DATA STRUCTURES & ALGORITHM

---

1. Array

- **Type**: Linear
- **Description**: A collection of elements stored at contiguous memory locations.
- **Complexity**:
  - Access: O(1)O(1)O(1)
  - Search: O(n)O(n)O(n)
  - Insertion: O(n)O(n)O(n) (if resizing is needed)
  - Deletion: O(n)O(n)O(n)
- **Example**: Array of integers `[1, 2, 3, 4]`.
- **Use Cases**: Useful for storing multiple values of the same type. Often used when the size of data is known and doesn't change frequently.
- **Types**: Static array (fixed size), Dynamic array (resizable)
- **What it is**: A collection of items stored at contiguous memory locations.
- **How it works**: Arrays store elements in a single, fixed-size block of memory, which makes it easy to access any item using an index (like `array[0]` for the first item).
- **Example**: Let's say we have an array of numbers `[10, 20, 30, 40]`. We can access `20` directly using `array[1]`.
- **Use**: Good for storing fixed-size data. However, inserting or deleting an element in the middle of an array requires shifting elements, which is slow.

---

2. Linked List

- **Type**: Linear
- **Description**: A collection of nodes where each node contains data and a reference to the next node.
- **Complexity**:
  - Access: O(n)O(n)O(n)
  - Search: O(n)O(n)O(n)
  - Insertion: O(1)O(1)O(1) (at the head)
  - Deletion: O(1)O(1)O(1) (if pointer to node is available)
- **Example**: Singly Linked List (1 → 2 → 3)
- **Use Cases**: Useful in scenarios where frequent insertion and deletion are required.
- **Types**: Singly linked list, Doubly linked list, Circular linked list
- **What it is**: A collection of nodes, where each node contains data and a pointer (or link) to the next node.
- **How it works**: Unlike arrays, linked lists don't store elements in contiguous memory. Each node knows where the next one is, allowing the list to be flexible with memory.
- **Example**: In a linked list `1 → 2 → 3`, the node with data `1` has a pointer to the node `2`, and `2` points to `3`.

- **Use**: Useful when frequent insertions and deletions are needed because there's no need to shift elements. Drawback: Accessing elements is slower as you must traverse the list from the beginning.

Singly Linked List

- **What it is**: A linked list where each node points to the next node only.
- **How it works**: Each node has two parts: data and a pointer to the next node. Traversal happens from the head to the end.
- **Example**: For $1 \rightarrow 2 \rightarrow 3$, each node points only to the next node.
- **Use**: Simple structure, useful for stacks or when only forward traversal is required.

Doubly Linked List

- **What it is**: A linked list where each node points to both the next and the previous nodes.
- **How it works**: Each node has three parts: data, a pointer to the next node, and a pointer to the previous node, allowing traversal in both directions.
- **Example**: For $1 \rightleftarrows 2 \rightleftarrows 3$, 2 has pointers to both 1 and 3.
- **Use**: Useful for bidirectional traversal, like in navigation systems (back and forward buttons) and undo/redo functionality in applications.

Circular Linked List

- **What it is**: A linked list where the last node points back to the head, creating a circular structure.
- **How it works**: The last node's `next` pointer connects to the first node, and in the doubly circular version, the first node's `previous` pointer connects to the last node.
- **Example**: $1 \rightarrow 2 \rightarrow 3 \rightarrow 1$ forms a loop.
- **Use**: Useful in scenarios like round-robin scheduling, where we cycle through elements repeatedly.

---

3. Stack

- **Type**: Linear
- **Description**: A LIFO (Last In First Out) data structure.
- **Complexity**:
  - Push: O(1)O(1)O(1)
  - Pop: O(1)O(1)O(1)
  - Peek: O(1)O(1)O(1)
- **Example**: Stack of plates where the last plate added is the first one removed.
- **Use Cases**: Used for function call management, undo operations in editors, and parsing expressions.
- **Types**: Array-based stack, Linked list-based stack
- **What it is**: A LIFO (Last In, First Out) structure, where the last element added is the first one removed.

- **How it works**: Stack operations are limited to one end, the "top." The main operations are **push** (add to the top) and **pop** (remove from the top).
- **Example**: Imagine a stack of books; you can only take the book on top and add new books on top. So if you have books `[A, B, C]` (top is C), you push `D`, and the stack becomes `[A, B, C, D]`.
- **Use**: Used for function calls (last function called is the first to complete), undo mechanisms in text editors, and expression evaluation.

Simple Stack (Basic Stack)

- **Description**: The traditional or basic stack follows the Last-In-First-Out (LIFO) principle, meaning the last element added is the first to be removed.
- **Operations**: Basic operations include:
  - `push` (add an item to the top),
  - `pop` (remove the item from the top),
  - `peek` (view the item on the top without removing it).
- **Usage**: Used in function call management, expression evaluation, parsing, and undo operations in software.

Double-Ended Stack (Deque or Double-Ended Queue)

- **Description**: A more versatile stack that allows `push` and `pop` operations on both ends of the stack.
- **Operations**: Along with basic `push` and `pop`, it includes:
  - `push_front` (insert at the front),
  - `pop_front` (remove from the front),
  - `push_back` and `pop_back`.
- **Usage**: Suitable for applications needing flexibility in adding/removing elements from both ends, such as in memory management, scheduling, and deque-based algorithms.

---

4. Queue

- **Type**: Linear
- **Description**: A FIFO (First In First Out) data structure.
- **Complexity**:
  - Enqueue: O(1)O(1)O(1)
  - Dequeue: O(1)O(1)O(1)
- **Example**: Waiting line at a service center where the first person in line is the first to be served.
- **Use Cases**: Task scheduling, breadth-first search in graphs, managing resources.
- **Types**: Circular Queue, Priority Queue, Deque (Double-ended queue)
- **What it is**: A FIFO (First In, First Out) structure, where the first element added is the first one removed.

- **How it works**: Elements are added at the "rear" and removed from the "front."
- **Example**: Think of a line at a ticket counter. The first person in line is the first to get served. If a queue has `[A, B, C]` (A is in the front), when `D` is enqueued, it becomes `[A, B, C, D]`. When `A` is dequeued, the queue is `[B, C, D]`.
- **Use**: Task scheduling, handling requests in servers, and breadth-first search in graphs.

Circular Queue

- **What it is**: A queue where the last position is connected back to the first position to make a circle.
- **How it works**: When the queue's end is reached, it wraps around to the beginning if there's space. This avoids wasting space that a traditional queue might need to resize.
- **Example**: Imagine a circular queue of size 5 with elements `[10, 20, 30, 40, 50]`. After dequeuing two elements, the front moves to `30`, and new elements can be added to the first two positions.
- **Use**: Useful in scenarios with fixed-size buffers, like CPU scheduling or in network data buffers.

Priority Queue

- **What it is**: A queue where each element has a priority. Elements with higher priority are dequeued before elements with lower priority.
- **How it works**: Items are enqueued with priority. When dequeuing, the item with the highest priority (or lowest number, if smallest value has highest priority) is removed first.
- **Example**: Hospital ER queue where critical patients are attended to before others, regardless of arrival order.
- **Use**: Task scheduling, Dijkstra's shortest path algorithm, and event-driven simulations.

Double-Ended Queue (Deque)

- **What it is**: A queue where elements can be added or removed from both ends.
- **How it works**: Supports `enqueueFront`, `enqueueRear`, `dequeueFront`, and `dequeueRear` operations, providing more flexibility than a traditional queue.
- **Example**: `[10, 20, 30, 40]` allows inserting or removing items from either end, so you could remove `10` from the front or `40` from the back.
- **Use**: Useful for data structures that need fast access and flexibility from both ends, like implementing a browser history, sliding window algorithms, and palindrome checkers.

---

5. Hash Table

- **Type**: Non-linear

- **Description**: Stores key-value pairs with fast lookups using a hash function.
- **Complexity**:
    - Insert: O(1)O(1)O(1) (average case)
    - Delete: O(1)O(1)O(1) (average case)
    - Search: O(1)O(1)O(1) (average case)
- **Example**: Dictionary in Python, where `key: value` pairs are stored.
- **Use Cases**: Implementing dictionaries, caching, and lookup tables.
- **Types**: Separate chaining, open addressing
- **What it is**: A structure that maps keys to values for efficient lookup using a hash function.
- **How it works**: Each key is converted to a hash code, which maps to an index where the value is stored. If two keys hash to the same index, a technique like chaining or open addressing resolves the collision.
- **Example**: For a phone book where `{"John": 123456, "Alice": 789101}`, "John" is the key and `123456` is the value.
- **Use**: Fast data lookup (like dictionaries in Python). Good for caching, database indexing, and lookups. Drawback: Hash collisions can make lookups slower.

Open Addressing

- **Description**: All elements are stored in the hash table itself. When a collision occurs, it tries to find the next available slot using a probing technique (e.g., linear probing, quadratic probing, or double hashing).
- **Usage**: Used in situations where memory is a concern, and you want to avoid the overhead of linked structures (e.g., C++'s `std::unordered_map`).

Separate Chaining

- **Description**: Each bucket in the hash table stores a linked list (or another data structure) to handle collisions. Multiple elements that hash to the same index are stored in the list.
- **Usage**: Commonly used when handling a large number of collisions, or when the size of the table is difficult to predict (e.g., Python's `dict`).

---

6. Tree

- **Type**: Non-linear
- **Description**: A hierarchical structure with a root node, branches, and leaves.
- **Complexity**:
    - Search: O(logn)O(\log n)O(logn) for balanced trees, O(n)O(n)O(n) for unbalanced trees
    - Insertion: O(logn)O(\log n)O(logn) for balanced trees
- **Example**: Binary Search Tree (BST)
- **Use Cases**: Used in hierarchical data representation like file systems, XML/HTML parsing, and databases.

- **Types**: Binary tree, AVL tree, Red-Black tree, B-Tree, Trie
- **What it is**: A hierarchical structure with a root node, branches, and leaves.
- **How it works**: Trees consist of nodes connected by edges. Each node can have children, forming a tree-like structure. Common operations include insertion, deletion, and traversal.

Example: A binary tree where each node has at most two children:

```
   10
  /  \
 5    20
```

- Here, `10` is the root, `5` is the left child, and `20` is the right child.
- **Use**: Organizing hierarchical data (like file systems), databases (like B-trees), and searching (like binary search trees).

Binary Search Tree (BST)

- **Description**: A binary tree where each node has at most two children. The left child contains a value less than the parent node, and the right child contains a value greater than the parent node.
- **Operations**:
  - **Search**: O(log n) on average, O(n) in the worst case.
  - **Insertion**: O(log n) on average.
  - **Deletion**: O(log n) on average.
- **Usage**: Common in applications where efficient search, insert, and delete operations are required, like in databases and file systems.

---

7. Graph

- **Type**: Non-linear
- **Description**: A set of vertices (nodes) connected by edges.
- **Complexity**:
  - Add Vertex: O(1)O(1)O(1)
  - Add Edge: O(1)O(1)O(1)
  - Search: O(V+E)O(V + E)O(V+E) for BFS/DFS
- **Example**: Social network connections, where each user is a node connected by relationships.
- **Use Cases**: Networking, social connections, route optimization, recommendation systems.
- **Types**: Directed, Undirected, Weighted, Unweighted, Cyclic, Acyclic
- **What it is**: A set of nodes (vertices) connected by edges.
- **How it works**: Each node can connect to multiple other nodes, forming a network. Graphs can be **directed** (edges have directions) or **undirected** (no directions).
- **Example**: A social network where each person is a node, and friendships are edges connecting nodes.

- **Use**: Used in networks (like social media, web pages linking to each other), and finding paths between locations (like Google Maps).

Directed Graph (Digraph)

- **Description**: A graph where edges have a direction, represented as ordered pairs (u, v).
- **Usage**: Common for representing one-way relationships, such as in social media followings, web links, and network flows.

Undirected Graph

- **Description**: A graph where edges have no direction, represented as unordered pairs (u, v).
- **Usage**: Used for representing bidirectional relationships, such as in social networks (friendship connections) and road networks.

---

8. Heap

- **Type**: Non-linear
- **Description**: A complete binary tree that maintains a specific order property.
- **Complexity**:
    - Insert: O(logn)O(\log n)O(logn)
    - Delete Min/Max: O(logn)O(\log n)O(logn)
- **Example**: Min-Heap (parent node is smaller than child nodes)
- **Use Cases**: Priority Queue implementations, scheduling algorithms, and graph algorithms.
- **Types**: Min-Heap, Max-Heap, Fibonacci Heap
- **What it is**: A complete binary tree that maintains a specific order, where each parent node is greater (Max-Heap) or smaller (Min-Heap) than its children.
- **How it works**: Heaps are built to efficiently retrieve the minimum or maximum element. In a Min-Heap, the smallest element is at the root. Insertion and deletion adjust the heap to maintain the order property.
- **Example**: In a Min-Heap with `[3, 10, 15]`, `3` is the minimum and root. Adding `1` would restructure the heap to `[1, 3, 15, 10]`.
- **Use**: Priority queues (task scheduling), finding the smallest/largest element quickly, and certain graph algorithms.

---

9. Sorting Algorithms

- **Type**: Algorithms (not data structures)
- **Description**: Algorithms to arrange elements in a specific order.
- **Complexity**: Varies based on algorithm
- **Examples**:

    - **Bubble Sort**: O(n2)O(n^2)O(n2), simple but slow
    - **Quick Sort**: O(nlogn)O(n \log n)O(nlogn) average case
    - **Merge Sort**: O(nlogn)O(n \log n)O(nlogn) (stable)
    - **Insertion Sort**: O(n2)O(n^2)O(n2) but good for small or nearly sorted arrays
- **Use Cases**: Data organization, database query optimization, analytics.
- **Types**: Stable (e.g., Merge Sort), Unstable (e.g., Quick Sort)
- **What it is**: A simple sorting algorithm that repeatedly swaps adjacent elements if they are in the wrong order.
- **How it works**: It "bubbles" the largest unsorted element to the end of the list in each pass. This is done until the list is sorted.
- **Example**: For `[5, 3, 8, 1]`, bubble sort compares and swaps:
    - `[3, 5, 8, 1] → [3, 5, 1, 8] → [3, 1, 5, 8] → [1, 3, 5, 8]`
- **Use**: Educational purposes and small lists. It's inefficient for large lists with a complexity of O(n2)O(n^2)O(n2).

Bubble Sort

a simple sorting algorithm that repeatedly steps through a list, compares adjacent items, and swaps them if they are in the wrong order.

How it works:
Start at the beginning of the list and compare each pair of adjacent elements. Swap them if the first one is greater than the second. Continue until no more swaps are needed.

Example:
For [5, 1, 4, 2, 8], compare 5 and 1 → swap → [1, 5, 4, 2, 8].
Then compare 5 and 4 → swap → [1, 4, 5, 2, 8].
Compare 5 and 2 → swap → [1, 4, 2, 5, 8].
Finally, compare 5 and 8 → no swap → [1, 4, 2, 5, 8].
Repeat the process until the list is sorted.

Use:
Best for small or nearly sorted data. It has a worst-case time complexity of O(n²), which makes it inefficient for large lists but adaptive for partially sorted ones.

Merge Sort

- **What it is**: A divide-and-conquer sorting algorithm that splits the array into halves, sorts each half, and merges them back together.
- **How it works**:
    - Recursively divide the array until each half has one element.
    - Merge two halves by comparing elements, sorting as they combine back.

Example: `[38, 27, 43, 3, 9]` is split and sorted as:

```
Split: [38, 27, 43, 3, 9] → [38, 27, 43] and [3, 9] → [27, 38, 43]
and [3, 9] → Merged to [3, 9, 27, 38, 43]
```

-
- **Use**: Efficient for large lists, with a time complexity of O(nlogn)O(n \log n)O(nlogn). Used in cases where stable sorting is required.

Selection Sort

- **What it is**: A simple sorting algorithm that repeatedly finds the minimum element and places it at the beginning of the unsorted part.
- **How it works**:
    - Start with the first element, find the smallest element in the rest of the array, and swap them.
    - Repeat for the next position until sorted.

Example: For `[64, 25, 12, 22, 11]`, selection sort proceeds by finding `11` and placing it at the start, then `12`, and so on:
`[11, 12, 22, 25, 64]`

-
- **Use**: Simple but inefficient for large lists (O(n2)O(n^2)O(n2)). Useful for small lists or when memory swaps are expensive.

Insertion Sort

- **What it is**: A sorting algorithm that builds the sorted array one item at a time by comparing and inserting it in the correct position.
- **How it works**:
    - Start with the second element and compare it to all elements before it, inserting it in the correct position.

Example: For `[5, 3, 8, 6, 2]`, insert `3` before `5`, then `8` stays, `6` goes between `5` and `8`, and finally, `2` goes to the beginning:
`[2, 3, 5, 6, 8]`

- **Use**: Best for small or mostly sorted data, as it is efficient for short lists (O(n2)O(n^2)O(n2) worst-case) but adaptive.

---

10. Searching Algorithms

- **Type**: Algorithms (not data structures)
- **Description**: Algorithms to find elements in data structures.
- **Complexity**: Varies based on algorithm
- **Examples**:
    - **Linear Search**: O(n)O(n)O(n)
    - **Binary Search**: O(logn)O(\log n)O(logn) for sorted data
- **Use Cases**: Searching in databases, application of data filtering, and retrieval systems.

- **What it is**: A search algorithm that finds an element's position in a sorted array by repeatedly dividing the search range in half.
- **How it works**: Start with the middle element. If it's equal to the target, stop. If the target is smaller, repeat on the left half; if larger, repeat on the right half.
- **Example**: In `[1, 3, 5, 7, 9]`, to find `7`, check the middle `5`. Since `7 > 5`, discard the left half and check the right, where `7` is found.
- **Use**: Efficient searching in sorted arrays or lists. Complexity is O(logn)O(\log n)O(logn).

Linear Search

- **What it is**: A simple search algorithm that checks each element in the list sequentially until it finds the target element.
- **How it works**:
    - Start from the first element and check each item until you find the target.
- **Example**: In `[5, 3, 8, 6, 2]`, searching for `8` means checking each element until `8` is found.
- **Use**: Useful for unsorted data where binary search cannot be applied. Complexity is O(n)O(n)O(n), making it slower for large lists but simple to implement.

Binary Search

is an efficient algorithm for finding a target value within a **sorted array** by repeatedly dividing the search interval in half.

How it works:

1. Start with the entire sorted array.
2. Find the middle element of the array.
3. If the middle element is the target, you're done.
4. If the target is smaller than the middle element, repeat the search on the left half.
5. If the target is larger than the middle element, repeat the search on the right half.
6. Continue halving the search range until the target is found or the range is empty.

Example:
For the array `[1, 3, 5, 7, 9, 11]` and target `7`:

- Start with the middle element (5). Since 7 > 5, discard the left half.
- Now the new range is `[7, 9, 11]`. The middle element is 9. Since 7 < 9, discard the right half.
- The new range is `[7]`, which is the target.

Use:
Best for searching in **sorted arrays**. It has a time complexity of **O(log n)**, making it much faster than linear search (O(n)) for large datasets.
