Data Structures Cheatsheet

A quick reference guide to common data structures, their properties, and common use cases. This cheatsheet provides a concise overview for developers and students.

Arrays and Linked Lists

Arrays

Definition:	Contiguous block of memory holding elements of the same type.
Access:	Random access (O(1)) using index.
Insertion/Deletion:	O(n) in the worst case (shifting elements).
Use Cases:	Storing and accessing elements by index, implementing stacks and queues.
Memory:	Requires contiguous memory; can lead to fragmentation.
Example:	`arr = [1, 2, 3, 4, 5]` `print(arr[2]) # Output: 3`

Linked Lists

Definition:	Collection of nodes, each containing data and a pointer to the next node.
Access:	Sequential access (O(n)).
Insertion/Deletion:	O(1) if the node is known.
Use Cases:	Implementing stacks, queues, and graphs; dynamic memory allocation.
Memory:	Non-contiguous memory; more flexible memory usage.

Example:	`class Node:`
	`  def __init__(self, data):`
	`  self.data = data`
	`  self.next = None`

Comparison

Arrays offer faster access but slower insertion/deletion compared to linked lists. Linked lists use memory more efficiently in dynamic scenarios.
Arrays require a contiguous block of memory, while linked lists can be scattered in memory.

Stacks and Queues

Stacks

Definition:	LIFO (Last-In, First-Out) data structure.
Operations:	Push (add element), Pop (remove element), Peek (view top element).
Implementation:	Arrays or linked lists.
Use Cases:	Function call stack, expression evaluation, backtracking.
Time Complexity:	O(1) for push and pop operations.

Example:	`stack = []`
	`stack.append(1) # Push`
	`stack.pop()  # Pop`

Queues

Definition:	FIFO (First-In, First-Out) data structure.
Operations:	Enqueue (add element), Dequeue (remove element).
Implementation:	Arrays or linked lists.
Use Cases:	Task scheduling, breadth-first search (BFS).
Time Complexity:	O(1) for enqueue and dequeue operations (using linked list or circular array).

Example:	`from collections import deque`
	`queue = deque()`
	`queue.append(1) # Enqueue`
	`queue.popleft() # Dequeue`

Comparison

Stacks and queues differ in their ordering principle: LIFO vs. FIFO. Stacks are used for tasks that require reversing order, while queues maintain order.
Stacks often manage function calls, while queues handle task scheduling and processing.

Trees

Binary Trees

Definition:	Each node has at most two children: left and right.
Traversal Methods:	Inorder, Preorder, Postorder.
Use Cases:	Expression parsing, decision trees.

Example:	`class Node:`
	`  def __init__(self, data):`
	`  self.data = data`
	`  self.left = None`
	`  self.right = None`

Balanced vs Unbalanced:	Balanced trees have height O(log n), while unbalanced trees can have height O(n).

Binary Search Trees (BST)

Definition:	Binary tree where for each node, all nodes in the left subtree are smaller, and all nodes in the right subtree are larger.
Operations:	Search, insert, delete.
Time Complexity:	O(log n) on average, O(n) in the worst case (unbalanced tree).
Use Cases:	Efficient searching, sorting, and retrieval.

Example:	`# Insert operation in BST`
	`def insert(root, data):`
	`  if root is None:`
	`  return Node(data)`
	`  else:`
	`  if data < root.data:`
	`  root.left = insert(root.left, data)`
	`  else:`
	`  root.right = insert(root.right, data)`
	`  return root`

Heaps

Definition:	Special tree-based data structure that satisfies the heap property: Min-Heap (parent <= children) or Max-Heap (parent >= children).
Types:	Binary Heap, Fibonacci Heap.
Use Cases:	Priority queues, heap sort.
Time Complexity:	O(log n) for insertion and deletion.

Example:	`import heapq`
	
	`heap = []`
	`heapq.heappush(heap, 5) # Insert`
	`heapq.heappop(heap) # Remove min`

Hash Tables

Core Concepts

Definition:	Data structure that implements an associative array abstract data type, which maps keys to values.
Hash Function:	Function that maps keys to indices in the array.
Collision Handling:	Techniques to handle multiple keys mapping to the same index (e.g., chaining, open addressing).
Use Cases:	Implementing dictionaries, caching, symbol tables.

Example:	`dictionary = {}`
	`dictionary['apple'] = 1`
	`print(dictionary['apple']) # Output: 1`

Collision Resolution Techniques

Chaining:	Each index in the hash table points to a linked list of key-value pairs.
Open Addressing:	If a collision occurs, probe for an empty slot in the table (e.g., linear probing, quadratic probing, double hashing).
Time Complexity:	O(1) average case (with good hash function), O(n) worst case (all keys map to the same index).
Load Factor:	Ratio of the number of entries to the number of buckets. High load factors increase collision probability.

Considerations

Choosing a good hash function is crucial for the performance of a hash table. A poorly chosen hash function can lead to frequent collisions and O(n) performance.
Load factor should be monitored and the hash table resized when it exceeds a certain threshold to maintain good performance.