

Data Structures: Organizing Information in Code



- **What we cover:** a language-neutral model of collections, stacks, queues, and trees



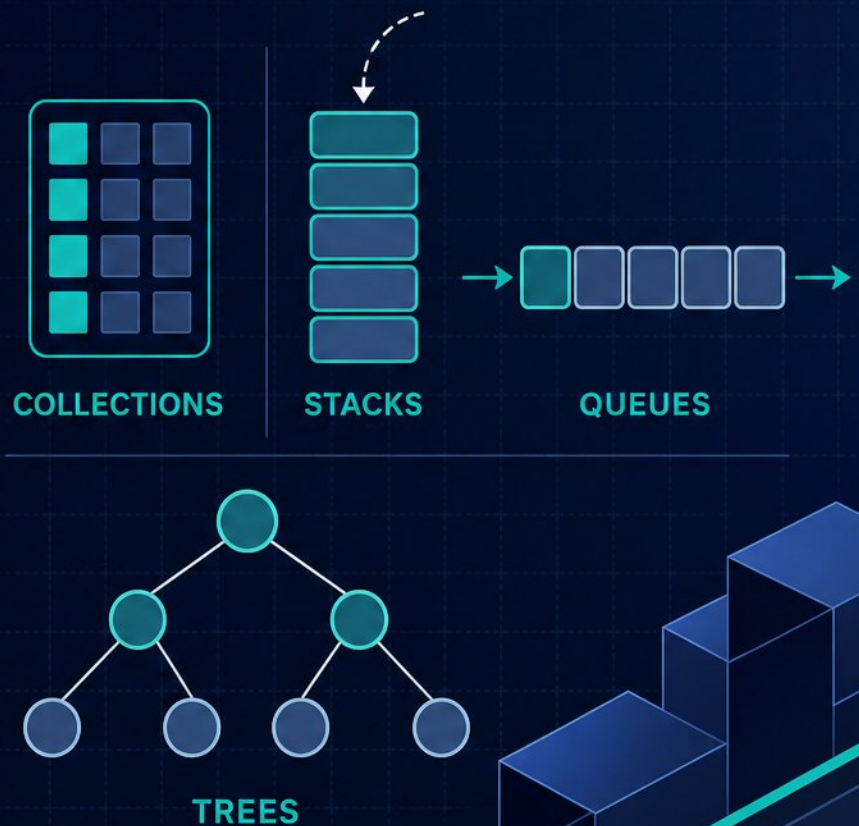
- **Why it matters:** your choice affects correctness, clarity, and real-world performance



- **How we learn:** conceptual models, pseudocode, and analogies, not syntax

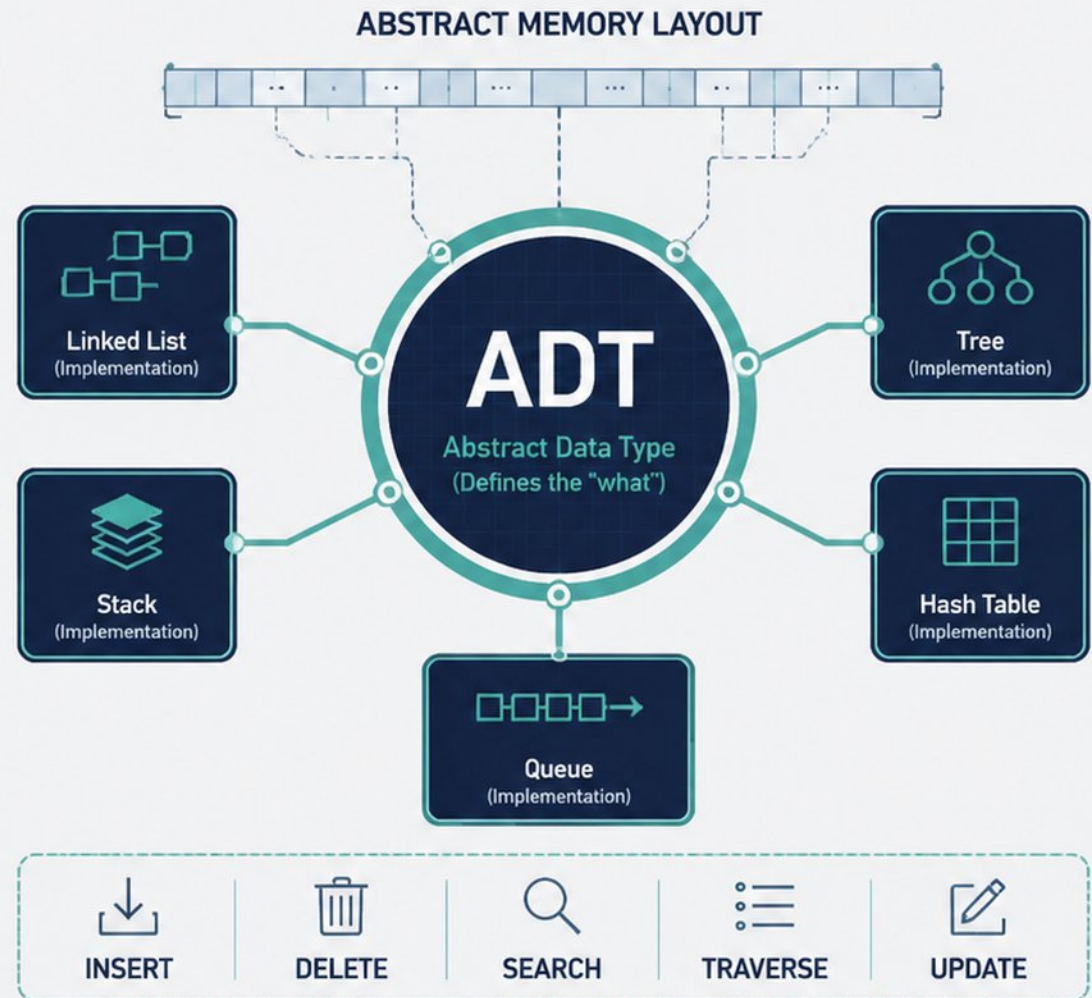


- **What you gain:** recognize structures, explain behavior, and reason about trade-offs



What Is a Data Structure?

- A data structure organizes and stores data for efficient access and modification
- Abstract Data Type (ADT) defines behavior (the "what"); data structure provides the implementation (the "how")
- Core operations: insert, delete, search, traverse, and update
- Language-neutral approach: focus on behavior and constraints, using pseudocode for illustration

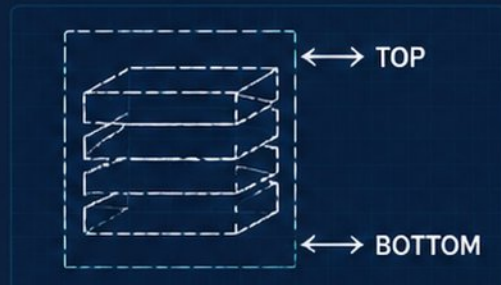


The ADT vs. Data Structure Distinction

- An ADT defines **what** operations are allowed, not **how** they work.
- A data structure is the concrete implementation of an ADT.
- **Example:** Stack (ADT) vs. Array-based Stack (data structure).
- This separation lets us reason about behavior independently from performance.
- One ADT can have many implementations, each with different trade-offs.

STACK (ADT)

Abstract Interface (What)



Allowed Operations



push(x)

Add an element to the top.



pop()

Remove the top element.



peek()

View the top element.



isEmpty()

Check if the stack is empty.

VS.

ARRAY-BASED STACK (DATA STRUCTURE)

Concrete Implementation (How)

Abstract Memory Layout



```
array[0....n-1]    // fixed-size array
top = -1           // index of top element
```

```
push(x):   top = top + 1; array[top] = x
pop():     x = array[top]; top = top - 1
peek():    return array[top]
isEmpty(): return (top == -1)
```



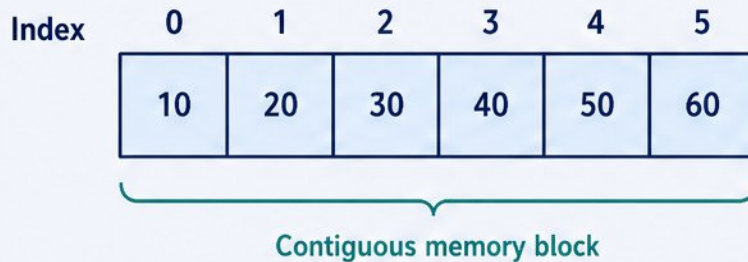
Key Takeaway

An ADT defines the behavior (the contract). A data structure determines how that behavior is realized in memory.

The Linear Collections: Two Fundamental Layouts

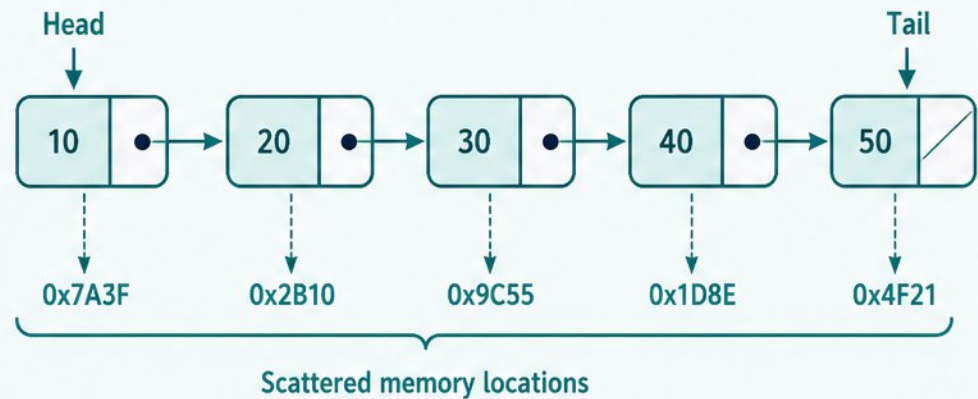
Abstract Memory Layout

ARRAY (Contiguous Memory)



VS

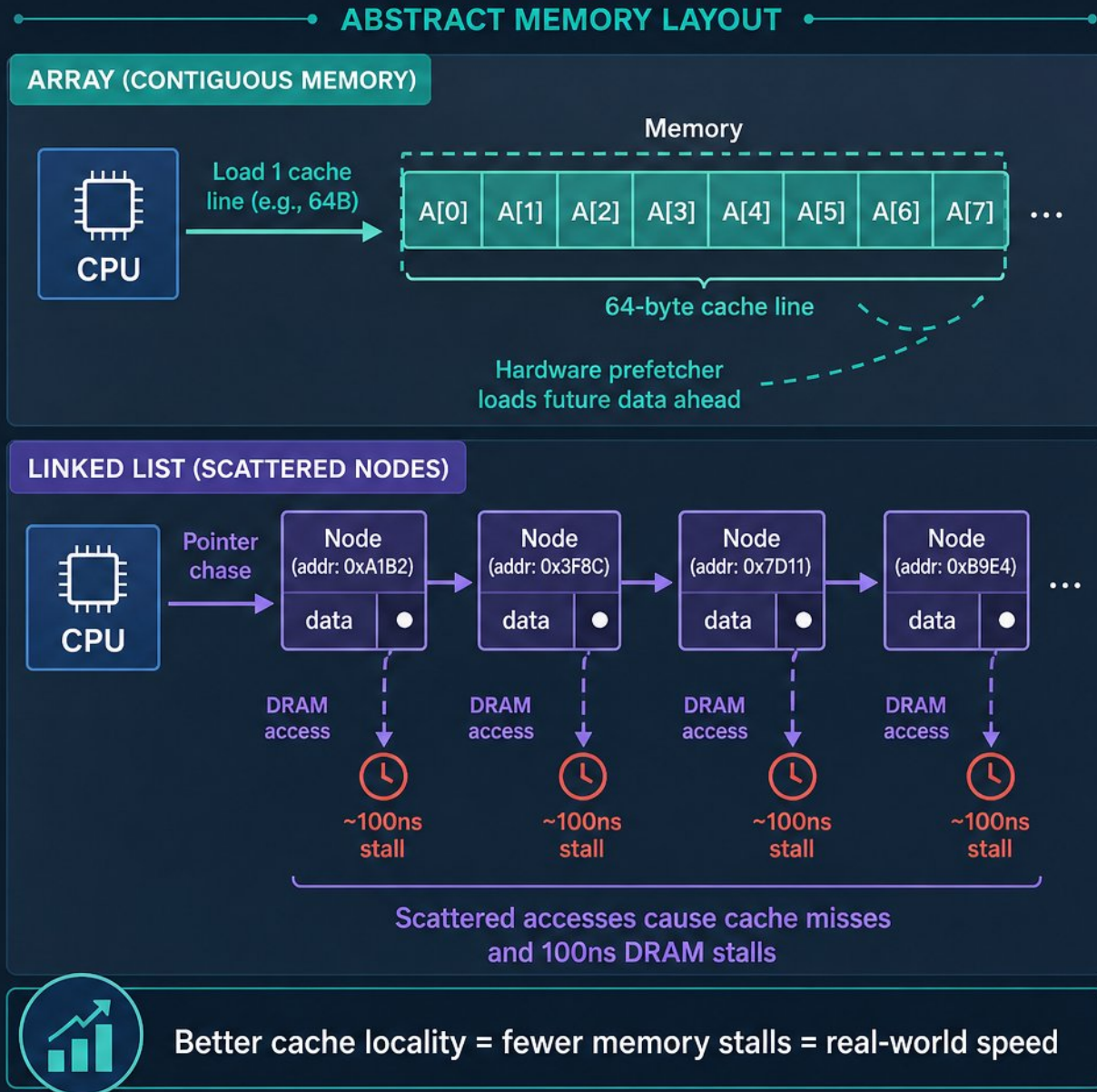
LINKED LIST (Non-contiguous Memory)



- 
 - 
 - 
 - 
 - 
- **Arrays:** elements in contiguous memory block, $O(1)$ index access
 - **Linked lists:** scattered nodes connected by pointers, $O(n)$ index access
 - **Array insertion/deletion in middle:** $O(n)$ due to element shifting
 - **Linked list insertion/deletion at known node:** $O(1)$ pointer update
 - **Key terms:** node, head, tail, pointer, index

Why Arrays Usually Win in Practice

- Performance is **not just Big-O**: CPU cache locality has a massive impact on real speed.
- Arrays load a **cache line** of adjacent elements; the prefetcher loads future data ahead.
- Linked list nodes are scattered; each pointer chase risks a **100ns DRAM stall**.
- **$O(1)$** insertion only helps if you already hold the pointer; finding it requires **$O(n)$** traversal.
- This is why dynamic arrays (Python `list`, Java `ArrayList`) are the default collection.



When Linked Lists Are the Right Tool



- Win when you already hold a pointer to the mutation node: truly $O(1)$



- Canonical example: LRU Cache — hash map gives $O(1)$ node access, doubly linked list gives $O(1)$ reorder/eviction



- Other wins: $O(1)$ range splicing, iterator stability across mutations, pure pointer-reversal problems

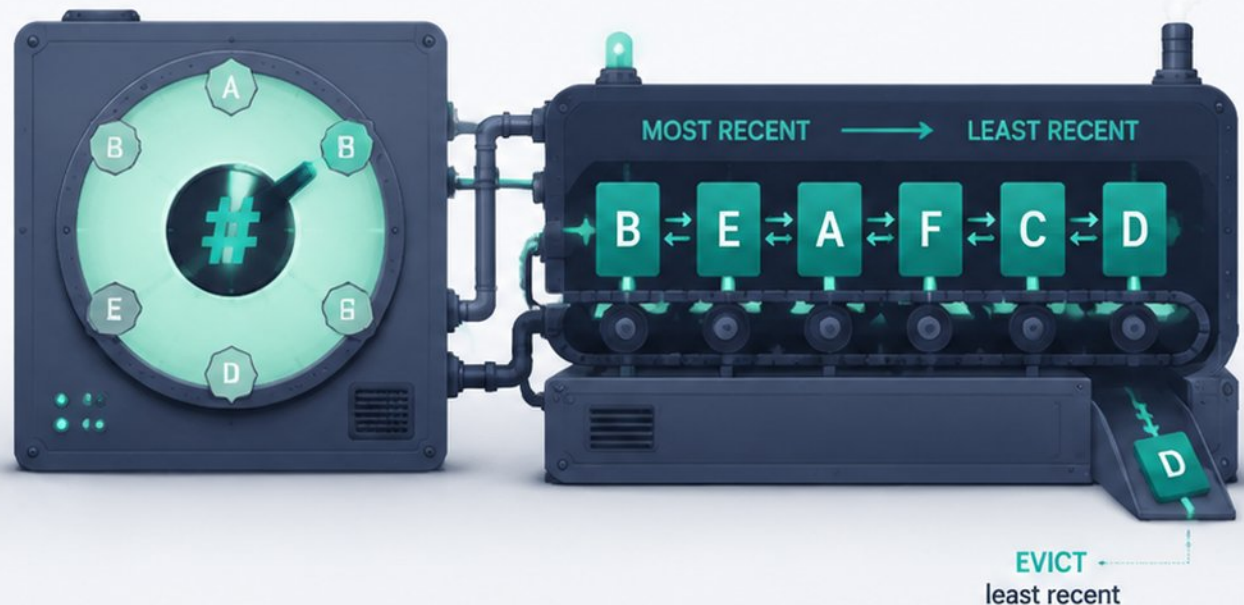


- Production FIFO queues use circular buffers, not linked lists

LRU CACHE: A TWO-PART MACHINE

1) HASH MAP DIAL
 $O(1)$ node access

2) DOUBLY LINKED LIST CONVEYOR
 $O(1)$ reorder / eviction



GET / PUT
key lookup



MOVE TO FRONT
 $O(1)$



EVICT FROM BACK
 $O(1)$



Hash map gives $O(1)$ where a node is.
Doubly linked list makes reordering and eviction $O(1)$.

Stacks: Last-In, First-Out (LIFO)



- Last-in, first-out: the most recently added element is removed first



- Core operations: push (add to top), pop (remove from top), peek (view top)



- Only the top is accessible—this constraint guarantees predictable behavior



- Common use cases: undo/redo, function call stack, expression evaluation, backtracking



- All core operations run in $O(1)$ constant time



Real-World Analogy

Queues: First-In, First-Out (FIFO)



- **FIFO principle:** oldest waiting element processed first (like a ticket line)



- **Core operations:** enqueue (add to rear), dequeue (remove from front), front/peek



- **Insertion at rear, removal from front** ensures fairness and order preservation



- **Common uses:** task scheduling, BFS, print spooling, request buffering

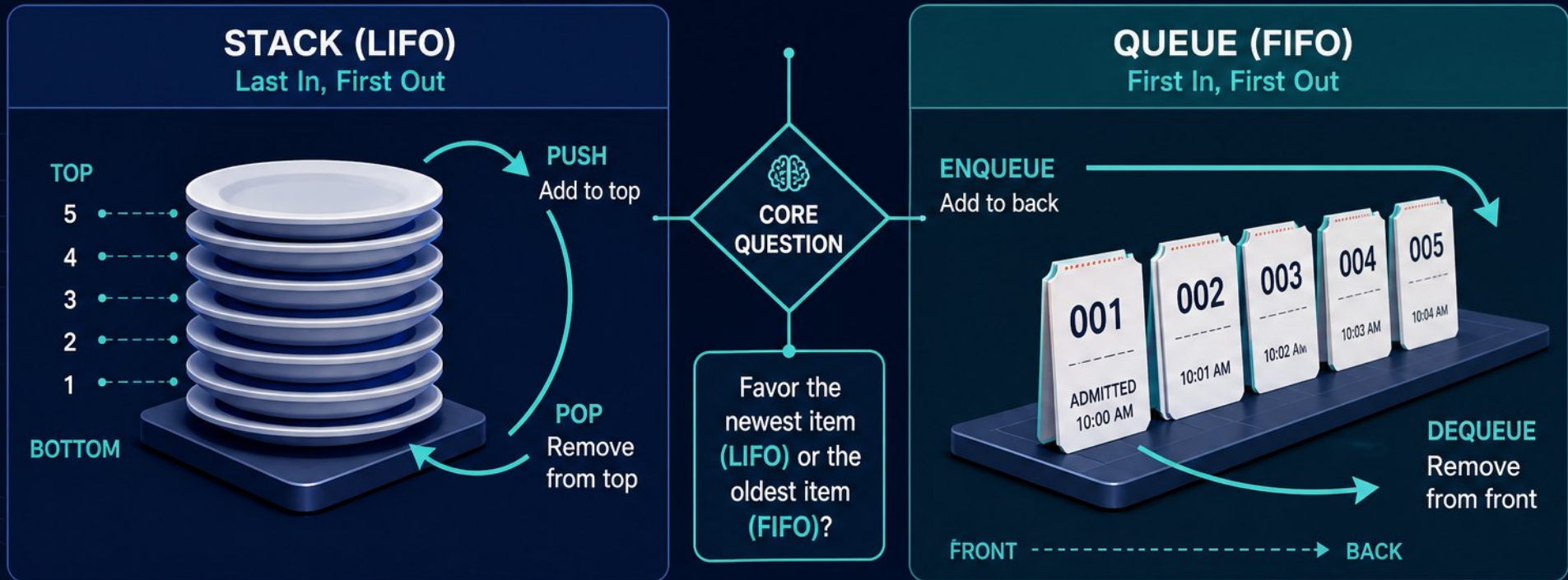


- **Circular queue:** fixed-size array achieving $O(1)$ enqueue/dequeue without shifting

REAL-WORLD ANALOGY



Stacks vs. Queues: The Decision Framework



- ? | - Core question: favor the newest item (LIFO) or the oldest item (FIFO)?
- 📦 | - Choose a Stack for "most recent" dependencies: undo, backtracking, DFS, nested matching.
- 👥 | - Choose a Queue when fairness or arrival order matters: scheduling, BFS, request buffers.
- 🔒 | - Both are restricted-access structures; the constraint is a behavioral guarantee.
- ↔️ | - Swap a stack for a queue in BFS, and you get DFS—the structure dictates the algorithm.

Pseudocode Conventions and Common Patterns

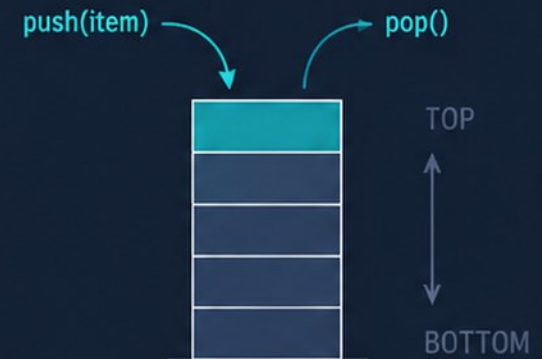
- We'll use consistent language-neutral pseudocode to model data structures and their operations
- Stack pattern: `stack.push(item)`, `item = stack.pop()`, `item = stack.peek()`
- Queue pattern: `queue.enqueue(item)`, `item = queue.dequeue()`, `item = queue.front()`
- These patterns are the foundation for implementing these structures in any language

1 # Stack pattern

2 `stack.push(item)`

3 `item = stack.pop()`

4 `item = stack.peek()`

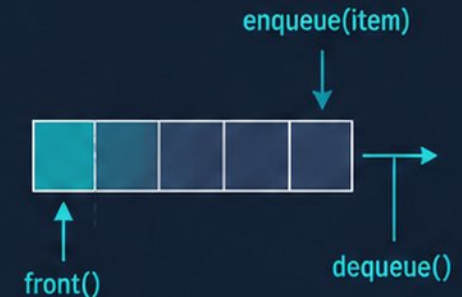


6 # Queue pattern

7 `queue.enqueue(item)`

8 `item = queue.dequeue()`

9 `item = queue.front()`

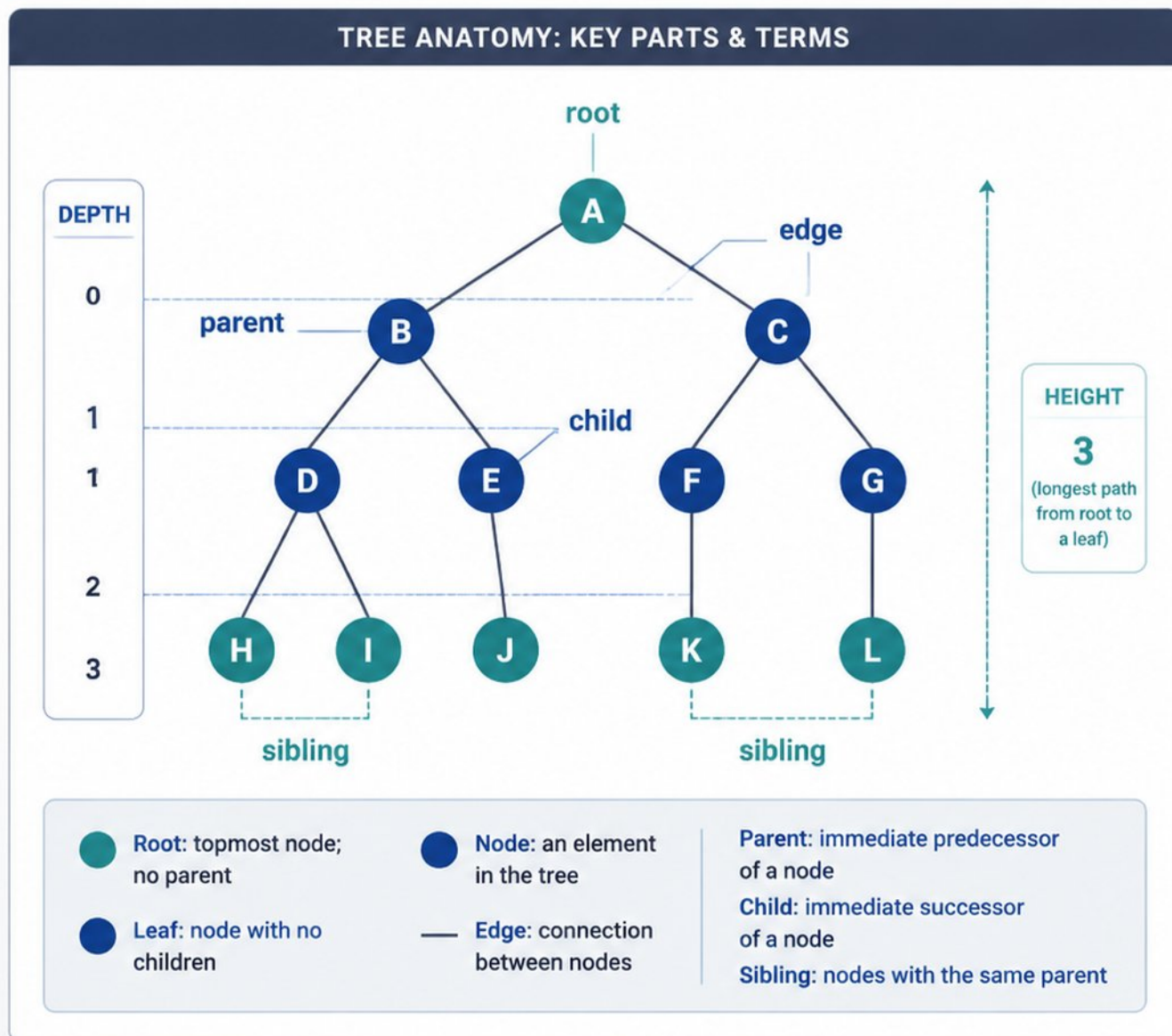




Stepping Off the Line:

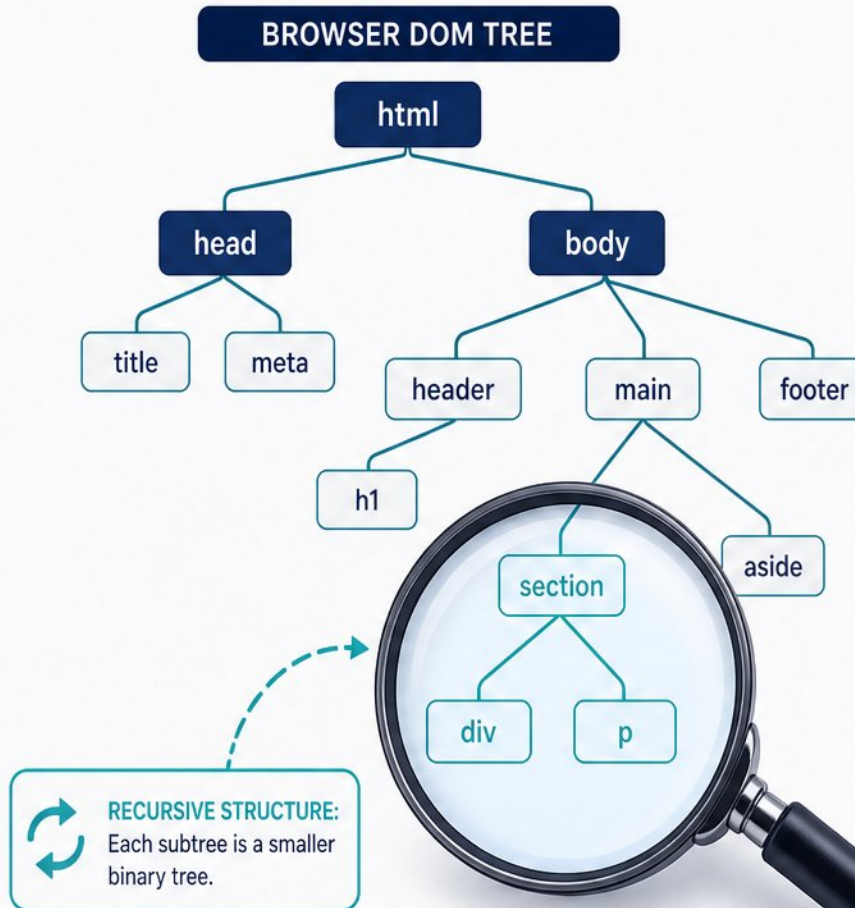
Introducing Hierarchical Data

- Linear structures model sequences; hierarchical structures model parent-child relationships
- A tree is a collection of nodes connected by edges, with a single root and no cycles
- Key terms: root, leaf, node, edge, parent, child, sibling, depth, height
- Every subtree is itself a tree, making recursion a natural fit for tree operations

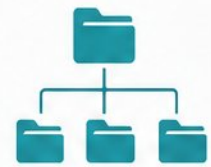


Binary Trees: The Fundamental Building Block

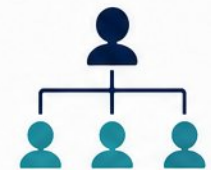
- A binary tree: each node has at most two children—left and right.
- Real-world examples: file systems, org charts, and the browser DOM.
- Depth-first search (DFS) uses a stack to explore deep branches first.
- Breadth-first search (BFS) uses a queue to explore level by level.
- The traversal strategy determines the data structure: stack for DFS, queue for BFS.



REAL-WORLD ANALOGY



File Systems



Org Charts



Browser DOM



DFS (Depth-First Search)

Uses a stack (LIFO). Explores as far down a branch as possible before backtracking.

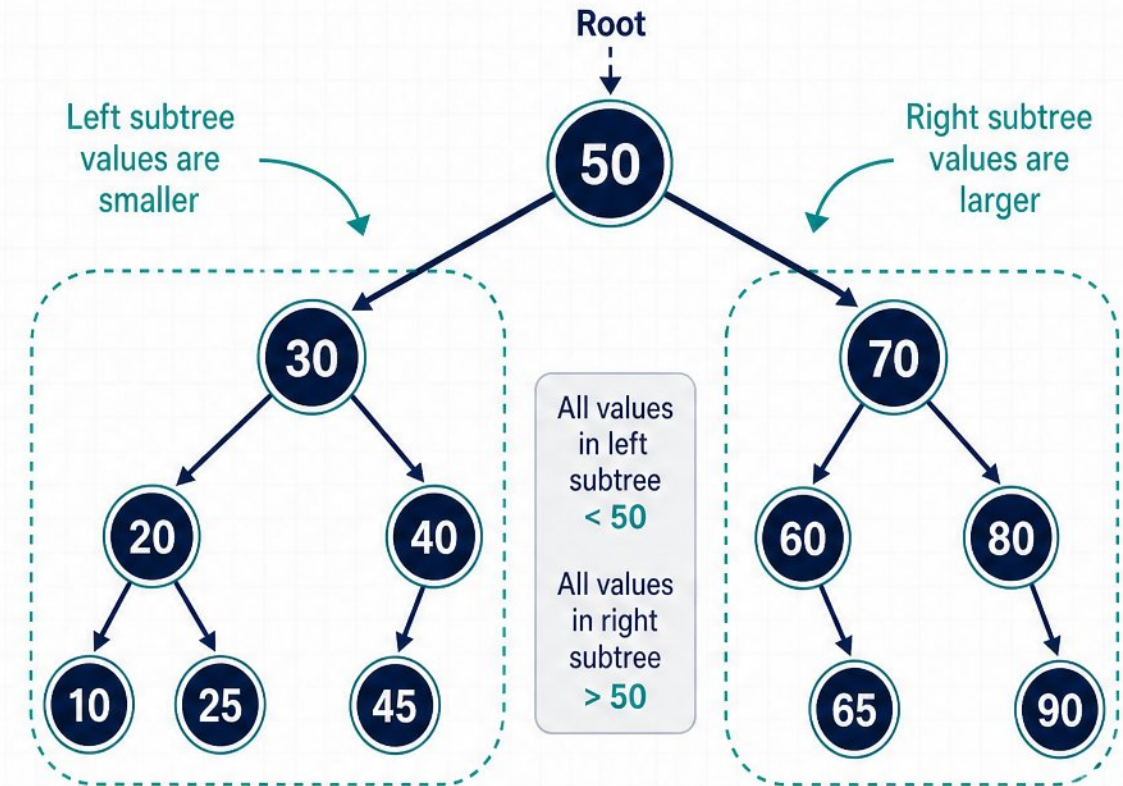


BFS (Breadth-First Search)

Uses a queue (FIFO). Explores nodes level by level from left to right.

Binary Search Trees: Organized for Fast Lookup

- The BST property: left subtree values are smaller, right subtree values are larger
- Enables efficient search, insertion, and deletion—mirroring binary search on a sorted array
- In a balanced BST, operations are $O(\log n)$; you halve the search space with each comparison
- Combines the insert efficiency of a linked list with the search efficiency of a sorted array
- Tree shape depends entirely on insertion order; sorted data creates a degenerate linked list



Mirrors binary search on a sorted array

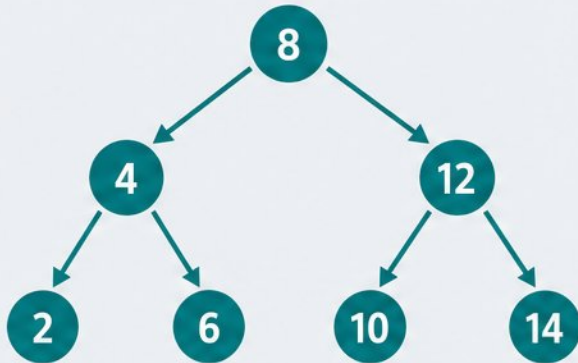
10	20	25	30	40	45	50	60	65	70	80	90
----	----	----	----	----	----	----	----	----	----	----	----

Go left: target is smaller

Go right: target is larger

The Hidden Danger of Degenerate Trees

BALANCED BST

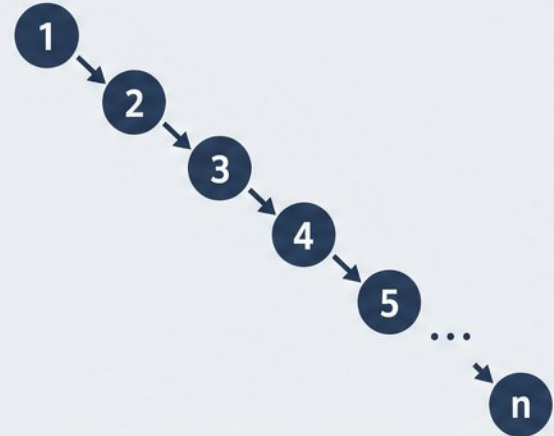


Height: $O(\log n)$
Operations: Efficient



INSERTION
ORDER
MATTERS

DEGENERATE TREE



Height: $O(n)$
Operations: Linear (worst-case)



- Inserting sorted data creates a degenerate tree, essentially a linked list
- Operations degrade from $O(\log n)$ to $O(n)$ worst-case
- The tree's shape—and performance—depends entirely on insertion order
- A basic BST offers no guarantee of balance
- Self-balancing variants (AVL, Red-Black) automatically restructure to maintain $O(\log n)$ height

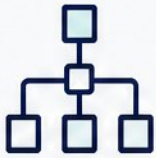
A Tale of Two Traversals: DFS vs. BFS



- **DFS** uses a **stack** to explore deep first, backtracking when needed.



- **BFS** uses a **queue** to explore level by level, guaranteeing shortest paths.

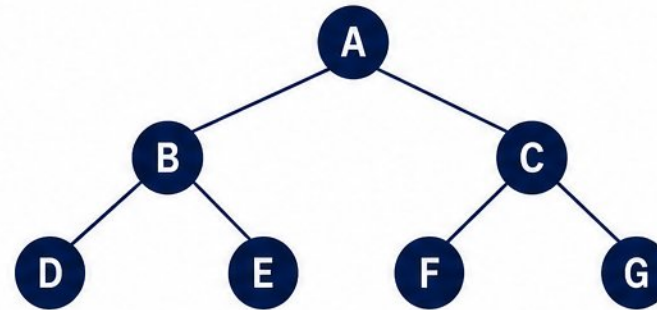


- The same tree, different secondary structure → completely different traversal order.

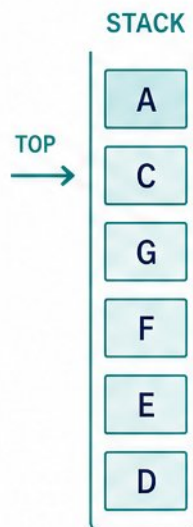


- **Core principle:** choose the structure that enforces your required processing order.

THE SAME TREE



DFS (Depth-First Search)



DFS Traversal Order

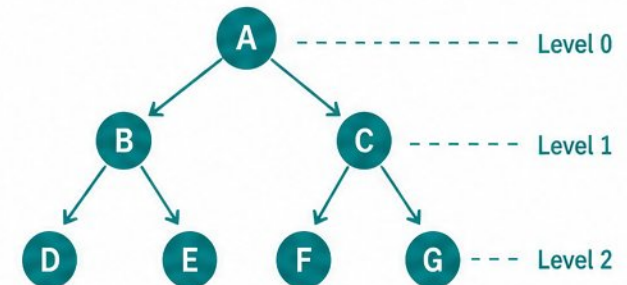


Explores as deep as possible before backtracking.

BFS (Breadth-First Search)

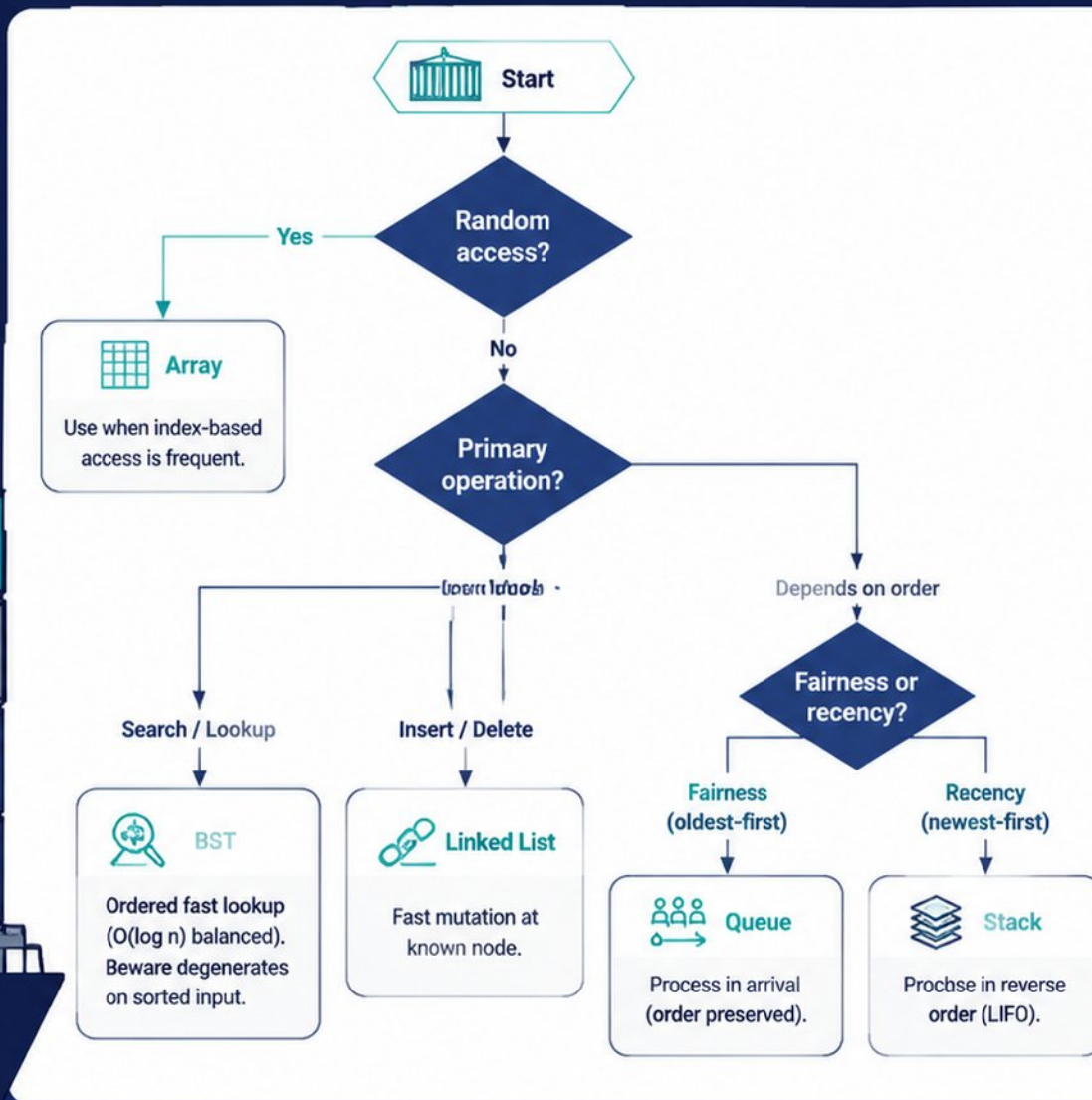


BFS Traversal Order



Explores level by level, guaranteeing shortest paths.

Choosing the Right Data Structure: A Practical Guide



- Identify the dominant operation: random access, search, insert, delete, or ordering?



- Ask: fairness (oldest-first) → Queue; recency (newest-first) → Stack.



- Array: fast reads, cache-friendly. Linked list: fast mutation at known node.



- BST: ordered fast lookup ($O(\log n)$ balanced), but degenerates on sorted input.



- Combine structures: LRU cache pairs a hash map with a doubly linked list.

Key Takeaways and Next Steps

- - Data structures optimize access; the right choice depends on your dominant operations
- - Covered: linear collections, stacks, queues, and trees
- - LIFO and FIFO are guarantees about behavior, not limitations
- - Practice: implement each structure in your language using pseudocode models
- - Up next: hash tables, graphs, and priority queues



Array



Linked List



Stack



Queue



Tree

NEXT STOPS →



Hash Tables →



Graphs →



Priority Queues →

PersonWise.

PersonWise • AI digital-human course platform



Scan the code or click anywhere to watch this course live, with voice Q&A

personwise.ai/t/42e2de25/public