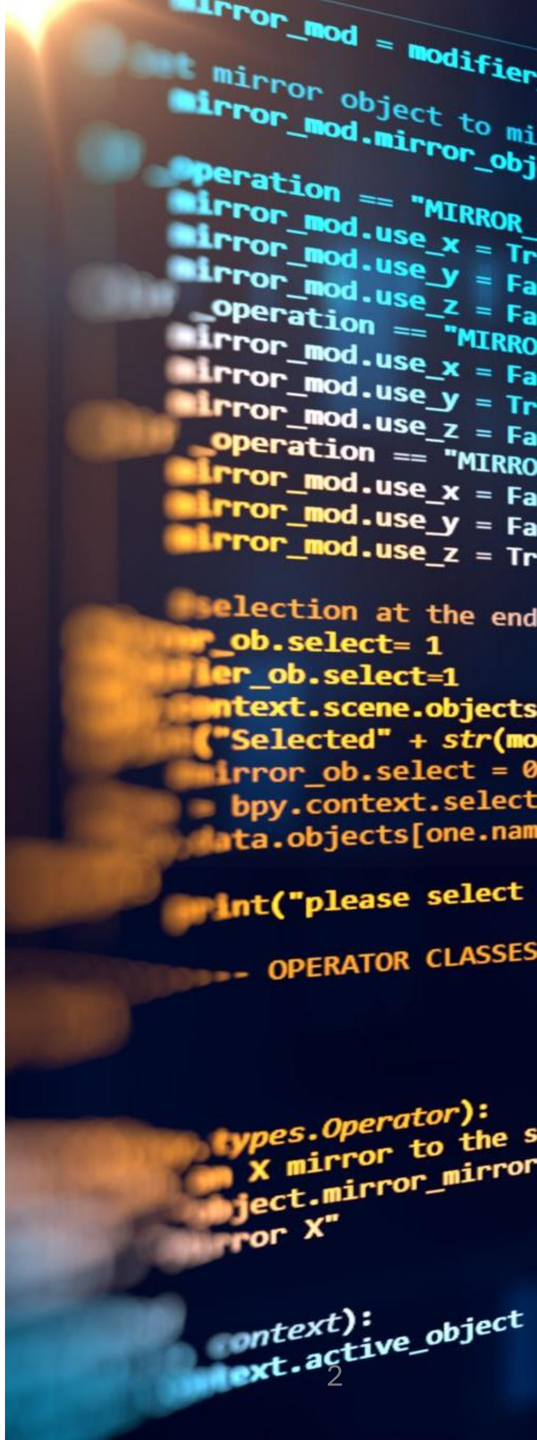
Please use the following QR code to check in and record your attendance.

CS 1037
Fundamentals of Computer Science II

Abstract Data Structure (cont.)

Ahmed Ibrahim



Collections

- **Collection**: a group of items we wish to treat as a **conceptual unit**.
- Choosing the **right** collection type can significantly impact a solution's performance and clarity. Properly choosing a collection for a given problem can improve the solution's efficiency and simplicity.
- **In programming**, a conceptual unit refers to a single, unified entity that groups **multiple items** or **elements** together, treating them as <u>one coherent object or collection</u>: Abstract Data Type (**ADT**).

Abstract Data Structure

An **Abstract Data Type (ADT)** is a model where the data type is defined by its **behaviour** (the operations that can be performed) rather than by its **implementation**.

ADTs describe **what data is stored** and **what operations can be done** on that data without specifying how these operations are **implemented internally**.

- **Key Characteristics of ADTs:**
  - **Encapsulation**: The internal workings of an ADT (such as data structures or algorithms used) are hidden from the user.
  - **Operations**: ADTs are defined by the operations that can be performed on them, such as insertion, deletion, or access of elements.
  - **Independence from Implementation**: ADTs can be implemented using various data structures, but the user is unaware of these details.

Example

- Imagine you're designing a system that tracks unique customer IDs for a **rewards program**.

You need to:

  - Ensure each customer ID is unique.
  - Frequently check if a particular customer ID is already part of the program.
  - Occasionally, iterate over all customer IDs in the order they were added.

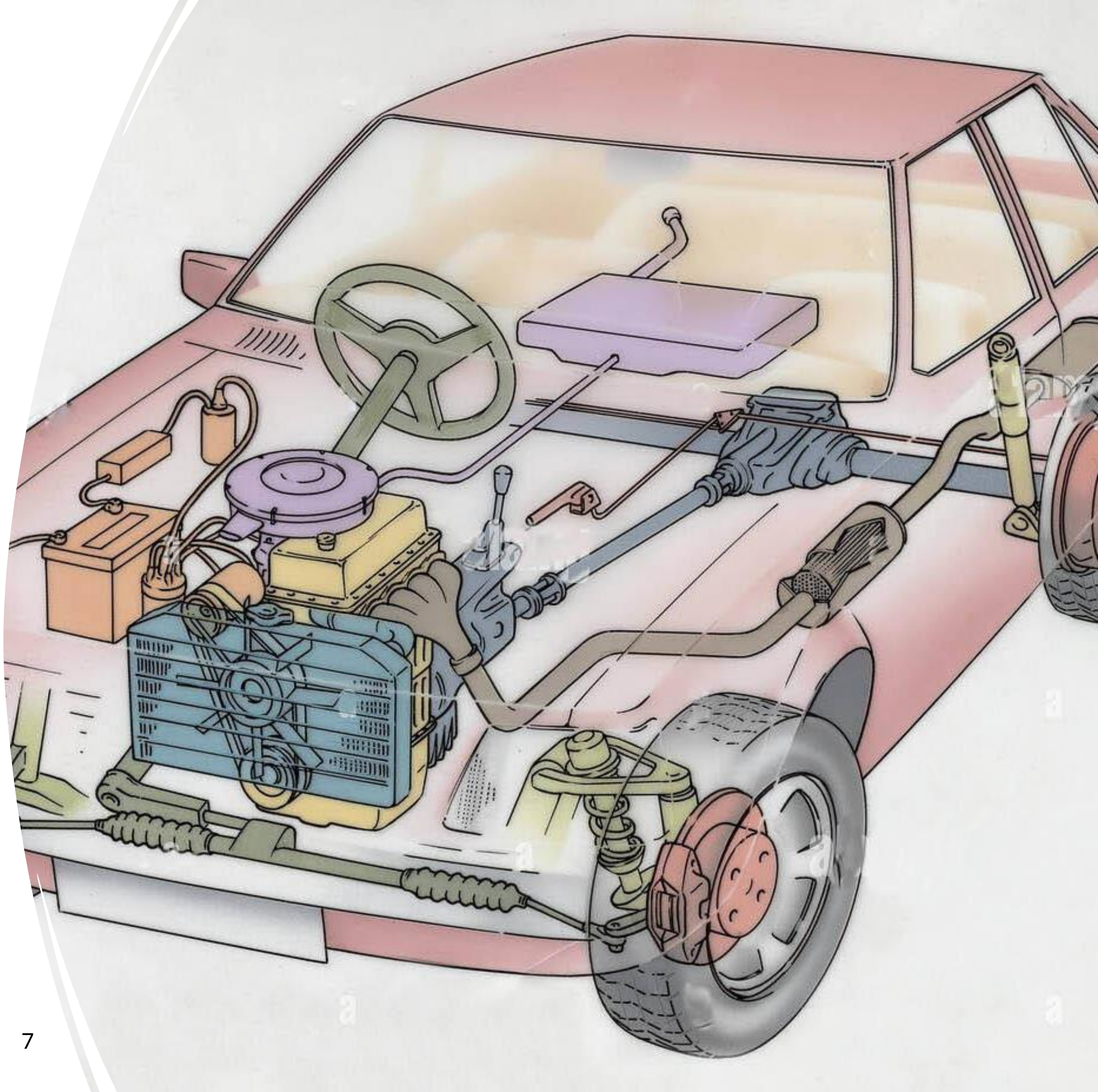
- Ideas:
  - A collection that cannot contain duplicate elements: **Set**ADT
  - A collection for holding elements before processing: **Queue**ADT
  - And much more ...

Abstract Data Type (ADT) cont.

- ADTs define the **behaviour** and **properties** of a collection, focusing on **what** operations it should support, such as **adding** and **removing** elements from a collection, but <u>leaving out **how** these operations are performed</u>.
- ADTs do not specify implementation details like:
  - **Which data structure** is used to store the elements?
  - **Where** elements are added and **how** they are reorganized when removed.
- This **abstraction** allows ADTs to act as a **blueprint**, letting us select the **right structure** for our program's requirements while leaving room for flexible implementations tailored to specific use cases and performance needs.

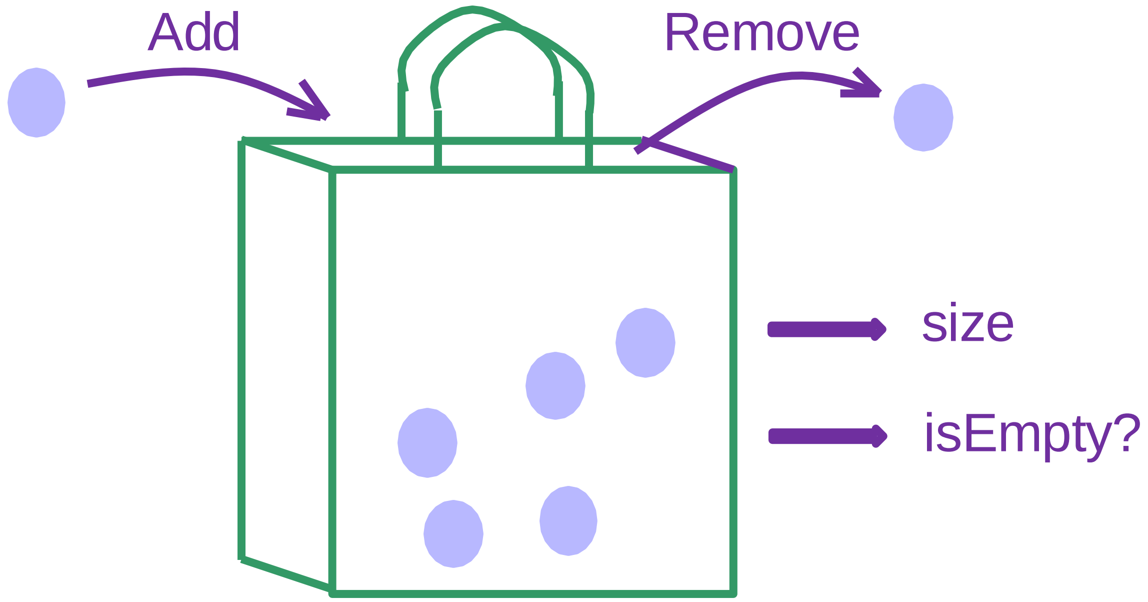
Abstraction

- Abstraction separates the purpose of an entity from its implementation or how it works
- Example in real life: a car (**we do not have to know how an engine works to drive a car**)



Example: The Bag ADT

- Here's an example of an ADT design process for a Bag ADT.
- A bag is a collection that holds multiple data items with **no specific order** or **unique constraints**. It provides the following **operations**:
  - **Add**: Insert a new data item into the bag.
  - **Remove**: Delete a specified data item from the bag.
  - **Count**: Returns the number of occurrences of the element x in the bag.



The Bag ADT: Usage Scenarios

- Bags are used in various applications where **duplicates** are allowed, and the order of elements does not matter. Some examples include:
  - **Inventory Management**: In a game or store inventory system, a bag can hold multiple instances of the same item (e.g., multiple potions or books).
  - **Word Frequency Counting**: A bag can be used to count **occurrences** of words in a document without caring about the order.

Common ADTs and Their Use Cases

ADT	Characteristics	When to Use
List	Ordered, allows duplicates	When order matters or when accessing by index
Set	Unordered, no duplicates	When uniqueness of items is required
Queue	FIFO (First-In-First-Out)	For tasks that need sequential processing
Stack	LIFO (Last-In-First-Out)	For tasks that require reversing or backtracking
Map	Key-value pairs, unique keys	When mapping relationships or fast lookups are needed
Bag	Unordered, allows duplicates	When collecting items with no order or uniqueness requirements

Examples of ADT Operations

- **Queue ADT:** A queue provides **enqueue**() and **dequeue**() operations, but its internal representation can be an array, linked list, or any other structure.
- **Stack ADT:** A stack allows operations like **push**(), **pop**(), and **peek**(). However, how the stack is implemented (using an array or a linked list) is abstracted.
- **List ADT:** A list provides operations like **insertion**, **deletion**, and **access of elements** by index, but the underlying storage mechanism (such as a dynamic array or linked list) is hidden.
- **Set ADT:** A set supports operations like **adding**, **removing**, and **checking for membership** of elements without exposing how the elements are stored.

Choosing the Right ADT for a Problem

- **Efficiency**: Choosing the right ADT optimizes memory and processing power.
- **Simplicity**: The right ADT improves code readability, maintenance, and alignment with problem requirements.

Question!

You are designing a software system for a library to manage book inventory and user requests. The system must meet the following requirements:

1. Track each book's unique ISBN number and allow quick lookups by ISBN.
2. Allow multiple copies of the same book in the inventory.
3. Process book requests from users in the order they are received.

Which combination of ADTs would be most suitable to meet these requirements?

A) Use a List to store all books and a Queue to process user requests.

B) Use a Set to store unique ISBNs and a Queue to manage user requests.

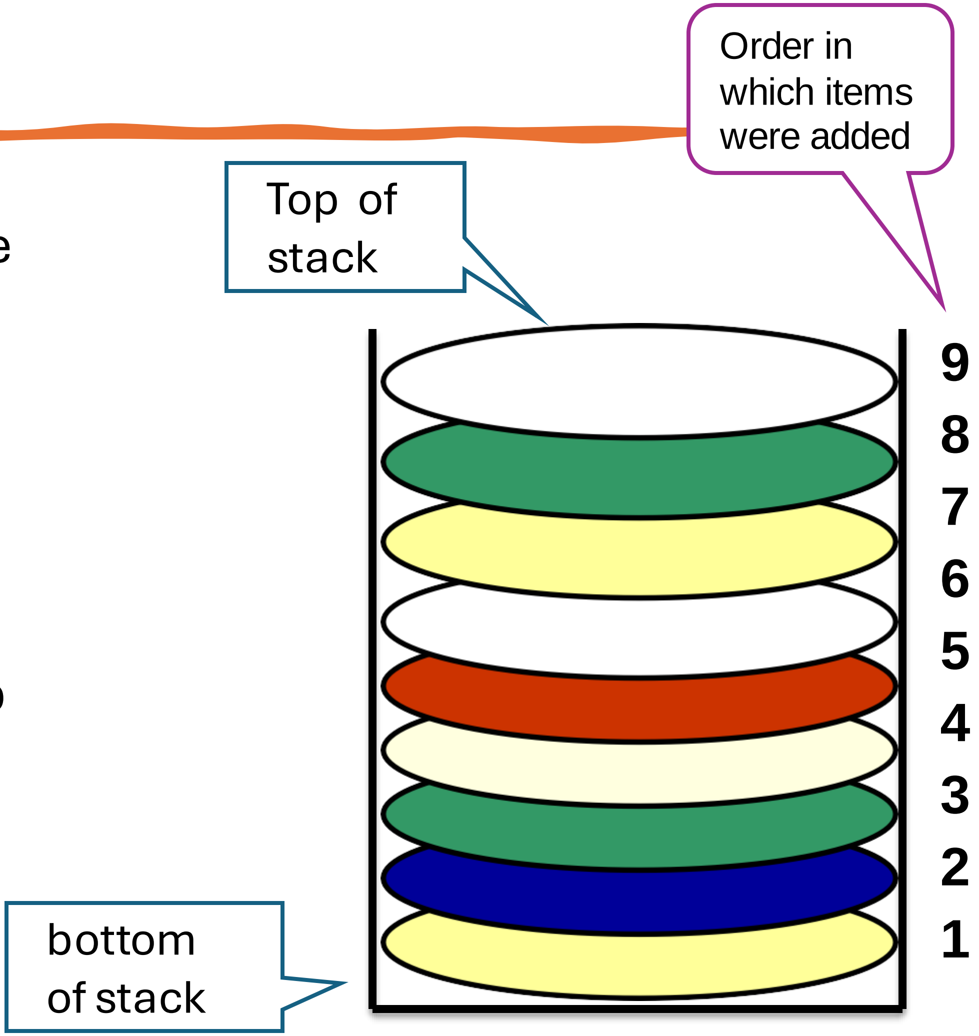
C) Use a Map for ISBN lookups, a Bag for multiple copies, and a Queue for user requests.

D) Use a Stack to manage book inventory and a Queue to handle user requests.

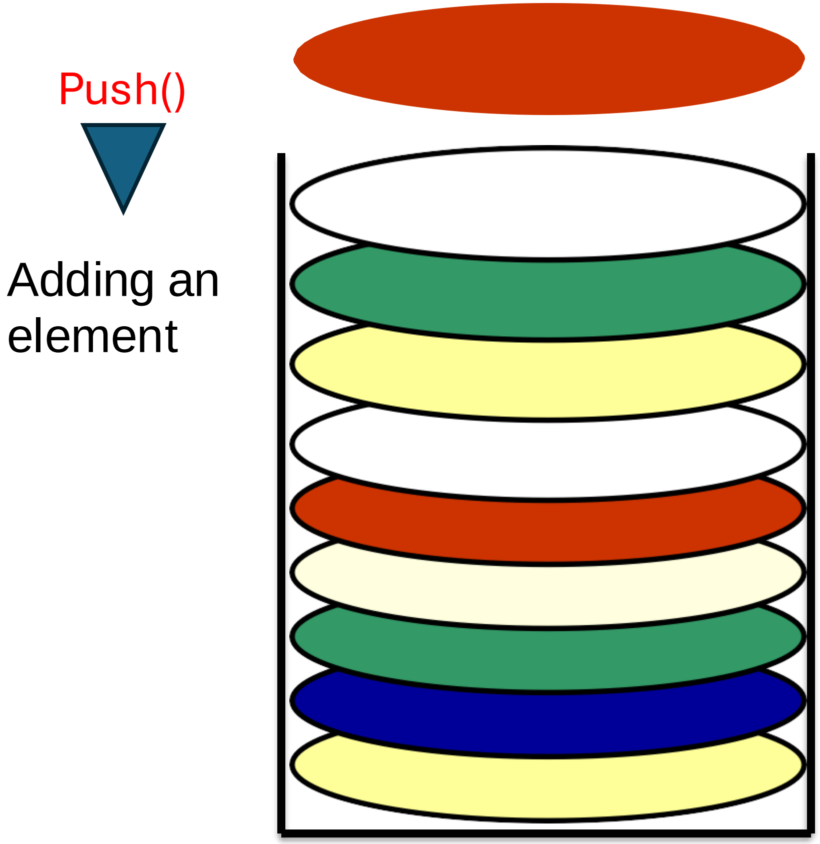
The Stack ADT

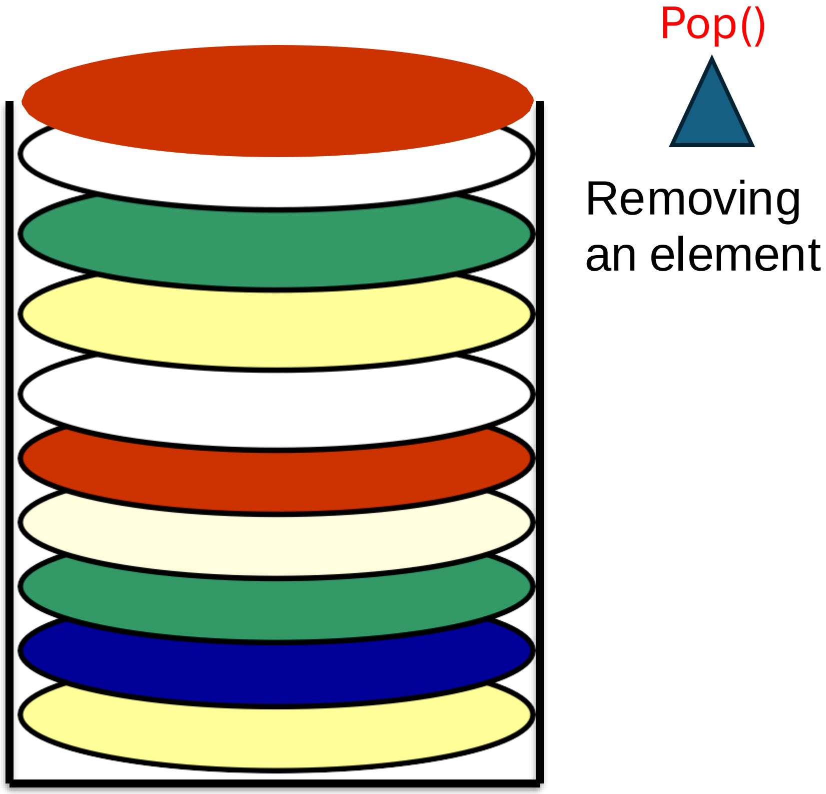
The Stack ADT

- A stack is a collection of items organized so that only the item at the top is accessible.
- You can visualize it as a container open at just one end, with items stacked on top of each other.
- Each time you add an item, it is added to the previous ones, and when you remove an item, only the one on top is accessible.
- Stack is a LIFO (Last In, First Out) structure



Stack Operations: Push and Pop





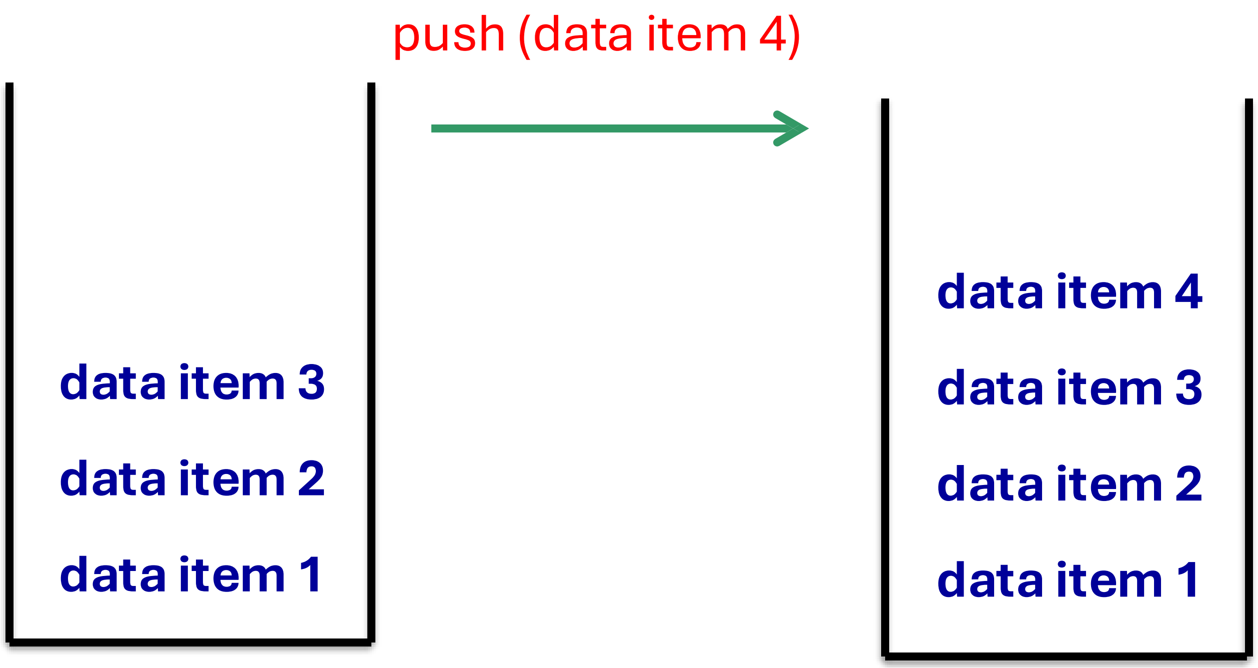
Stack (ADT) cont.

- A stack's characteristic is the **Last-In-First-Out** (or simply **LIFO**), meaning it pops the latest pushed element.
- A stack is **empty** if there is no element in the stack.
- The **length** of a stack is the number of elements in the stack.

Stack ADT Operations

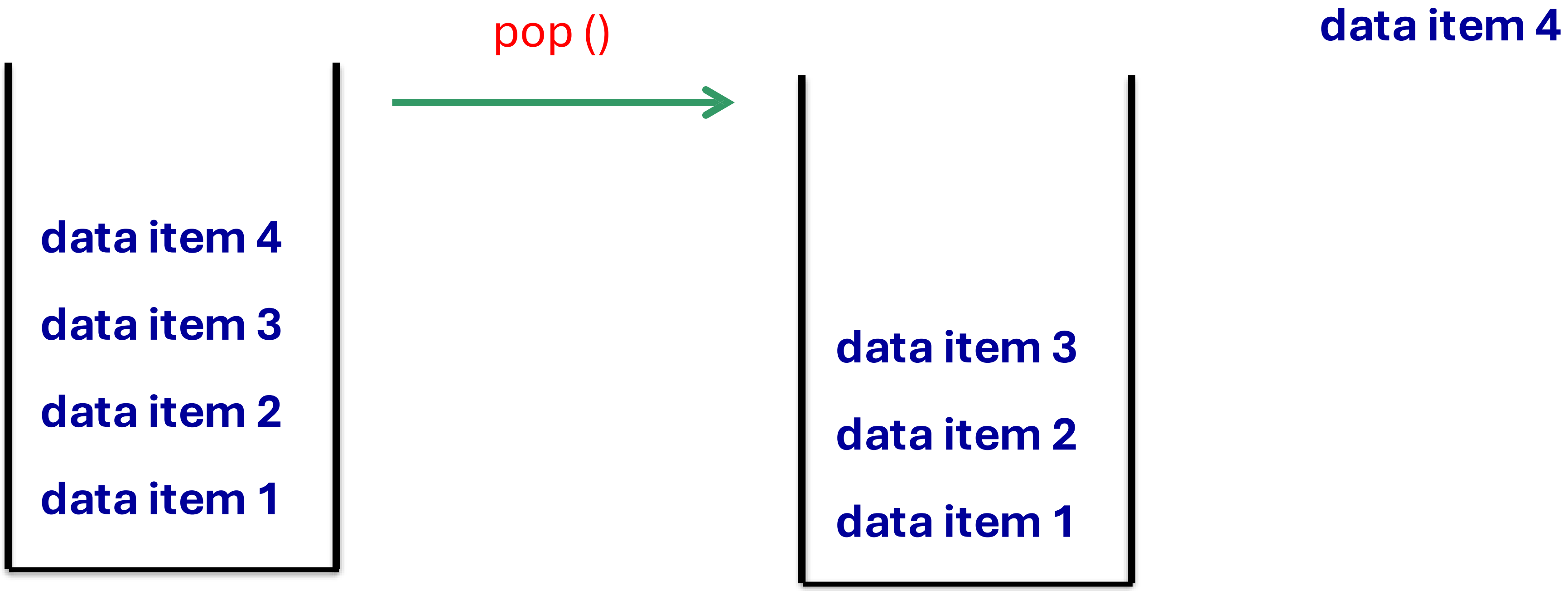
Stack Operations: Push

- Push: adds an element at the top of the stack



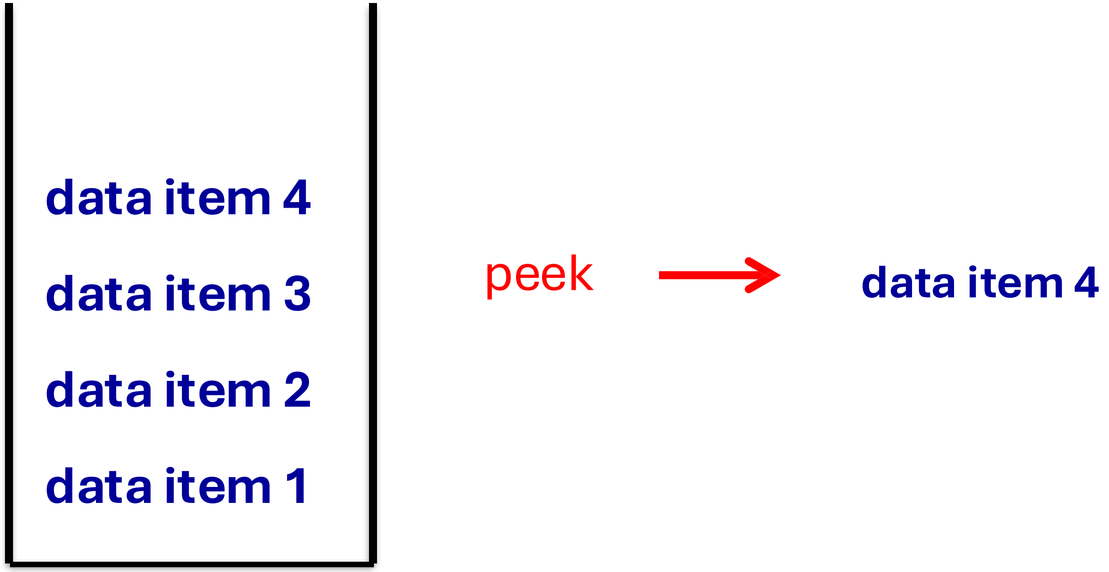
Stack Operations: Pop

- Pop: removes the element at the top of the stack



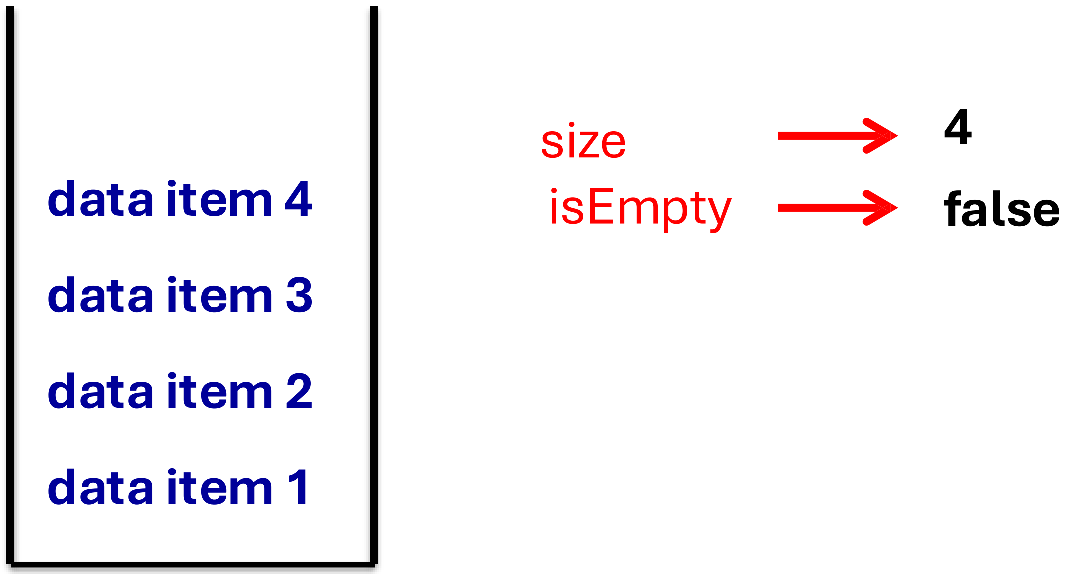
Stack Operations: Peek

- Peek: returns the element at the top of the stack without removing it



More Stack Operations

- size: number of elements in the stack
- isEmpty: true if the stack is empty



Stack Data Structure Using Arrays

- A stack is a linear data structure that follows the Last In, First Out (LIFO) principle.
- The following is an array-based implementation of a stackADT in C. The structure Stack contains three fields:

```c
typedef struct {
int top;
int capacity;
int *array;
} Stack;
```

Keeps track of the index of the top element

Defines the maximum number of elements the stack can hold

A pointer to an integer array that stores the stack elements.

Creating a Stack Using Arrays in C

The following function **initializes** a stack structure with a specified **capacity** using arrays.

Here's a breakdown of how it works:

defines the maximum size of the stack

```
Stack *createStack(int capacity) {
  Stack *s = (Stack *)malloc(sizeof(Stack));
  s->capacity = capacity;
  s->top = -1;
  s->array = (int *)malloc(s->capacity * sizeof(int));
return s; }
```

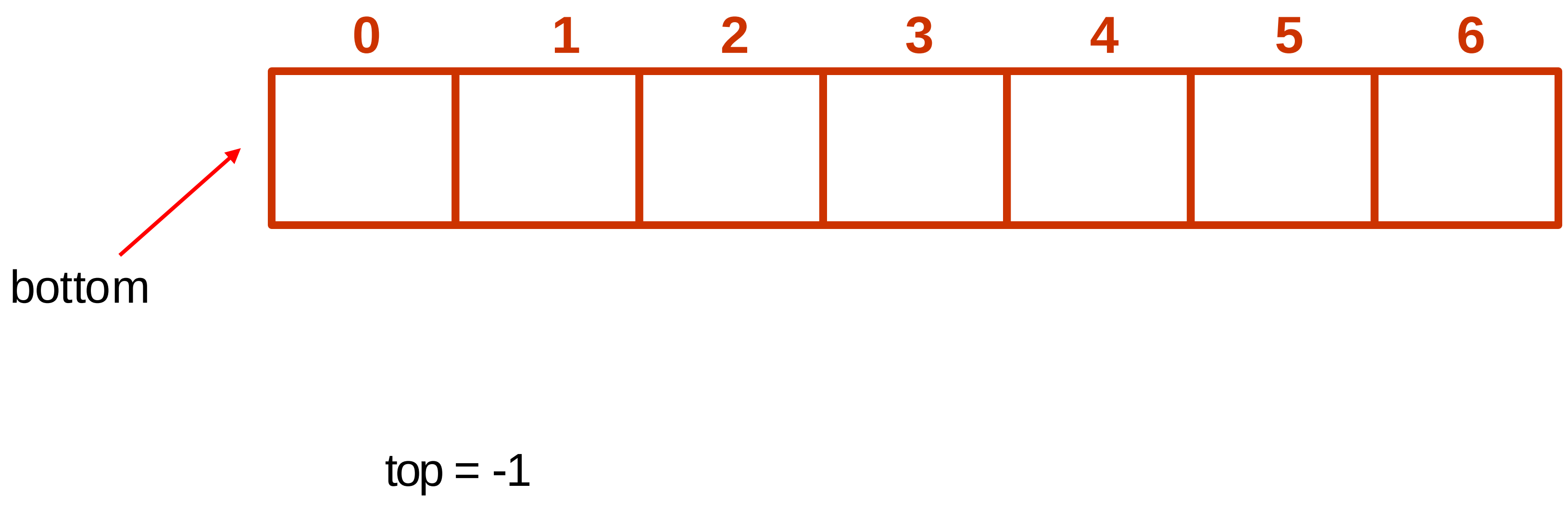
Allocates memory for the stack structure itself

indicate the stack is initially empty

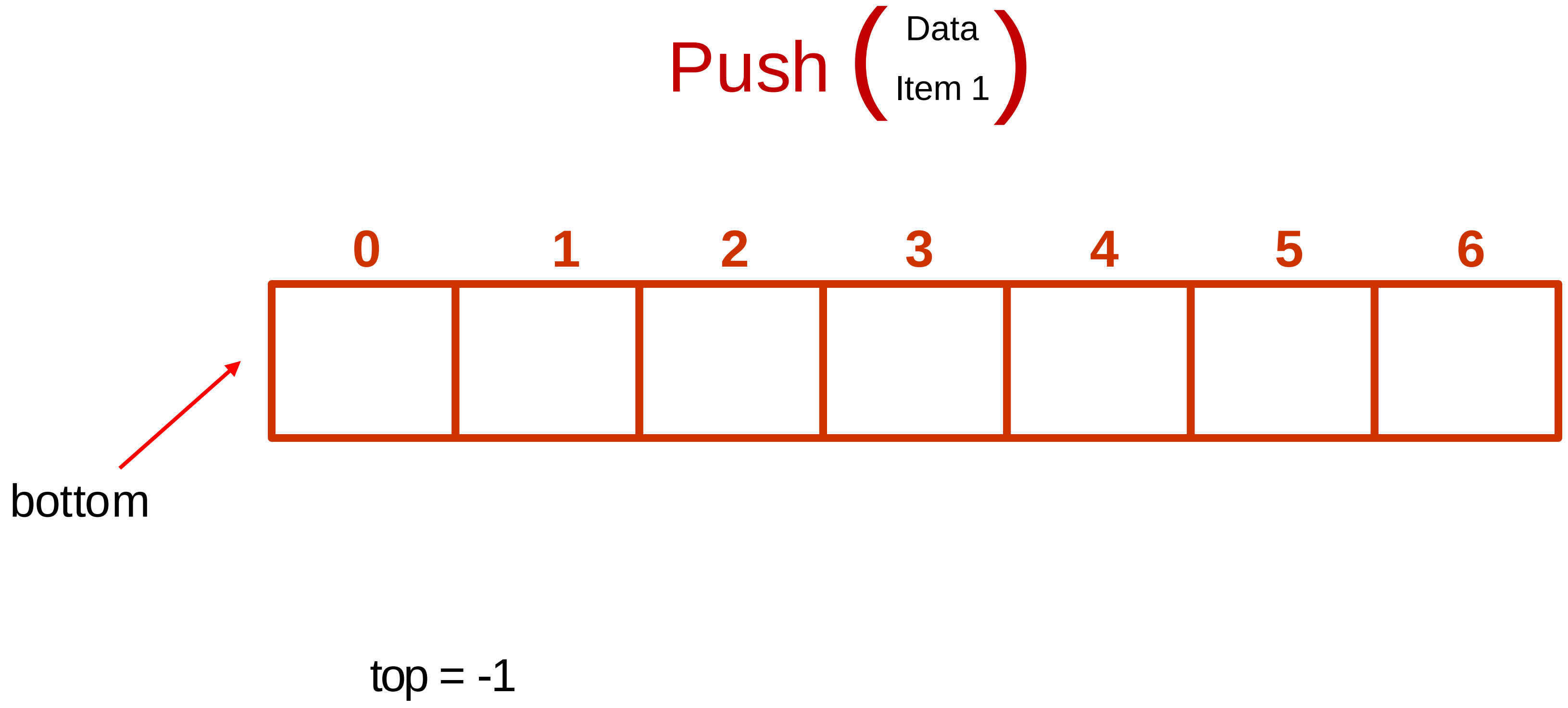
Sets up an array within the stack to hold elements

The Array-Based Stack

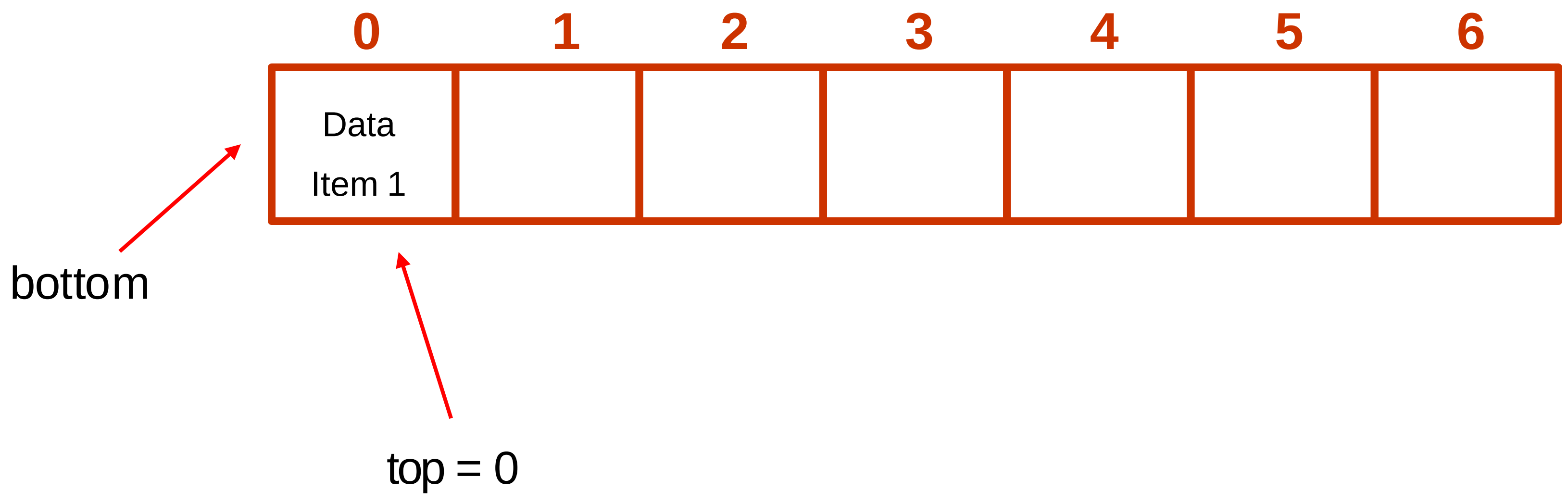
- Initial state of the Stack



Pushing a Data Item

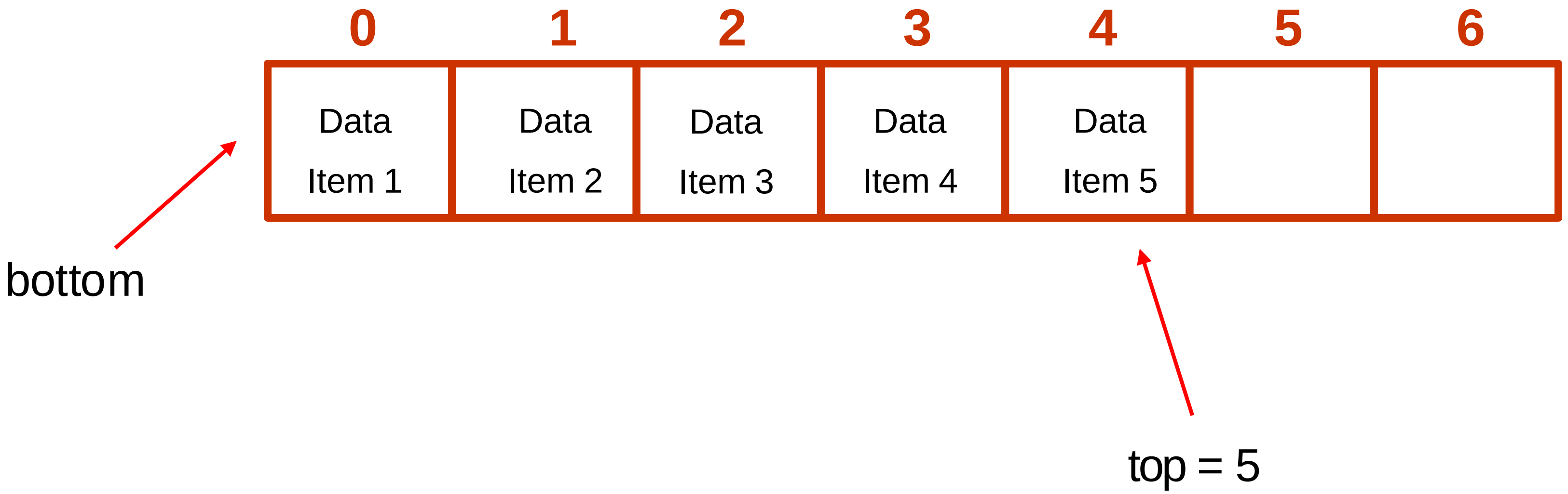


Stack After Adding a New Data Item



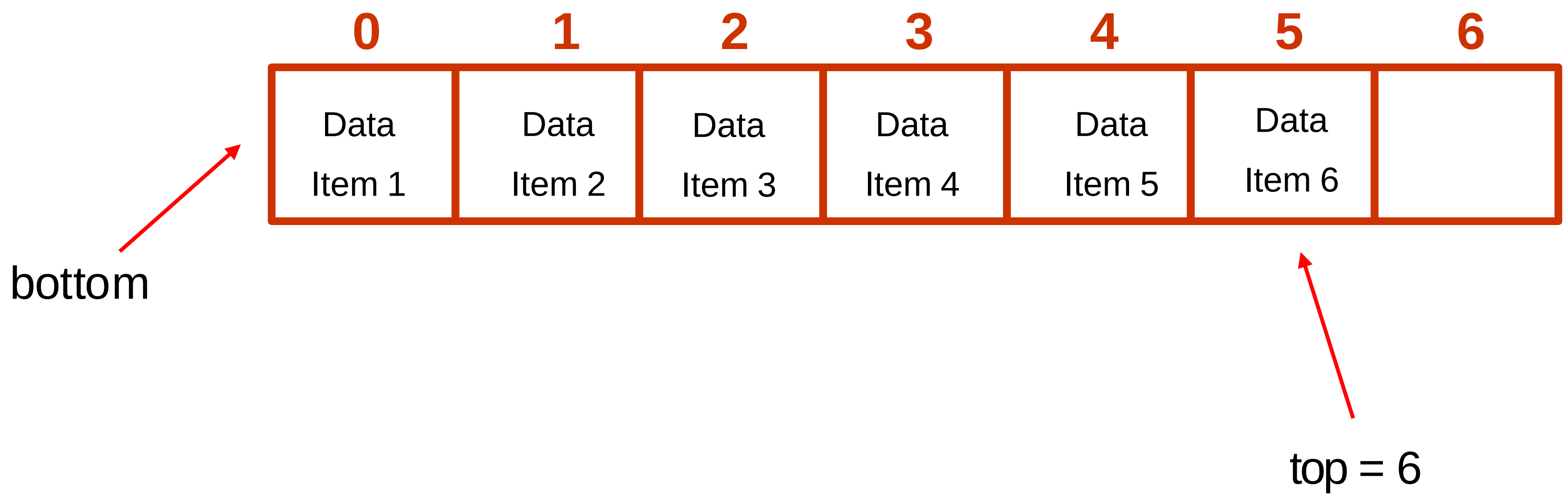
Filling Up the Stack with Data

- After Adding 5 data items
- We do not need to keep track of the bottom, as the bottom of the stack is always at index 0

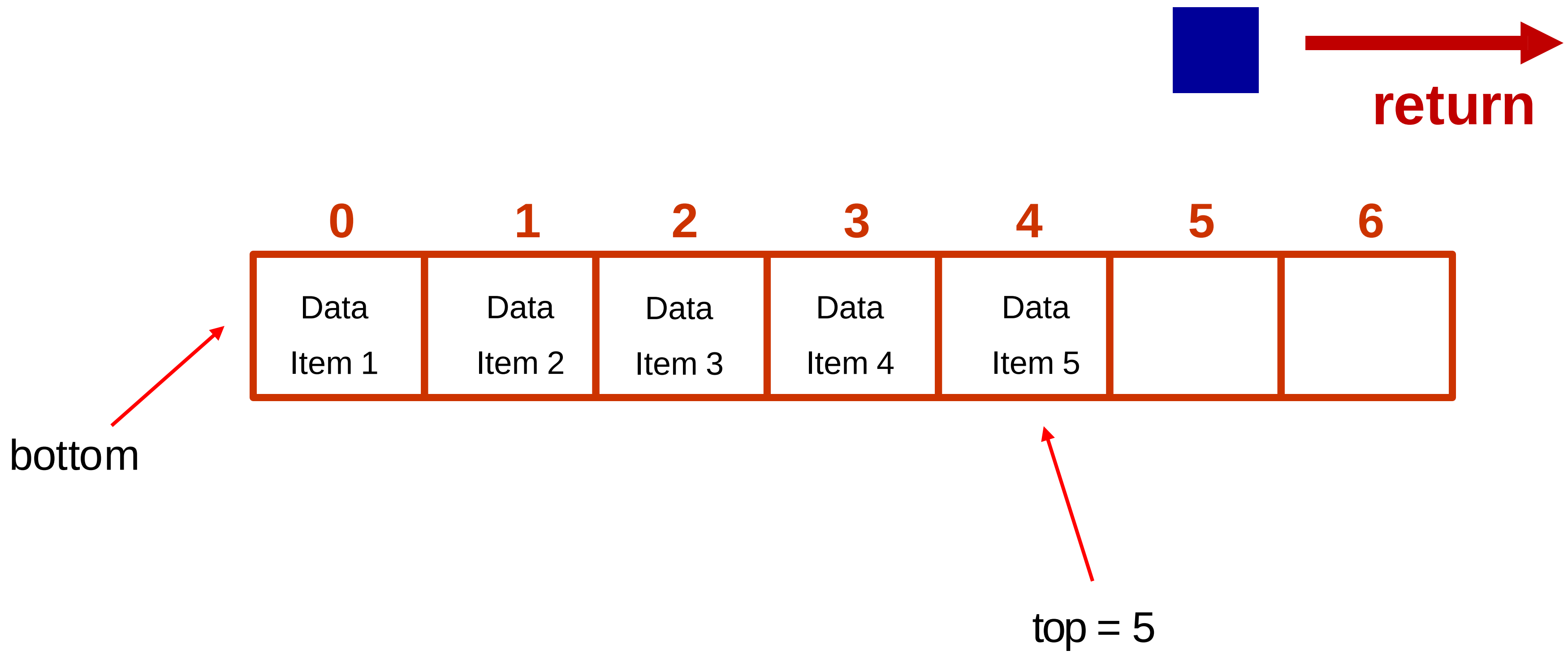


Removing an Element from the Stack

Pop ()



Returning the Popped Element



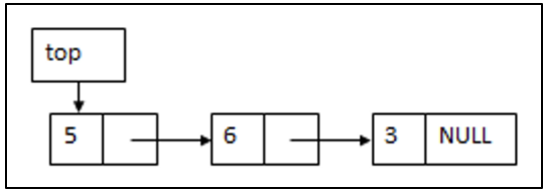
Linked List Stack

- Using a singly linked list to implement a stack, a pointer points to the first node.

A member to store the integer data for each node

```
typedef struct Node {
int data;
struct Node* next;
} Node;
```

A pointer to the next node in the stack.



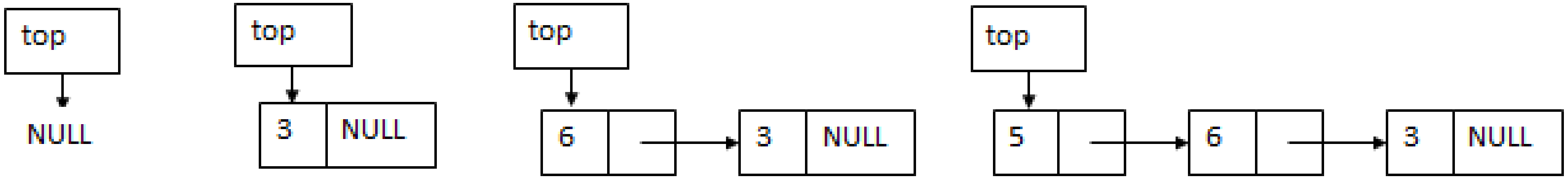
Linked List Stack with 3 data items

```
typedef struct {
Node* top;
} Stack;
```

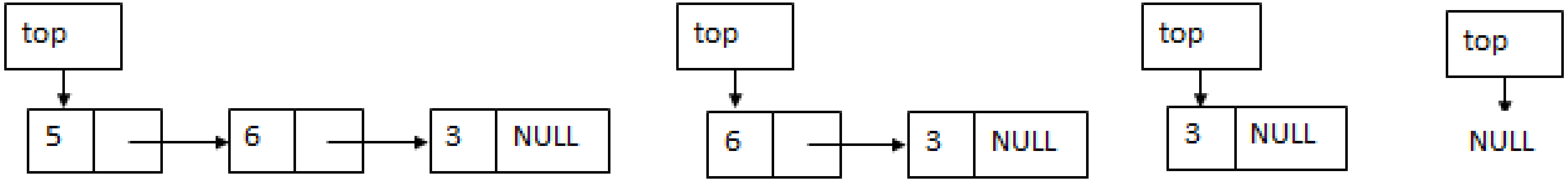
A pointer to the top node in the stack

- The stack is empty if top = NULL. The stack operations are done as follows.

Linked List Stack Operations: Push, Pop, and Peek



Empty stack push 3 push 6 push 5

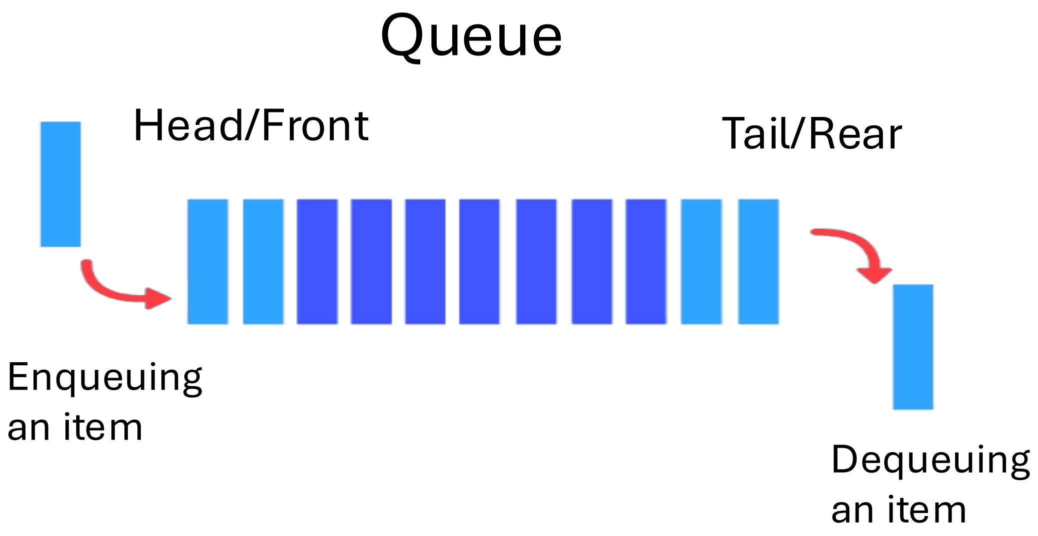


peek, return 5 pop pop pop empty stack

The Queues ADT

The Queues ADT

- The concepts of **queues** consist of the **abstract queue** and **queue data structures** (the implementations of the abstract queue).
- The abstract queue's characteristic is the **First-In-First-Out** (or simply FIFO), which deletes the first element currently in the data structure.



- Example – **CPU Task Scheduling:** In round-robin scheduling, processes waiting to be executed are placed in a queue, and the CPU handles them in the order they arrive.

Properties of the Queue ADT

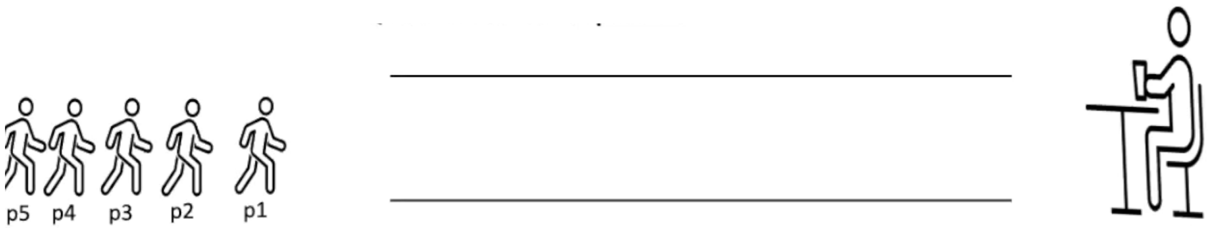
- The properties of a queue are as follows:
  - A queue is a **linear collection** of data elements with three main operations: enqueue, dequeue, and peek.
    - **Enqueue** adds an element to the back of the queue. The order of elements is based on the time they were added, with the earliest at the front and the latest at the back.
    - **Dequeue** removes the front element from the queue.
    - **Peek** retrieves the front element without removing it.
- **Note**: The dequeue and peek operations can be combined into one dequeue operation which returns the front element and removes the front from the queue.

Underflow vs. Overflow

- A queue is said to be empty if it does not contain an element. Deletion cannot be done when a queue is empty; such a situation is called **underflow**.
- The length of a queue is the number of elements in the queue. When the length reaches the maximum length that a queue is allowed, insertion can not be done, and such a situation is called **overflow**.

Queue ADT: Real-World Operations

Simulation of Real-World Scenario: Waiting List

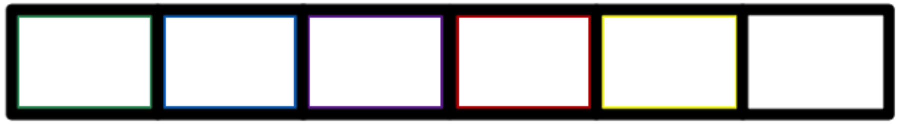


- A **queue data structure** is an implementation of the abstract queue.
- A queue can be implemented using an **array** or **linked list** representation, with two accessing variables, front and rear, representing the queue's front and rear (back) positions.

Source:
https://www.youtube.com/watch?app=desktop&v=HcB1P9sJZB4

Array-Based Queue Implementation

- A simple array queue is a queue implementation with an array representation. The front variable presents the front position where deletion is done, and the rear variable represents the rear position where deletion is done.
- A simple array queue is created by creating an array of MAX (given) size and front = -1, rear = -1.



front = -1,
rear = -1.

Step-by-Step Operations in an Array-Based Queue

index	0	1	2	3	4	5	6	7	8	9
a[i]	X	X	X	X	X	X	X	X	X	X

Empty queue: front =rear = -1

index	0	1	2	3	4	5	6	7	8	9
a[i]	6	X	X	X	X	X	X	X	X	X

Insert 6: front = rear = 0

index	0	1	2	3	4	5	6	7	8	9
a[i]	6	5	X	X	X	X	X	X	X	X

Insert 5: front=0, rear = 1

index	0	1	2	3	4	5	6	7	8	9
a[i]	6	5	4	X	X	X	X	X	X	X

Insert 4: front=0, rear = 2

index	0	1	2	3	4	5	6	7	8	9
a[i]	6	5	4	X	X	X	X	X	X	X

delete: front=1, rear = 2

Queue Data Structure Using Arrays

- A queue is a linear data structure that follows the First In, First Out (FIFO) principle.
- The following is an array-based implementation of a queue in C. The structure Queue contains three fields:

```c
typedef struct {
int front;
int rear;
int capacity;
int *array;
} Queue;
```

Keeps track of the front element in the queue

Keeps track of the last element in the queue

Defines the maximum number of elements the queue can hold

A pointer to an integer array that stores the queue elements

Creating a Queue Using Arrays in C

The following function **initializes** a stack structure with a specified **capacity** using arrays.

Here's a breakdown of how it works:

```
Queue *createQueue(int capacity) {
  Queue *q = (Queue *)malloc(sizeof(Queue));
  q->capacity = capacity;
  q->front = -1;
  q->rear = -1;
  q->array = (int *)malloc(q->capacity * sizeof(int));
return q;
}
```

Allocates memory for the queue structure itself

Defines the maximum size of the queue

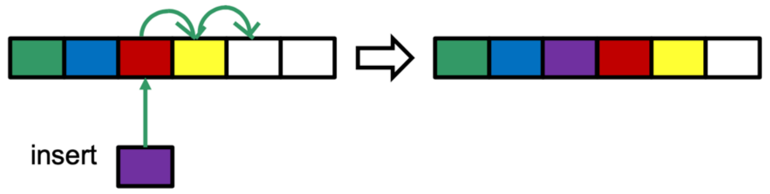
Indicates the queue is initially empty at the front

Indicates the queue is initially empty at the rear

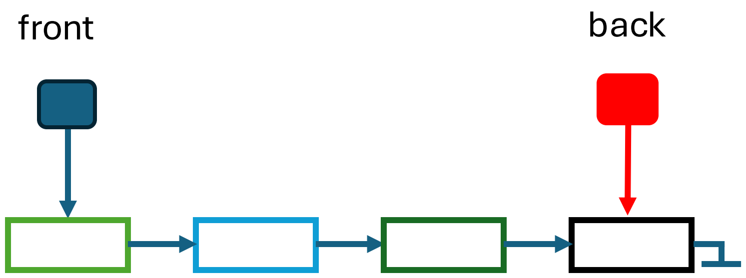
Sets up an array within the queue to hold elements

Linked List Queues

- The array queues have two drawbacks:
  1. The length of its array bounds the length of the queue.
  2. It wastes space if the length of a queue is much shorter than the length of the array.



- A **linked list queue** stores queue data values in a singly linked list and uses two pointers, front and rear, to represent the front and rear positions.



- A linked list queue is empty if both front and rear are NULL.
- The queue operations are defined as follows.
  - The **enqueue** operation first creates a node containing the data value, inserts the node after the rear (back) node, and updates both front and rear.
  - The **dequeue** operation deletes the front node (i.e., the node pointed by the front pointer) and updates the front and rear.
- The peek operation returns the data value in the front node.

Unit Testing in C

Introduction to Unit Testing in C

- Unit testing is the process of testing individual units or components of code to ensure they work as intended.
- In C, unit testing helps identify bugs early, ensures code reliability, and makes maintenance easier.
- Each test validates a small part of the code, typically a function, by checking if it produces the expected output.

Benefits of Unit Testing

- Catch errors at the component level before they propagate.
- This increases confidence that each part of the code functions correctly.
- It is also easier to make changes without breaking existing functionality.
- Well-written tests describe the code's functionality.

Strategies for Unit Testing in C

- **Isolate Functions** – Test each function independently to ensure each performs as expected.
- **Use Stubs** – Replace dependencies with simplified code to isolate the unit under test.
- **Edge Cases** – Include tests for boundary conditions, such as zero values, maximum values, or empty inputs.
- **Code Coverage** – Aim for high coverage, ensuring all paths within a function are tested.


Thank you