Memory Representation & Introduction to Abstract Data Types (ADT)

Understanding the foundation of efficient programming through memory organisation and abstract thinking

Why Memory Representation Matters

Performance Impact

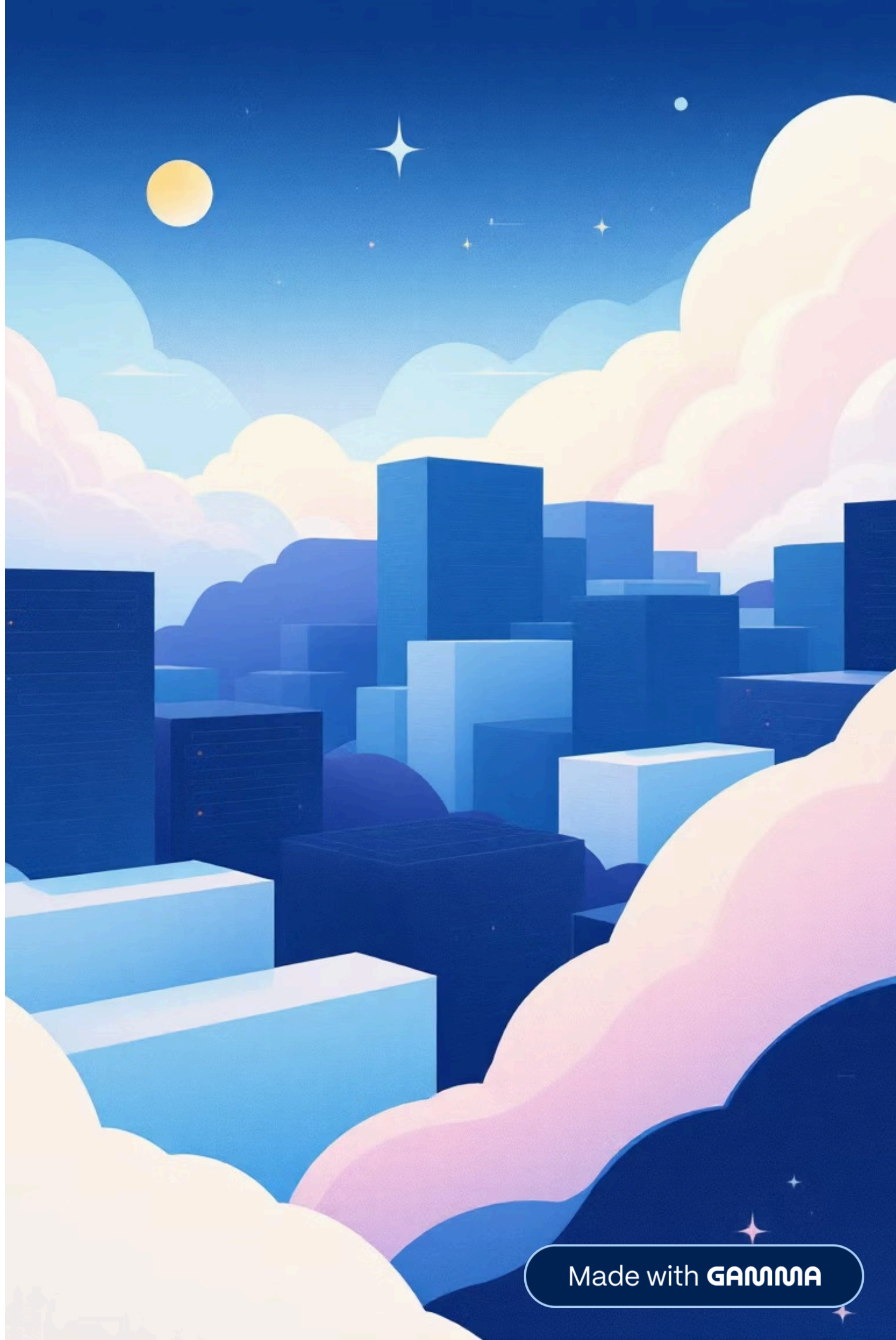
Memory layout directly affects access speed and resource utilisation in your programmes

Programme Correctness

Understanding memory prevents bugs and ensures reliable code execution

Efficiency Gains

Contiguous memory storage enables lightning-fast array element access



What is an Abstract Data Type (ADT)?

01

Defines Operations

Specifies *what* operations are possible on data, not *how* they work internally

02

Creates Abstraction

Provides a conceptual layer that hides complex internal memory representation details

03

Enables Flexibility

Stack ADT supports push/pop operations whether using arrays or linked lists underneath

Key Features of ADTs

Abstraction

Users interact with clean operations rather than messy internal structures

Modularity

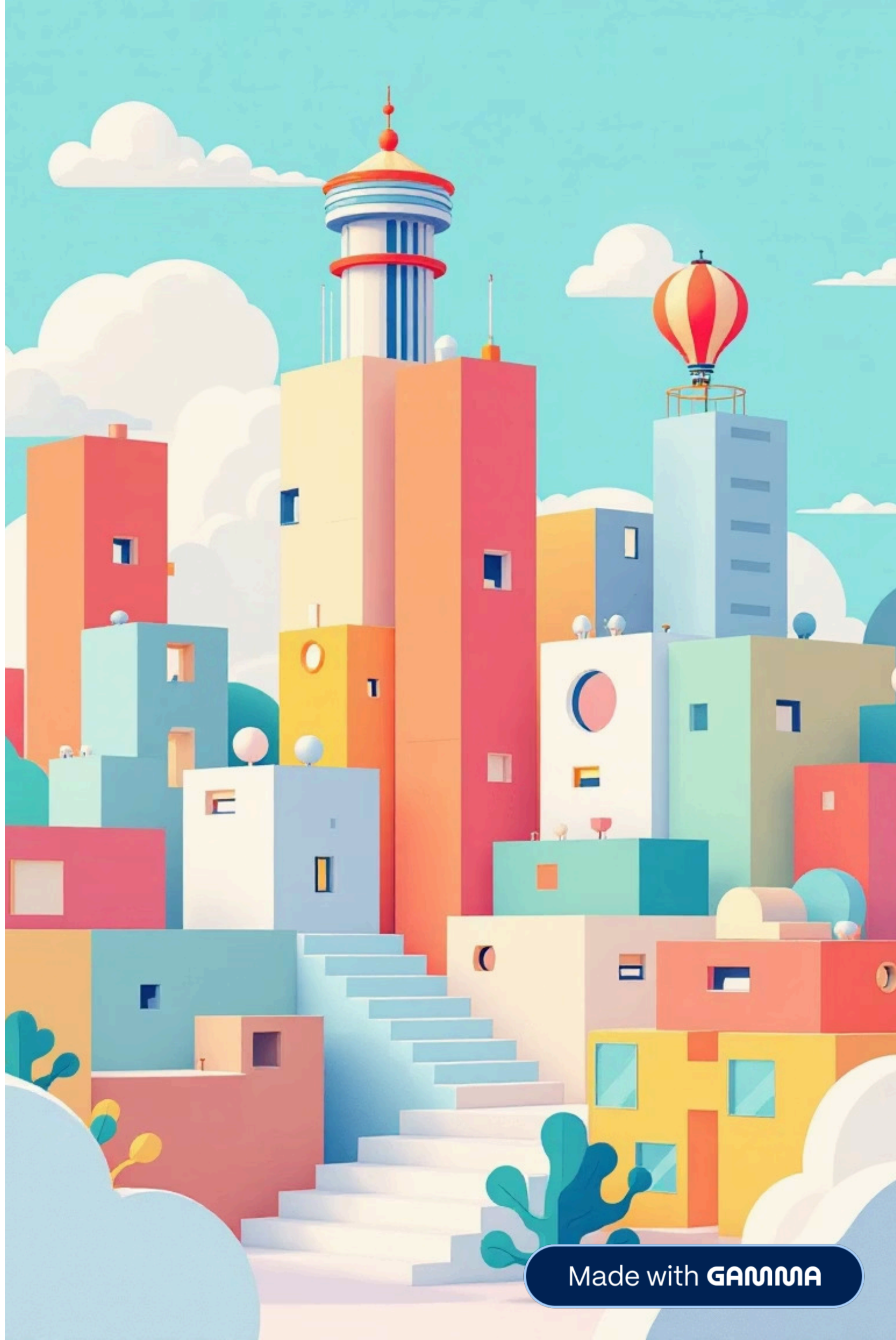
ADTs combine seamlessly to construct sophisticated, complex data structures

Encapsulation

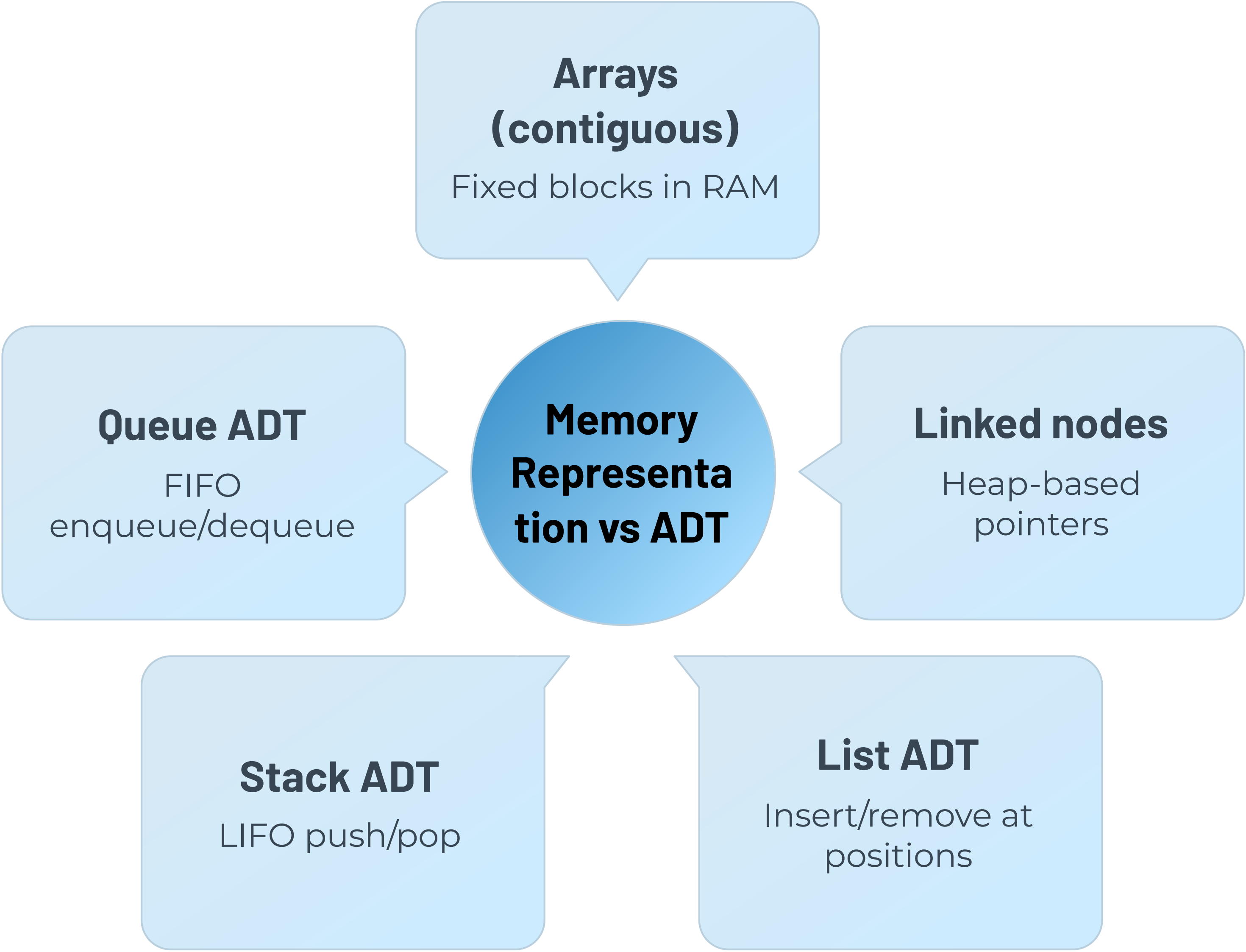
Internal data and implementation details remain completely hidden from users

Implementation Independence

Multiple implementations possible without changing the external interface



Memory Representation vs ADT



Memory Representation vs ADT

Arrays (contiguous)

Fixed blocks in RAM

Linked nodes

Heap-based pointers

List ADT

Insert/remove at positions

Stack ADT

LIFO push/pop

Queue ADT

FIFO enqueue/dequeue

1

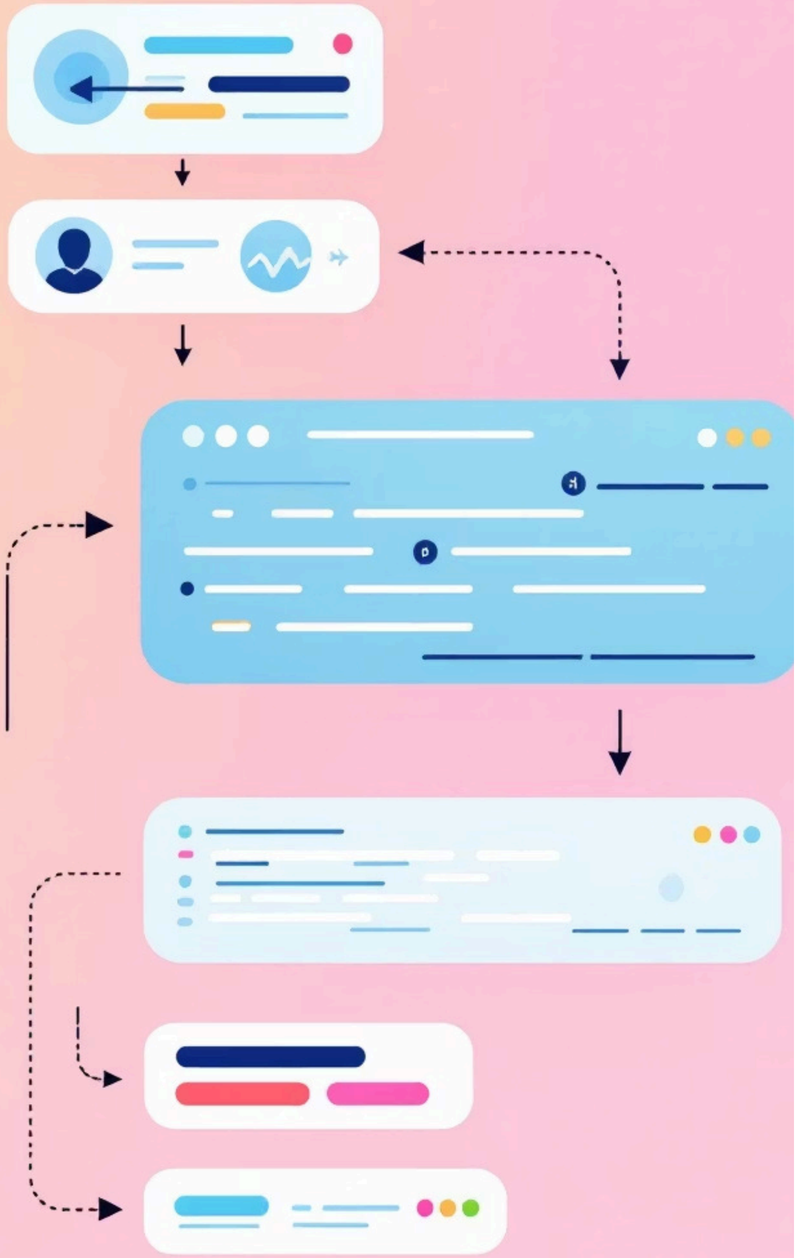
Physical Layer

Memory representation defines how data is actually stored in RAM - arrays, linked nodes, contiguous blocks

2

Logical Layer

ADTs specify behaviour and operations like push, pop, enqueue without exposing implementation



Common Abstract Data Types

List ADT

Ordered collection supporting insert, delete, and traverse operations for flexible data management

Stack ADT

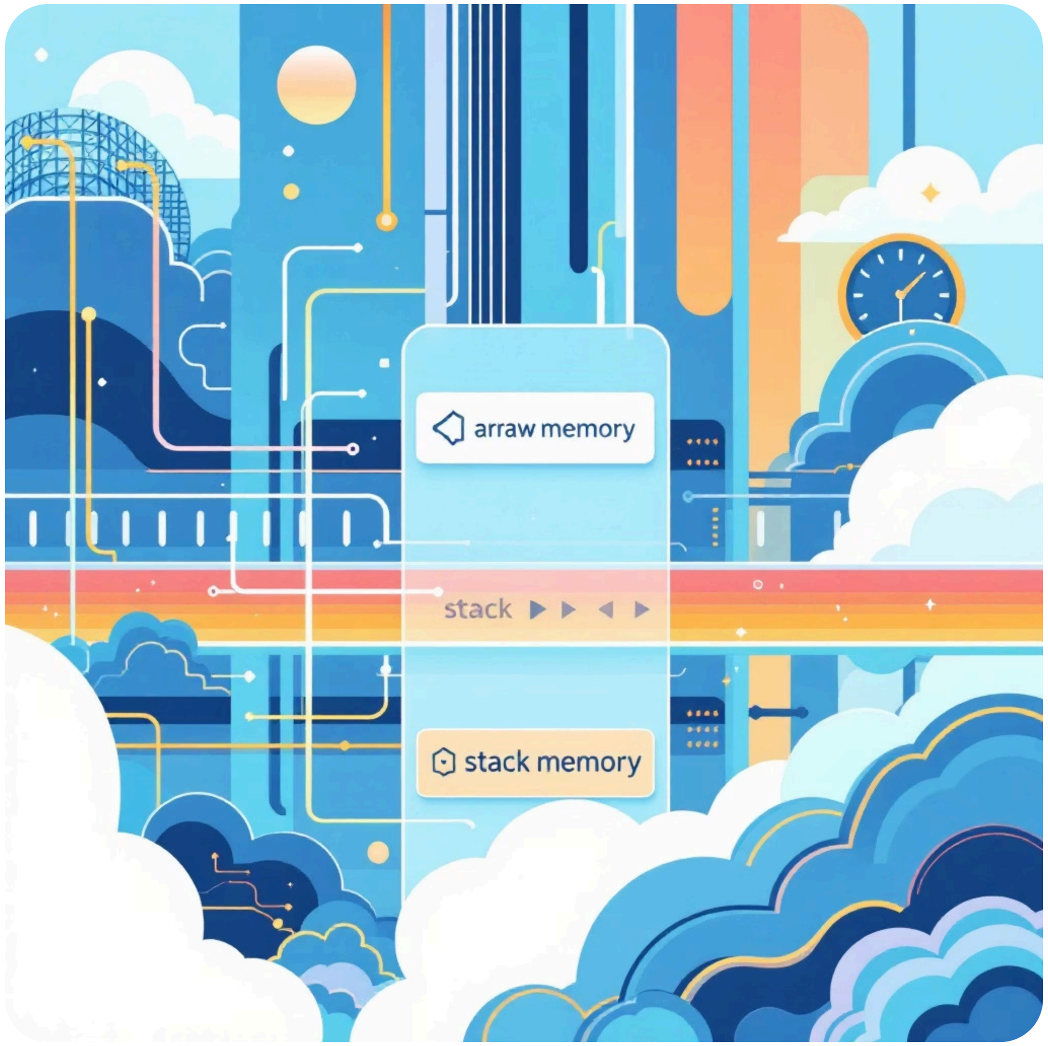
LIFO (Last In, First Out) structure with push, pop, and peek operations - like a stack of plates

Queue ADT

FIFO (First In, First Out) structure with enqueue, dequeue, and front operations - like a checkout queue

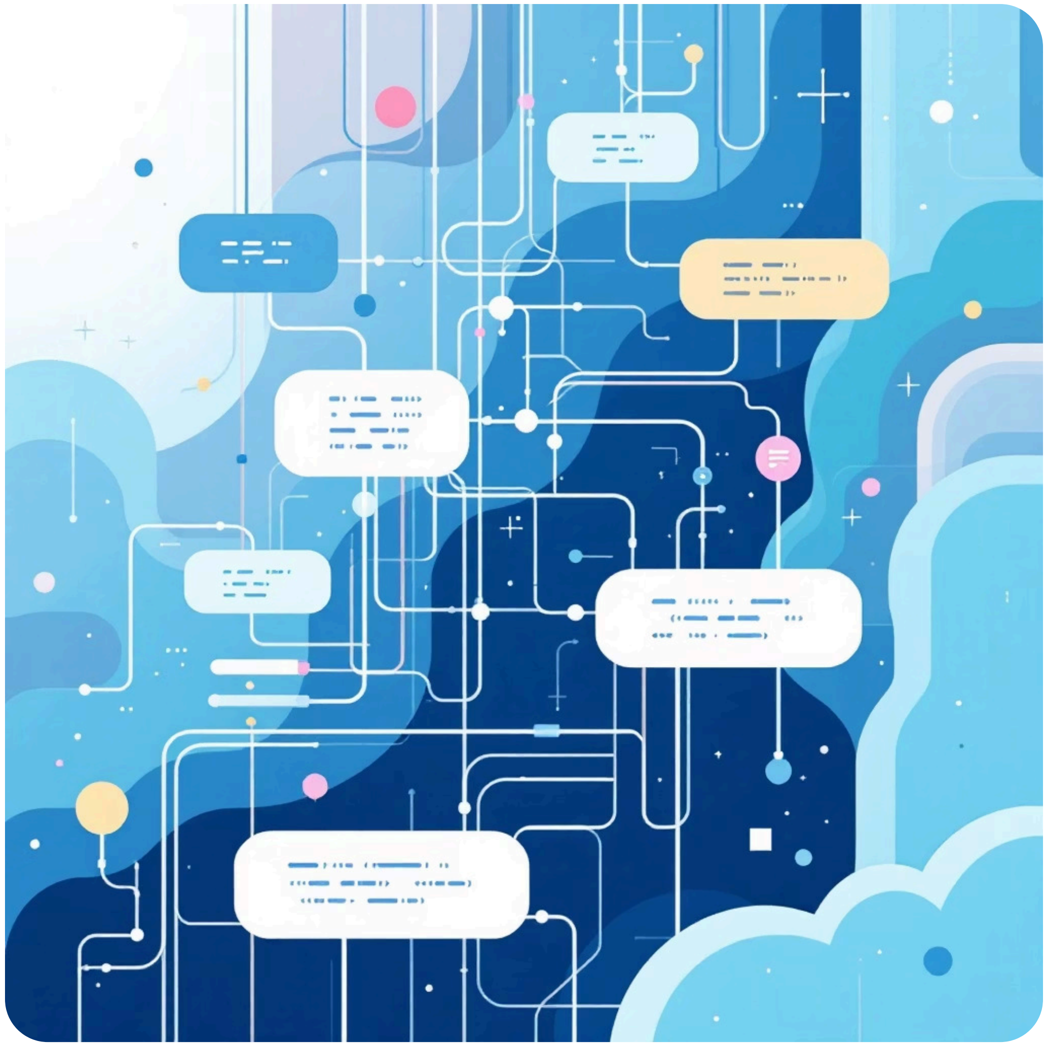
Stack Implementation Variants

Array-Based Stack



- Fixed size allocation
- Contiguous memory layout
- Fast random access
- Memory efficient

Linked-List Stack



- Dynamic size flexibility
- Nodes linked by pointers
- Memory allocated on demand
- No size limitations

Both implementations provide identical push/pop interfaces despite completely different memory layouts underneath

Stack ADT in Practice

```
// Stack ADT Interface - Implementation Independent
Stack myStack = new Stack()

// Push operations
myStack.push(10)
myStack.push(20)
myStack.push(30)

// Pop operations
value = myStack.pop()    // Returns 30
value = myStack.pop()    // Returns 20

// Peek without removing
top = myStack.peek()     // Returns 10

// Check if empty
isEmpty = myStack.empty() // Returns false
```

The beauty of ADTs: identical code works regardless of whether the stack uses arrays or linked lists internally