Comprehensive Data Structures Guide — RAG Optimized Edition

A structured, retrieval-optimized reference on data structures with theory, code, and examples.

1. Introduction

Data structures are the foundation of computer science and software engineering. They define the way data is stored, accessed, and manipulated. This guide provides a detailed, language-neutral overview of the most important data structures, with Python code snippets to illustrate concepts. It is designed for integration into Retrieval-Augmented Generation (RAG) systems — enabling effective information retrieval and context-aware learning.

2. Types of Data Structures

Data structures can be broadly classified into two main categories: - **Linear Data Structures**: Elements are arranged sequentially (e.g., arrays, linked lists, stacks, queues). - **Non-linear Data Structures**: Elements have hierarchical or network relationships (e.g., trees, graphs).

3. Arrays

An array is a fixed-size collection of elements of the same type. Elements are stored in contiguous memory locations. Key operations include traversal, insertion, deletion, and search.

Example (Python):

```
# Creating and traversing an array arr = [10, 20, 30, 40] for item in arr: print(item) #
Insertion arr.append(50) # Deletion arr.remove(30)
```

4. Linked Lists

A linked list is a linear data structure in which each element (node) contains data and a reference to the next node. Types include singly, doubly, and circular linked lists.

```
class Node: def __init__(self, data): self.data = data self.next = None class LinkedList: def
__init__(self): self.head = None def insert(self, data): new_node = Node(data) new_node.next
= self.head self.head = new_node def display(self): temp = self.head while temp:
print(temp.data, end=" -> ") temp = temp.next
```

5. Stacks and Queues

A **stack** follows LIFO (Last In First Out) and a **queue** follows FIFO (First In First Out) principles.

```
# Stack implementation using list stack = [] stack.append(10) stack.append(20) stack.pop() #
Queue implementation using collections.deque from collections import deque queue = deque()
queue.append(1) queue.append(2) queue.popleft()
```

6. Trees

A tree is a non-linear data structure consisting of nodes connected by edges. The top node is called the root, and nodes with no children are called leaves. Common types include Binary Tree, Binary Search Tree (BST), AVL Tree, and Heap.

```
class Node: def __init__(self, key): self.key = key self.left = None self.right = None def
inorder(root): if root: inorder(root.left) print(root.key, end=" ") inorder(root.right)
```

7. Graphs

Graphs consist of nodes (vertices) and edges connecting them. They can be directed or undirected, weighted or unweighted. Common algorithms: BFS, DFS.

```
graph = { 'A': ['B', 'C'], 'B': ['A', 'D', 'E'], 'C': ['A', 'F'], 'D': ['B'], 'E': ['B', 'F'],
'F': ['C', 'E'] } def bfs(graph, start): visited = set() queue = [start] while queue: vertex
= queue.pop(0) if vertex not in visited: print(vertex, end=" ") visited.add(vertex)
queue.extend(set(graph[vertex]) - visited)
```

8. Hash Tables

A hash table stores key-value pairs using a hash function for fast lookup. Collisions are handled using chaining or open addressing.

```
hash_table = {} hash_table["apple"] = 10 hash_table["banana"] = 20
print(hash_table.get("apple"))
```

9. RAG-Specific Applications

In Retrieval-Augmented Generation systems, efficient data structures are essential for managing embeddings, indices, and context lookups. For example: - Trees (e.g., KD-Trees) and Hash Tables enable fast similarity searches in vector databases. - Graph structures support knowledge graph retrieval and reasoning. - Queues and stacks can manage multi-turn conversation state in dialogue systems.

10. Summary and Complexity Table

Data Structure	Time Complexity (Search)	Time Complexity (Insert)	Space
Array	O(n)	O(n)	O(n)
Linked List	O(n)	O(1)	O(n)
Stack	O(1)	O(1)	O(n)
Queue	O(1)	O(1)	O(n)
BST	O(log n)	O(log n)	O(n)
Hash Table	O(1)	O(1)	O(n)
Graph	O(V+E)	O(V+E)	O(V+E)

End of Document