UNIT 3 · STACKS

10. Stack ADT and Implementation

CSC211 · Algorithms and Data Structures

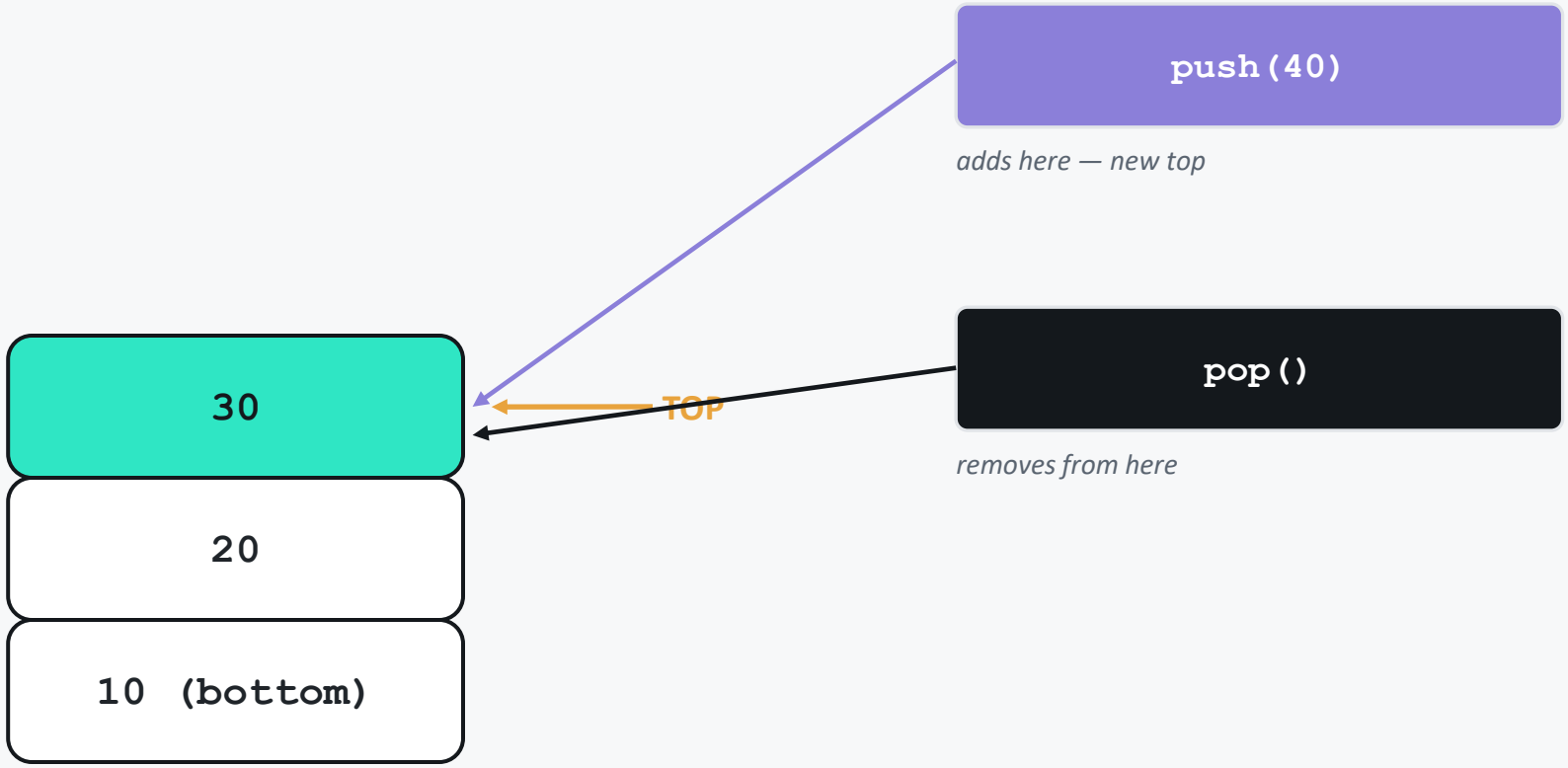
COMSATS University Islamabad, Lahore Campus

Agenda

1. The stack concept and the LIFO principle
2. The Stack ADT: push, pop, peek
3. Array-based and linked-list-based implementations, traced
4. Overflow, underflow, and choosing between the two

The Stack Concept: LIFO

Like a physical stack of plates: add to the top, remove from the top — never the middle or bottom without removing everything above it first.



L.I.F.O. — Last In, First Out: whatever was pushed most recently is the first thing popped.

The Stack ADT

As an Abstract Data Type, a stack promises exactly these operations, regardless of how it's implemented underneath.

`push(value)`	Add value to the top of the stack
`pop()`	Remove and return the value at the top
`peek()` / `top()`	Return the value at the top, without removing it
`isEmpty()`	Report whether the stack has any elements at all

Array-Based Implementation

array_stack.cpp

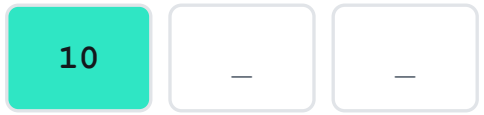
```
class ArrayStack {
    static const int CAPACITY = 100;
    int data[CAPACITY];
    int topIndex;   // -1 means empty
public:
    ArrayStack() : topIndex(-1) {}
    bool isFull() const { return topIndex == CAPACITY - 1; }

    void push(int value) {
        if (isFull()) throw overflow_error("Stack overflow");
        data[++topIndex] = value;
    }
    int pop() {
        if (isEmpty()) throw underflow_error("Stack underflow");
        return data[topIndex--];
    }
};
```

push writes then increments (++topIndex); pop reads then decrements (topIndex--) — both are O(1).

Tracing the Array's Underlying State

topIndex = -1

push(10) → topIndex = 0

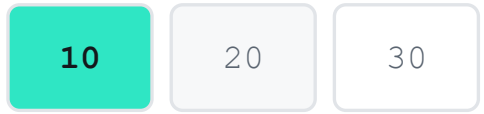


push(20) → topIndex = 1

push(30) → topIndex = 2

pop()→30 → topIndex = 1

pop()→20 → topIndex = 0



pop() never erases data — it just decrements topIndex. Faded cells are still physically in memory but invisible to the ADT; the next push silently overwrites them.

Linked-List Implementation

linked_stack.cpp

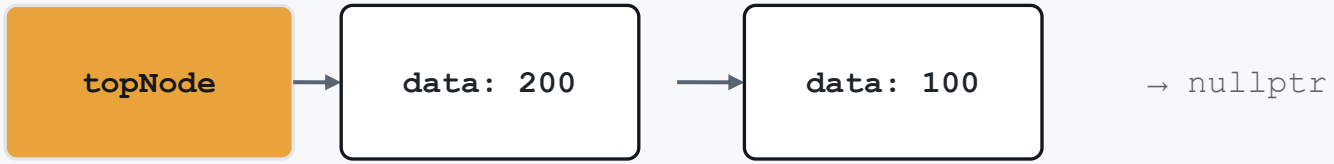
```
class LinkedStack {
    Node* topNode;
public:
    LinkedStack() : topNode(nullptr) {}

    void push(int value) {
        Node* newNode = new Node(value);
        newNode->next = topNode;
        topNode = newNode;
    }
    int pop() {
        if (isEmpty()) throw underflow_error("Stack underflow");
        Node* oldTop = topNode;
        int value = oldTop->data;
        topNode = topNode->next;
        delete oldTop;
        return value;
    }
};
```

push inserts at the head; pop removes from the head. Neither ever walks the list — no fixed-capacity limit, no resize.

Linked Stack: Nodes on the Heap

topNode = nullptr (empty)

after push(100), push(200)



after pop() → 200 (node freed)

Each pop() genuinely frees the old top node's memory — no leftover "dead" data, since each node is its own heap allocation.

Array-Based vs. Linked-List-Based

	Array-based stack	Linked-list-based stack
Maximum size	Fixed at construction (or resize-and-copy)	Grows one node at a time, limited by memory
Memory / element	Just the element itself	Element plus one next pointer
Cache behavior	Excellent — contiguous memory	Worse — nodes scattered on the heap
Best when...	Max size known or boundable	Size unpredictable, no pre-reservation

std::vector's growth strategy borrows the array stack's idea: it doubles capacity when full, keeping amortized push O(1) even though a resizing push is O(n).

Complexity of Stack Operations

O(1)*	O(1)	O(1)	O(1)
push()	pop()	peek()	isEmpty()

**push is O(1) unless the array is full and must resize-and-copy.*

Every core stack operation is O(1) in both implementations.

The difference is entirely about capacity: the array version has a hard limit (or needs a resize-and-copy step), while the linked-list version keeps growing one node at a time for as long as memory allows.

Overflow and Underflow: Real, Caught Exceptions

stack_edge_cases.cpp — CAPACITY = 5

```
try {
    stack.push(60);   // 6th push on a CAPACITY=5 stack
} catch (const overflow_error& e) {
    cout << "caught: " << e.what();  // "Stack overflow"
}

try {
    stack.pop();      // pop on an already-empty stack
} catch (const underflow_error& e) {
    cout << "caught: " << e.what();  // "Stack underflow"
}
```

Overflow
pushing onto an already-full
array-based stack only

Underflow
popping/peeking an empty stack
threatens BOTH implementations

LinkedStack has no CAPACITY and can't overflow the same way — but new itself throws std::bad_alloc if the system truly runs out of heap.

Key Takeaways

- A stack enforces LIFO (Last In, First Out) — the only element you can ever touch is the one on top.
- The Stack ADT has three core operations — push, pop, peek — each O(1) regardless of whether it's array- or linked-list-based.
- An array-based stack has a fixed capacity (or needs a resize); a linked-list-based stack grows indefinitely, one node at a time, at the cost of one pointer per element.
- Overflow (pushing when full) only threatens the array-based version; underflow (popping/peeking when empty) threatens both, and both implementations guard against it with a clear exception.
- Real code almost always reaches for std::stack — but understanding push/pop mechanics is what makes Lecture 11's stack-based algorithms make sense.