

Data Structures and Algorithms

1 Data Structures

1.1 Arrays

- **Definition:** A collection of elements identified by index or key.
- **Types:** Static (fixed size) and Dynamic (resizable).
- **Operations:** Insertion, Deletion, Access ($O(1)$ for access by index).

1.2 Linked Lists

- **Definition:** A linear collection of elements called nodes, where each node points to the next.
- **Types:** Singly Linked List, Doubly Linked List, Circular Linked List.
- **Operations:** Insertion, Deletion ($O(1)$ if the node is known), Traversal ($O(n)$).

1.3 Stacks

- **Definition:** A linear data structure that follows the Last In First Out (LIFO) principle.
- **Operations:** Push (insert), Pop (remove), Peek (top element).

1.4 Queues

- **Definition:** A linear data structure that follows the First In First Out (FIFO) principle.
- **Types:** Simple Queue, Circular Queue, Priority Queue, Deque.
- **Operations:** Enqueue (insert), Dequeue (remove), Peek (front element).

1.5 Hash Tables

- **Definition:** A data structure that stores key-value pairs for efficient retrieval.
- **Operations:** Insertion, Deletion, Search (average $O(1)$).

1.6 Trees

- **Definition:** A hierarchical data structure with a root node and child nodes forming a parent-child relationship.
- **Types:** Binary Tree, Binary Search Tree (BST), AVL Tree, Red-Black Tree, B-Trees, Heaps.
- **Operations:** Insertion, Deletion, Traversal (Inorder, Preorder, Postorder).

1.7 Graphs

- **Definition:** A collection of nodes (vertices) connected by edges.
- **Types:** Directed, Undirected, Weighted, Unweighted.
- **Representations:** Adjacency Matrix, Adjacency List.
- **Operations:** Traversal (DFS, BFS), Shortest Path (Dijkstra's, Bellman-Ford), Minimum Spanning Tree (Kruskal's, Prim's).

2 Algorithms

2.1 Sorting Algorithms

- **Bubble Sort:** Simple but inefficient. $O(n^2)$.
- **Selection Sort:** Selects the minimum element and swaps. $O(n^2)$.
- **Insertion Sort:** Builds the final sorted array one item at a time. $O(n^2)$.
- **Merge Sort:** Divide and conquer approach. $O(n \log n)$.
- **Quick Sort:** Divide and conquer approach, with a pivot element. Average $O(n \log n)$.
- **Heap Sort:** Utilizes a binary heap data structure. $O(n \log n)$.

2.2 Searching Algorithms

- **Linear Search:** Sequentially checks each element. $O(n)$.
- **Binary Search:** Requires sorted array, divides the array in half. $O(\log n)$.

2.3 Dynamic Programming

- **Definition:** Solves complex problems by breaking them down into simpler subproblems and storing the results of subproblems to avoid redundant computing.
- **Examples:** Fibonacci Sequence, Knapsack Problem, Longest Common Subsequence.

2.4 Greedy Algorithms

- **Definition:** Builds up a solution piece by piece, always choosing the next piece that offers the most immediate benefit.
- **Examples:** Coin Change Problem, Activity Selection, Huffman Coding.

2.5 Divide and Conquer

- **Definition:** Breaks a problem into smaller subproblems, solves them independently, and combines their solutions to solve the original problem.
- **Examples:** Merge Sort, Quick Sort, Binary Search.

2.6 Backtracking

- **Definition:** Builds a solution incrementally and removes those solutions that fail to satisfy the constraints of the problem.
- **Examples:** N-Queens Problem, Sudoku Solver, Hamiltonian Path.

2.7 Graph Algorithms

- **Depth First Search (DFS):** Explores as far as possible along each branch before backtracking. $O(V + E)$.
- **Breadth First Search (BFS):** Explores all neighbors at the present depth before moving on to nodes at the next depth level. $O(V + E)$.
- **Dijkstra's Algorithm:** Finds the shortest path from a source node to all other nodes in a weighted graph. $O(V^2)$ or $O(E + V \log V)$ with a priority queue.
- **Bellman-Ford Algorithm:** Computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph. $O(VE)$.
- **Kruskal's Algorithm:** Finds the minimum spanning tree for a connected, weighted graph. $O(E \log E)$.
- **Prim's Algorithm:** Finds the minimum spanning tree for a connected, weighted graph. $O(V^2)$ or $O(E + V \log V)$ with a priority queue.

3 Additional Concepts

3.1 Big O Notation

- **Definition:** Describes the upper bound of the time complexity or space complexity of an algorithm.
- **Common Complexities:** $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^n)$, $O(n!)$.

3.2 Recursion

- **Definition:** A function calls itself as a subroutine.
- **Base Case and Recursive Case:** Essential to prevent infinite loops.

3.3 Amortized Analysis

- **Definition:** Average time per operation over a sequence of operations, spread out over time.

3.4 Space Complexity

- **Definition:** The amount of memory an algorithm needs to run to completion.

4 Linear Data Structures

A linear data structure is a type of data structure in which elements are arranged sequentially, and each element is connected to its previous and next elements. Because of this arrangement, linear data structures are easy to implement and understand.

4.1 Types of Linear Data Structures

4.1.1 Arrays

- **Definition:** A collection of elements identified by index or key.
- **Operations:** Insertion, deletion, access.
- **Example:** An array of integers: $[1, 2, 3, 4, 5]$.

4.1.2 Linked Lists

- **Definition:** A collection of nodes where each node contains a data part and a reference (or link) to the next node in the sequence.
- **Types:** Singly linked list, doubly linked list, circular linked list.
- **Example:** A singly linked list with three nodes containing values 1, 2, and 3:

[1] -> [2] -> [3] -> NULL

4.1.3 Stacks

- **Definition:** A linear data structure that follows the Last In First Out (LIFO) principle.
- **Operations:** Push (insert), pop (remove), peek (view top element).
- **Example:** A stack of books where you can only take the top book off the stack.

4.1.4 Queues

- **Definition:** A linear data structure that follows the First In First Out (FIFO) principle.
- **Types:** Simple queue, circular queue, priority queue, deque.
- **Operations:** Enqueue (insert), dequeue (remove), peek (view front element).
- **Example:** A line of people waiting to buy tickets where the first person in line is the first to buy a ticket.

5 Concepts of LIFO and FIFO

5.1 LIFO (Last In First Out)

LIFO is a principle where the last element added to a structure is the first one to be removed. This principle is primarily used in stacks.

- **Example:** Think of a stack of plates. You can only take the top plate off the stack, and the last plate added is the first one to be removed.

Stack:

Top -> Plate 3
 Plate 2
 Plate 1

- **Push:** Adding a plate to the stack (Push Plate 4).
- **Pop:** Removing the top plate from the stack (Pop Plate 4).

5.2 FIFO (First In First Out)

FIFO is a principle where the first element added to a structure is the first one to be removed. This principle is primarily used in queues.

- **Example:** Think of a queue of people at a ticket counter. The person who arrives first gets served first.

Queue:

```
Front -> Person 1
        Person 2
        Person 3 -> Rear
```

- **Enqueue:** Adding a person to the end of the queue (Enqueue Person 4).
- **Dequeue:** Removing the person at the front of the queue (Dequeue Person 1).

6 Examples in Code

6.1 Stack (LIFO)

Listing 1: Stack implementation using a list

```
# Stack implementation using a list
stack = []

# Push operation
stack.append(1)
stack.append(2)
stack.append(3)

print("Stack_after_pushes:", stack)

# Pop operation
top_element = stack.pop()
print("Popped_element:", top_element)
print("Stack_after_pop:", stack)
```

Output:

```
Stack after pushes: [1, 2, 3]
Popped element: 3
Stack after pop: [1, 2]
```

6.2 Queue (FIFO)

Listing 2: Queue implementation using a list

```
# Queue implementation using a list
from collections import deque

queue = deque()

# Enqueue operation
queue.append(1)
queue.append(2)
queue.append(3)

print("Queue_after_enqueues:", list(queue))

# Dequeue operation
front_element = queue.popleft()
print("Dequeued_element:", front_element)
print("Queue_after_dequeue:", list(queue))
```

Output:

```
Queue after enqueues: [1, 2, 3]
Dequeued element: 1
Queue after dequeue: [2, 3]
```