $v \neq s$
2. Repeat $V - 1$ times: for each edge (u, v), if $dist[u] + w(u, v) < dist[v]$, update $dist[v]$ and $prev[v]$ (relaxation)
3. Check for negative cycle: for each edge (u, v), if $dist[u] + w(u, v) < dist[v]$, negative cycle exists

Why $V - 1$ iterations: Longest simple path has $\leq V - 1$ edges. Each iteration finds shortest paths with one more edge.
Negative cycle detection: If relaxation possible after $V - 1$ iterations, cycle exists.
When to use: Negative edges present, detect negative cycles, smaller graphs.

Minimum Spanning Tree (MST)

Properties:
- Spanning tree: connects all vertices, acyclic, $|E| = |V| - 1$
- MST: spanning tree with minimum total edge weight
- For V vertices: exactly $V - 1$ edges in MST

Cut Property: For any cut $(S, V - S)$, min-weight edge crossing cut is in some MST.
Cycle Property: For any cycle, max-weight edge in cycle is not in any MST (if weights distinct).
Prim's Algorithm (Cut Property): Time $O((V + E) \log V)$ with min-heap.
1. Start with arbitrary vertex s, add to MST set S
2. Initialize priority queue with all edges from s
3. While $|S| < V$: extract min-weight edge (u, v) crossing cut $(S, V - S)$
4. Add v to S, add edge to MST
5. Add all edges from v to unvisited vertices to queue
6. **Greedy:** Grows single tree by adding min-weight edge to tree

When to use: Dense graphs, need to grow from specific vertex, network design.
Kruskal's Algorithm (Cycle Property): Time $O(E \log E)$ or $O(E \log V)$.
1. Sort all edges by weight
2. Initialize union-find with each vertex in own set
3. For each edge (u, v) in sorted order:
   - If u and v in different sets: add edge to MST, union sets
   - Else: skip edge (would create cycle)
4. Stop when $V - 1$ edges added
5. **Greedy:** Adds min-weight edge that doesn't create cycle

When to use: Sparse graphs, edges already sorted, clustering problems.
MST Uniqueness: MST is unique if all edge weights are distinct. With duplicate weights, multiple MSTs may exist.

Key Bounds & Formulas

Graph Handshaking Lemma: $\sum_{v \in V} \deg(v) = 2|E|$
Number of edges in graph: Undirected: $|E| \leq \frac{V(V-1)}{2}$; Directed: $|E| \leq V(V - 1)$
Euler Path: Exists iff graph connected and exactly 0 or 2 odd-degree vertices.
Euler Circuit: Exists iff graph connected and all vertices have even degree.
Hamiltonian Path/Cycle: No simple test exists (NP-complete problem).
Tree Properties: $|E| = |V| - 1$; unique path between any two vertices; removing any edge disconnects tree; adding any edge creates exactly one cycle.
Complete Graph K_n: $|E| = \frac{V(V-1)}{2}$ (undirected), $|E| = V(V - 1)$ (directed).
DAG Properties: Has topological ordering; no back edges in DFS; at least one source (0 in-degree) and one sink (0 out-degree).
Bipartite Graph: No odd-length cycles; 2-colorable; can check with BFS/DFS.
Graph Density: $d = \frac{2|E|}{|V|(|V|-1)}$ for undirected; sparse if $|E| = O(|V|)$, dense if $|E| = \Theta(|V|^2)$.
Binary Tree Properties:
- Max nodes at level i: 2^i (root at level 0)
- Total nodes in complete tree of height h: $2^{h+1} - 1$
- Leaves in full binary tree: $\frac{n+1}{2}$ where n = total nodes
- Height vs nodes: $h = \lfloor \log_2 n \rfloor$ (complete tree)

B-tree $t = 2$ (2-3-4 tree): Each node has 1-3 keys, 2-4 children.
Black-height relation: $2^{bh(x)} - 1 \leq n(x)$ where $n(x)$ is # nodes in subtree.
RB tree height: $h(n) \leq 2 \log_2(n + 1)$, so $n \geq 2^{h/2} - 1$.
Path relaxation: $dist[v] = \min(dist[v], dist[u] + w(u, v))$ (Dijkstra, Bellman-Ford).
Union-Find (with path compression + union by rank): Nearly $O(\alpha(n))$ per operation where α is inverse Ackermann (effectively constant).
DFS Properties: Discovery time $d[v]$ and finish time $f[v]$; $d[u] < d[v] < f[v] < f[u]$ means v is descendant of u.
Connected Components: Undirected graph: run DFS/BFS, count # of times started from unvisited vertex.

Algorithm Selection

Search: Small→BST; Large/disk→B-tree; Balance guarantee→RB. **Shortest Path:** Unweighted→BFS; Weighted non-neg→Dijkstra; Negative edges→Bellman-Ford; All pairs→Floyd-Warshall. **Connectivity:** Check connected→DFS/BFS; Find SCCs→Kosaraju; Find path→DFS/BFS. **MST:** Dense→Prim's; Sparse→Kruskal's; Edges sorted→Kruskal's.

Common Exam Patterns

BST Deletion: Leaf→remove; 1 child→replace; 2 children→use successor/predecessor. **B-tree Ops:** Split at $2t - 1$ keys; merge/borrow at $< t - 1$. **RB Insert:** Red insert, uncle color→case. **LLRB:** Fix right-red→left-left-red→both-red bottom-up. **Graph:** BFS=queue (level), DFS=stack (depth). **Topo Sort:** Remove 0 in-degree iteratively. **Kosaraju:** Reverse edges phase 2, pop from stack. **Dijkstra:** Pick min unvisited; fails with negative. **Bellman-Ford:** Relax all edges $V - 1$ times; extra iteration=negative cycle. **MST Check:** Cut property (min edge crossing in MST) or cycle property (max in cycle not in MST). **MST Update:** Weight↓ in MST=no change; weight↑ in MST=remove, reconnect with min crossing; edge not in MST=add (cycle), remove max in cycle if new lighter.