01. Data Structures & Design and Analysis of Algorithms (DAA) - Textbook Reference

> **Module Focus**: Exhaustive, textbook-level reference on Linear/Non-Linear Data Structures, AVL Tree Rotations, Binary Heaps & Build-Heap Derivations, Graph Algorithms, Sorting Comparison Matrix, and Master Theorem Derivations.

---

1. Linear Data Structures

1.1 Memory Allocation & Layout Mechanics

Feature / Concept	Array	Linked List (Singly / Doubly)
Memory Allocation	Static / Contiguous block on Stack or Heap	Dynamic / Non-contiguous heap nodes
Cache Locality	**High** (Spatial locality enables CPU prefetching)	**Low** (Pointer chasing leads to frequent cache misses)
Access Pattern	Direct indexed access via base address + offset: $ \text{Addr}(i) = B + i \times \text{size}$	Sequential pointer traversal ($O(n)$ time)
Overhead	Zero per-element pointer overhead	1 pointer per node (Singly: 8 bytes) or 2 pointers (Doubly: 16 bytes)

1.2 Array vs Linked List Variations

```
Singly Linked List:   [Head] -> [Data | Next] -> [Data | Next] -> NULL

Doubly Linked List:   NULL <- [Prev | Data | Next] <-> [Prev | Data | Next] -> NULL

Circular List:        [Head] -> [Data | Next] -> [Data | Next] --+
                        ^                                       |
                        +---------------------------------------+
```

Operation	Array (Fixed/ Dynamic)	Singly Linked List	Doubly Linked List	Circular Linked List	Skip List
Random Access	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$ avg

Operation	Array (Fixed/ Dynamic)	Singly Linked List	Doubly Linked List	Circular Linked List	Skip List
Insert at Head	$O(n)$	$O(1)$	$O(1)$	$O(1)$ (with Tail ptr)	$O(\log n)$ avg
Insert at Tail	$O(1)$ amortized	$O(1)$ (with Tail ptr)	$O(1)$ (with Tail ptr)	$O(1)$ (with Tail ptr)	$O(\log n)$ avg
Delete Given Node	$O(n)$	$O(n)$ (requires prev node)	$O(1)$ (direct node reference)	$O(n)$	$O(\log n)$ avg
Search (Unsorted)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Search (Sorted)	$O(\log n)$ (Binary Search)	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$ avg

---

1.3 Stack Mechanics, Applications & Conversions

A **Stack** is a LIFO (Last-In, First-Out) linear structure supporting `push(x)` ($O(1)$), `pop()` ($O(1)$), and `peek()` ($O(1)$).

1. Infix to Postfix Conversion (Shunting-Yard Algorithm)

Given operators with precedence ($+,- \to 1; \times,/ \to 2; \text{^} \to 3$) and left-to-right associativity (^ is right-to-left): - **Operands**: Immediately append to output. - **Operator op_1**: While top of stack has operator op_2 with greater precedence (or equal precedence and op_1 is left-associative), pop op_2 to output. Push op_1. - **Left Parenthesis** `(` : Push onto stack. - **Right Parenthesis** `)` : Pop operators to output until `(` is encountered. Pop `(`.

2. Monotonic Stack

A stack maintaining elements in monotonically increasing or decreasing order. - **Application**: Next Greater Element (NGE), Largest Rectangle in Histogram ($O(n)$ time).

1.4 Queue Mechanics & Variants

A **Queue** is a FIFO (First-In, First-Out) structure supporting `enqueue(x)` ($O(1)$) and `dequeue()` ($O(1)$).

1. Circular Queue Mechanics

Prevents array drift where `front` advances leaving unusable space. - Modulo index updates: $$\text{rear} = (\text{rear} + 1) \pmod N$$ $$\text{front} = (\text{front} + 1) \pmod N$$ - **Full Condition**: $(\text{rear} + 1) \pmod N == \text{front}$ - **Empty Condition**: $\text{front} == -1$ (or $\text{count} == 0$)

2. Double-Ended Queue (Deque)

Allows insertion and deletion at both `front` and `rear` in $O(1)$ time. Used in sliding window maximum algorithms ($O(n)$ overall).

---

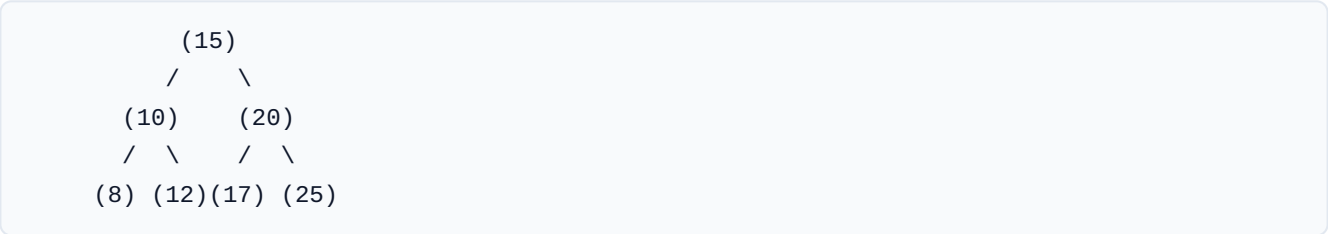
2. Non-Linear Data Structures: Trees & Heaps

2.1 Tree Terminology & Structural Types

- **Full/Strict Binary Tree**: Every node has either 0 or 2 children.
- **Complete Binary Tree**: All levels are completely filled except possibly the last, which is filled from left to right.
- **Perfect Binary Tree**: All internal nodes have 2 children, and all leaf nodes are at the same level. Total nodes $N = 2^{h+1} - 1$.
- **Balanced Binary Tree**: Height is bounded by $O(\log n)$, where height $h = \text{max depth}$.
- **Degenerate Tree**: Every internal node has only 1 child (effectively a linked list, height $h = n-1$).

2.2 Binary Search Tree (BST) Mechanics

A **BST** satisfies: $\text{Key}(\text{Left Subtree}) < \text{Key}(\text{Node}) < \text{Key}(\text{Right Subtree})$.



Traversal Mechanics:

1. **In-Order** (Left, Root, Right): $8, 10, 12, 15, 17, 20, 25 \implies$ Yields strictly ascending sorted order.
2. **Pre-Order** (Root, Left, Right): $15, 10, 8, 12, 20, 17, 25 \implies$ Serializes tree structure.
3. **Post-Order** (Left, Right, Root): $8, 12, 10, 17, 25, 20, 15 \implies$ Bottom-up evaluation / memory deletion.
4. **Morris In-Order Traversal**: Uses threaded temporary links (`right` child of predecessor points to current node) achieving **$O(n)$ time and $O(1)$ auxiliary space**.

BST Deletion Algorithm (3 Cases):

1. **Node is a Leaf**: Remove node directly.
2. **Node has 1 Child**: Replace node with its single child link.
3. **Node has 2 Children**:
4. Find **In-Order Successor** (smallest node in right subtree) or **In-Order Predecessor** (largest in left subtree).
5. Copy key of successor to target node.
6. Recursively delete the successor (which falls into Case 1 or Case 2).

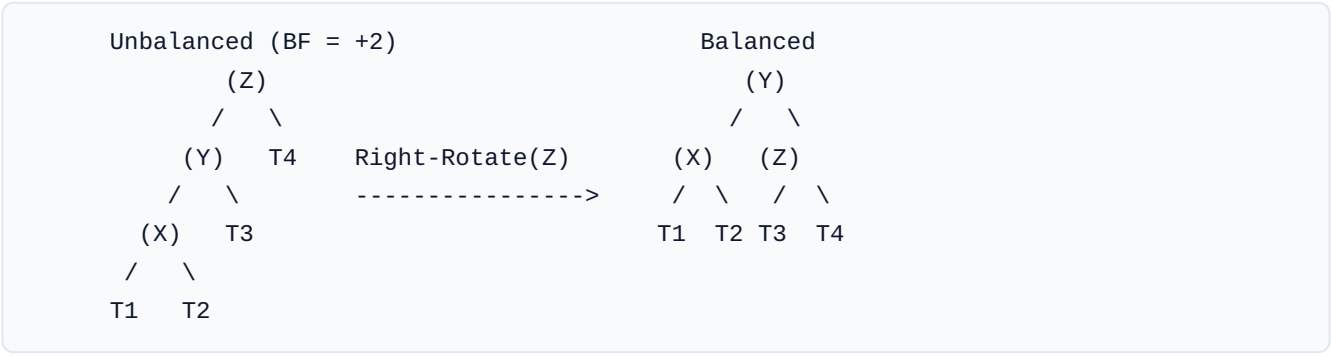
2.3 Height-Balanced Trees: AVL Trees & Rotation Mechanics

An **AVL Tree** enforces the height-balance property for every node: $$\text{Balance Factor } BF(\text{Node}) = \text{Height}(\text{Left Subtree}) - \text{Height}(\text{Right Subtree}) \in \{-1, 0, +1\}$$

When an insertion or deletion causes $|BF| > 1$, re-balancing is performed using **Rotations**.

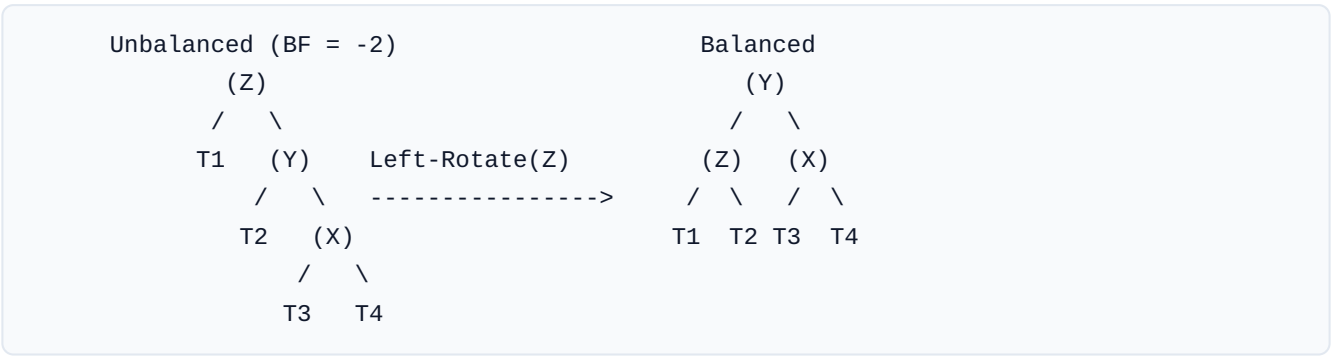
1. Left-Left (LL) Case $\to$ Single Right Rotation

Occurs when an insertion is in the **Left child of the Left subtree**.



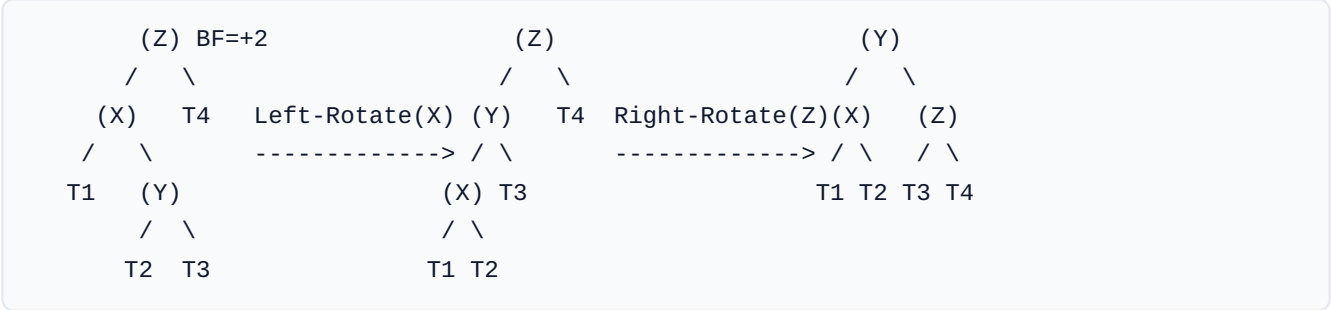
2. Right-Right (RR) Case $\to$ Single Left Rotation

Occurs when an insertion is in the **Right child of the Right subtree**.



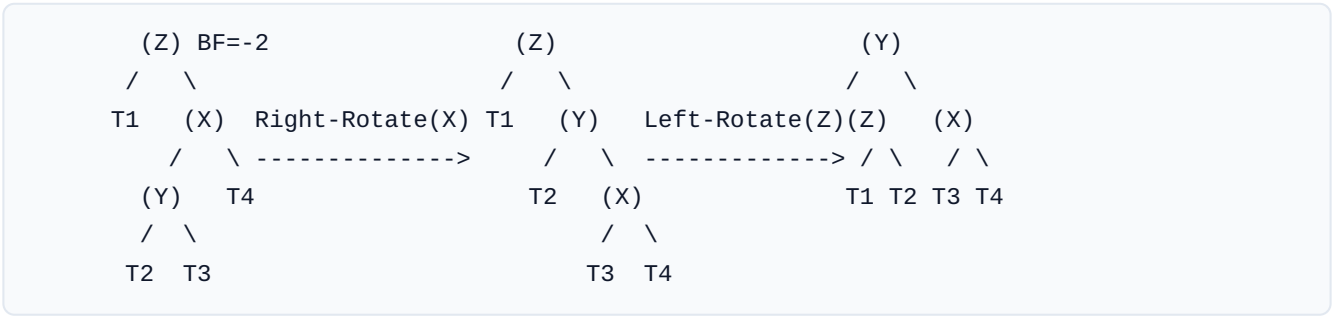
3. Left-Right (LR) Case $\to$ Double Rotation (Left on Left Child, Right on Root)

Occurs when an insertion is in the **Right child of the Left subtree**.



4. Right-Left (RL) Case $\to$ Double Rotation (Right on Right Child, Left on Root)

Occurs when an insertion is in the **Left child of the Right subtree**.



Feature	AVL Tree	Red-Black Tree
Height Bound	Strict: $h \le 1.44 \log_2 n$	Relaxed: $h \le 2 \log_2(n+1)$
Lookup Speed	Faster (due to flatter height)	Slightly slower
Insertion / Deletion	More rotations required to rebalance	Fewer rotations ($O(1)$ rotations for insert/delete)
Use Case	Read-heavy applications	General-purpose map/set implementations (e.g., C++ `std::map`, Java `TreeMap`)

2.4 Binary Heaps & Build-Heap $O(n)$ Proof

A **Binary Heap** is a complete binary tree backed by an array: - Parent Index: $\lfloor (i - 1) / 2 \rfloor$ - Left Child: $2i + 1$ - Right Child: $2i + 2$

Heap Operations:

- `insert(x)`: Append to end, `percolate_up` (heapify up) $\to O(\log n)$.
- `extract_min()` / `extract_max()`: Swap root with last element, pop last, `percolate_down` (heapify down) $\to O(\log n)$.

Mathematical Proof: Build-Heap Complexity is $O(n)$

`Build-Heap` operates by calling `percolate_down` on all non-leaf nodes starting from index $\lfloor n/2 \rfloor - 1$ down to `0`.

- Height of node at level h: h (where leaves are at height $h = 0$).
- Maximum number of nodes at height h in a tree of size n: $\lceil \frac{n}{2^{h+1}} \rceil$.
- Maximum work done by `percolate_down` at height h: $O(h) = C \cdot h$.

Total time complexity summation: $$T(n) = \sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot C \cdot h \le C \cdot n \sum_{h=0}^{\infty} \frac{h}{2^{h+1}} = \frac{C \cdot n}{2} \sum_{h=0}^{\infty} \frac{h}{2^h}$$

Let $S = \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{0}{1} + \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \dots$

Compute $\frac{1}{2}S$: $$\frac{1}{2}S = \frac{0}{2} + \frac{1}{4} + \frac{2}{8} + \frac{3}{16} + \dots$$

Subtract $\frac{1}{2}S$ from S: $$S - \frac{1}{2}S = \left(0 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots \right)$$ $$\frac{1}{2}S = \sum_{k=1}^{\infty} \frac{1}{2^k} = \frac{1/2}{1 - 1/2} = 1$$ $$S = 2$$

Substituting $S = 2$ back into total work equation: $$T(n) \le \frac{C \cdot n}{2} \times 2 = C \cdot n = O(n)$$

$\blacksquare$ **Build-Heap runs in linear time $O(n)$.**

---

3. Graph Algorithms

3.1 Graph Representations Comparison

Representation	Space Complexity	Edge Existence Check (u, v)	Edge Enumeration for u	Ideal For
Adjacency Matrix	$O(V^2)$	$O(1)$	$O(V)$	Dense Graphs ($E \approx V^2$)
Adjacency List	$O(V + E)$	$O(\text{deg}(u))$	$O(\text{deg}(u))$	Sparse Graphs ($E \ll V^2$)

3.2 Graph Traversals: BFS vs DFS

```
Graph:
    (0) --- (1)
     |       |
    (2) --- (3)
```

- **Breadth-First Search (BFS)**: Uses FIFO Queue. Finds **shortest path in unweighted graphs**. Time: $O(V + E)$, Space: $O(V)$.
- **Depth-First Search (DFS)**: Uses Call Stack / LIFO. Used for topological sort, cycle detection, strongly connected components. Time: $O(V + E)$, Space: $O(V)$.

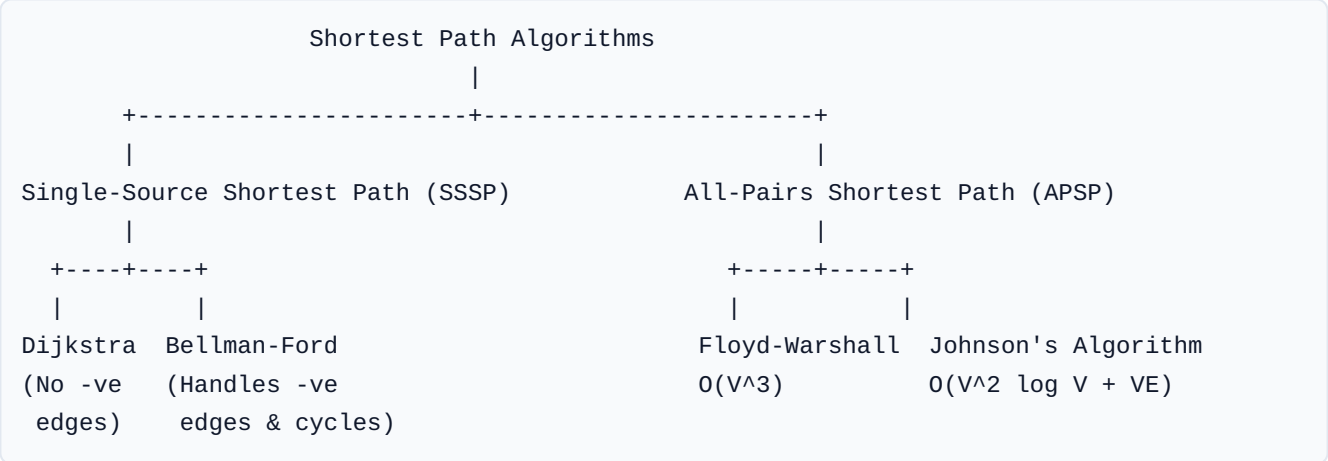
3.3 Topological Sorting (DAGs Only)

Linear ordering of vertices such that for every directed edge $u \to v$, vertex u comes before v.

1. **Kahn's Algorithm (In-Degree BFS)**:
2. Compute in-degree for all vertices.
3. Push all vertices with $\text{in-degree} = 0$ into a queue.
4. Dequeue u, append to order, decrement in-degree of all neighbors v. If in-degree of v becomes 0, push v.

5. If processed count $< V$, graph contains a **cycle**.
6. **DFS Post-Order Reversal**:
7. Perform DFS traversal. Push vertex to a stack after visiting all outgoing edges.
8. Pop stack to obtain topological order.

3.4 Shortest Path Algorithms Summary



1. Dijkstra's Algorithm (Greedy)

- **Condition**: Non-negative edge weights ($w(u,v) \ge 0$).
- **Data Structure**: Min-Heap Priority Queue.
- **Time Complexity**: $O((V + E) \log V)$ with binary heap; $O(E + V \log V)$ with Fibonacci heap.
- **Relaxation Step**: If $dist[u] + w(u,v) < dist[v]$, then update $dist[v] = dist[u] + w(u,v)$.

2. Bellman-Ford Algorithm (Dynamic Programming)

- **Condition**: Handles negative edge weights; detects negative cycles.
- **Mechanism**: Relax all E edges $V-1$ times.
- **Negative Cycle Detection**: Perform an V-th pass. If any $dist[v]$ can still be reduced, a **negative weight cycle exists**.
- **Time Complexity**: $O(V \cdot E)$.

3. Floyd-Warshall Algorithm (Dynamic Programming)

- **All-Pairs Shortest Path (APSP)**.
- **State Recurrence**: $dp[k][i][j] = \min(dp[k-1][i][j], dp[k-1][i][k] + dp[k-1][k][j])$.
- **Space Optimization**: $O(V^2)$ 2D matrix updated in-place.
- **Time Complexity**: $O(V^3)$.

3.5 Minimum Spanning Trees (MST)

For a connected, undirected, weighted graph, an MST connects all V vertices with $V-1$ edges minimizing total edge weight.

Algorithm	Paradigm	Main Data Structure	Time Complexity	Ideal For
Kruskal's	Greedy (Edge-based)	Disjoint Set Union (DSU) with Path Compression & Rank	$O(E \log E)$ or $O(E \log V)$	Sparse Graphs
Prim's	Greedy (Vertex-based)	Min-Heap Priority Queue	$O(E \log V)$	Dense Graphs

Disjoint Set Union (DSU) Operations:

- `find(i)` : Uses **Path Compression** $\to$ flattens tree structure ($O(\alpha(n))$ amortized time, where α is Inverse Ackermann function).
- `union(i, j)` : Uses **Union by Rank / Size** $\to$ attaches smaller tree under root of larger tree.

3.6 Strongly Connected Components (SCC) - Kosaraju's Algorithm

A subgraph is strongly connected if there is a directed path between every pair of vertices.

Kosaraju's 2-Pass DFS Algorithm ($O(V + E)$):

1. Order vertices by finishing time using full DFS on original graph G (push to stack).
2. Compute transposed graph G^T (reverse all edge directions).
3. Pop vertices from stack; if unvisited in G^T, run DFS from it. Each DFS tree forms a distinct **Strongly Connected Component**.

---

4. Algorithm Analysis & Asymptotic Bounds

4.1 Formal Asymptotic Notations

Let $f(n)$ and $g(n)$ be non-negative functions:

1. **Big-O (O) - Asymptotic Upper Bound**: $$f(n) = O(g(n)) \iff \exists c > 0, n_0 > 0 \text{ s.t. } 0 \le f(n) \le c \cdot g(n), \, \forall n \ge n_0$$
2. **Big-Omega (Ω) - Asymptotic Lower Bound**: $$f(n) = \Omega(g(n)) \iff \exists c > 0, n_0 > 0 \text{ s.t. } 0 \le c \cdot g(n) \le f(n), \, \forall n \ge n_0$$
3. **Big-Theta (Θ) - Tight Bound**: $$f(n) = \Theta(g(n)) \iff f(n) = O(g(n)) \text{ AND } f(n) = \Omega(g(n))$$
4. **Little-o (o) - Strict Upper Bound**: $$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$
5. **Little-omega (ω) - Strict Lower Bound**: $$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

4.2 Master Theorem for Divide-and-Conquer Recurrences

For recurrences of the form: $$T(n) = a T\left(\frac{n}{b}\right) + f(n) \quad \text{where } a \ge 1, b > 1$$

Full Derivation via Recursion Tree Method:

- **Level 0 (Root)**: 1 node of size n, work $= f(n)$
- **Level 1**: a nodes of size n/b, total work $= a \cdot f(n/b)$
- **Level j**: a^j nodes of size n/b^j, total work $= a^j \cdot f(n/b^j)$
- **Tree Depth**: $h = \log_b n$
- **Total Leaves**: $a^{\log_b n} = n^{\log_b a}$
- **Work at Leaves**: $\Theta(n^{\log_b a})$

Total summation: $$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f\left(\frac{n}{b^j}\right)$$

```
Level 0:                f(n)                        = f(n)
                       /    \
Level 1:           f(n/b)     f(n/b)                = a * f(n/b)
                  /   \       /   \
Level 2:       f(n/b²) ...  f(n/b²) ...             = a² * f(n/b²)
...
Level log_b(n):  (1)  (1)  (1)  (1) ...             = 0(n^(log_b a))
```

The 3 Standard Cases:

Case 1: Leaf-Dominated Work

If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$: $$T(n) = \Theta(n^{\log_b a})$$ *(The cost of solving subproblems at the leaves dominates).*

Case 2: Balanced Work Per Level

If $f(n) = \Theta(n^{\log_b a} \log^k n)$ for $k \ge 0$: $$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$$ *(Each level contributes equal asymptotic work).*

Case 3: Root-Dominated Work

If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, AND $f(n)$ satisfies the **Regularity Condition**: $$a f\left(\frac{n}{b}\right) \le c f(n) \quad \text{for some } c < 1 \text{ and all sufficiently large } n$$ $ $T(n) = \Theta(f(n))$$ *(The cost at the root dominates).*

Subtract-and-Conquer Recurrence

For recurrences of the form: $$T(n) = a T(n - b) + f(n) \quad (a > 0, b > 0)$$

- If $a < 1$: $T(n) = O(f(n))$
- If $a = 1$: $T(n) = O(n \cdot f(n))$
- If $a > 1$: $T(n) = O(a^{n/b} \cdot f(n))$

4.3 Sorting Algorithms Exhaustive Comparison Matrix

Algorithm	Best Time	Average Time	Worst Time	Auxiliary Space	Stable?	In-Place?	Paradigm	Key Characteristics / Notes
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	**Yes**	**Yes**	Swapping	$O(n)$ best case with flag check.
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	**No**	**Yes**	Selection	Minimizes number of swaps ($O(n)$ swaps total).
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	**Yes**	**Yes**	Incremental	Efficient for small n or nearly sorted arrays.
Shell Sort	$O(n \log n)$	$O(n^{4/3})$	$O(n^2)$	$O(1)$	**No**	**Yes**	Diminishing Increment	Generalization of insertion sort using gaps.
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	**Yes**	**No**	Divide & Conquer	Guaranteed $O(n \log n)$; excellent for linked lists.
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$ call stack	**No**	**Yes**	Divide & Conquer (Pivot)	Worst case occurs on sorted array with bad pivot.
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	**No**	**Yes**	Selection (Max-Heap)	In-place selection sort using binary heap.
Counting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(n + k)$	**Yes**	**No**	Non-Comparison (Keys in $[0, k]$)	Time/space depends on key range k.
Radix Sort	$O(d \cdot (n + k))$	$O(d \cdot (n + k))$	$O(d \cdot (n + k))$	$O(n + k)$	**Yes**	**No**	Non-Comparison (Digit-by-digit)	Uses stable counting sort per digit d.
Bucket Sort	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n)$	**Yes**	**No**	Distribution	Assumes uniform distribution over $[0, 1)$.