



Stacks and Queues

CEN207 Data Structures — Week 3

Asst. Prof. Dr. Uğur CORUH · Fall 2026–2027

Today's plan (3 hours)

Hour	Topic
1	Stack ADT · array stack · overflow Anim 1–2 · linked stack Anim 3
2	Stack apps: brackets, postfix, prefix, infix Anim 4–8 · recursion Anim 9–10
3	Hanoi Anim 11 · queue variants Anim 12–16 · self-check

Learning outcomes: LO.1 (explain fundamental data structures) · LO.7 (choose the right structure)

This week's concepts — where

Concept	Where
Stack ADT, LIFO, push/pop	Section 1
Expression algorithms (postfix, infix)	Section 2
Recursion, call stack, Tower of Hanoi	Sections 3–4
Queue ADT, FIFO, circular queue, deque	Section 5

How the code examples work

- Every idea has a complete **C** and **Java** program
- C: `gcc -std=c11 -Wall -Wextra -o /tmp/x file.c && /tmp/x`
- Java: `javac -d /tmp/j File.java && java -cp /tmp/j File`
- Sources: `code/week-03/c/` and `code/week-03/java/`
- Each program's expected output is in the week notes

Bridge from Weeks 1–2

- Weeks 1–2 gave you two tools: **pointers/memory** and the **linked list**
- This week reuses both, constantly, in new shapes
- New idea this week: restricting *where* you may touch a structure

Recap — Week 1: pointers and memory

- `int *p` holds an **address**, not a number
- `malloc` asks the OS for a box; `free` gives it back
- Forgetting `free` → **memory leak**
- Using memory after `free` → **dangling pointer**

Recap — Week 2: linked lists

- A linked list is a chain of **nodes**
- Each node: a value + a pointer to the next node
- Adding/removing rewires a couple of pointers — no shifting
- Today's linked stack/queue reuse this idea exactly

Map of the week — the one rule

- Both stacks and queues are **linear** structures
- A **stack** lets you touch only **one** end
- A **queue** makes you add at one end, remove at the other
- That single rule change decides what each is good for

Map of the week — at a glance

Stack topics	Queue topics
Array stack, linked stack	Array queue, circular queue
Brackets, postfix, infix	Linked-list queue
Recursion, call stack	Deque, multilevel queue
Tower of Hanoi	—

1. The Stack: Last In, First Out

A question to start

Visit three web pages, then click **Back** three times.

You land on page 2, then page 1, then nowhere.

The *last* page visited is the *first* one revisited.

A short history

- Late 1950s: "stack", "push", "pop" already in CS literature
- **1957** — Bauer & Samelson (Munich) patent a hardware stack
- **1946** — Turing used a similar idea in the ACE computer
- One of the oldest ideas in computing — still in every program

Intuition — a stack of plates

- Take a plate only from the **top**
- Add a plate only to the **top**
- Cannot reach the middle without removing everything above it
- Last plate placed is the first one removed — **LIFO**

Three operations

- **push** — put a new element on top
- **pop** — remove and return the top element
- **peek** (or **top**) — look at the top without removing it
- All three touch only **one end**

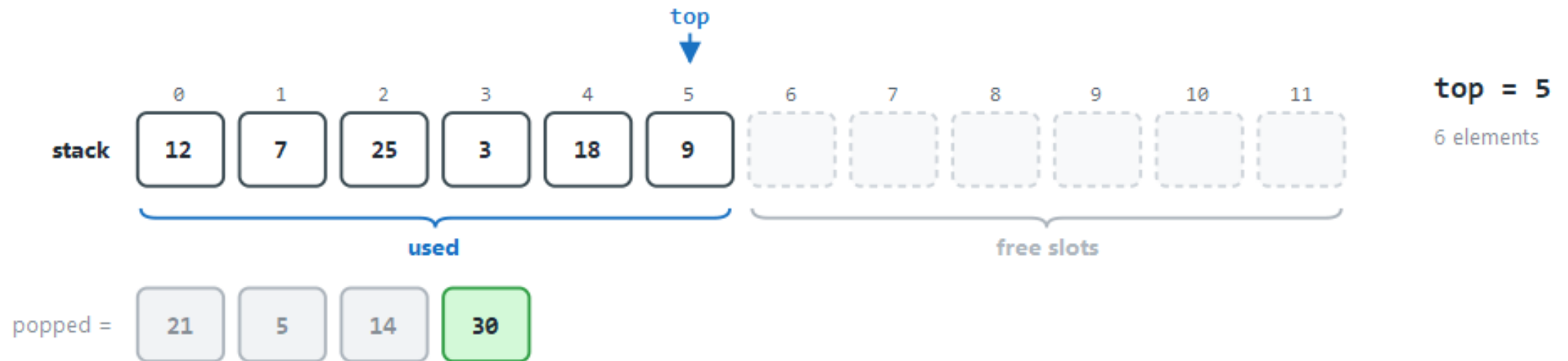
The Stack ADT

Operation	What it does	Precondition	Complexity
<code>push(x)</code>	Add <code>x</code> on top	not full (array)	$O(1)$
<code>pop()</code>	Remove/return top	not empty	$O(1)$
<code>peek()</code>	Return top, keep it	not empty	$O(1)$
<code>isEmpty()</code>	zero elements?	none	$O(1)$

Array stack — the idea

- Reserve a fixed-size array `data[CAP]`
- Keep one integer, `top` : index of the topmost slot
- Empty stack: `top == -1` — "no top yet"
- `push` : `top++` , write; `pop` : read, `top--`

Array stack: push and pop



Done: 10 pushes, 4 pops. Stack (bottom to top): 12, 7, 25, 3, 18, 9. Every operation is $O(1)$.

Code — push()

```
bool push(int x) {  
    if (top == CAP - 1) return false;  
    top = top + 1;  
    data[top] = x;  
    return true;  
}
```

Code — pop()

```
bool pop(int *out) {  
    if (top == -1) return false;  
    *out = data[top];  
    top = top - 1;  
    return true;  
}
```

Expected output

```
push(12) -> true    [top = 0]
push(21) -> true    [top = 9]
pop()      -> 21     [top = 8]
pop()      -> 5      [top = 7]
pop()      -> 14     [top = 6]
pop()      -> 30     [top = 5]
```

Why every operation is $O(1)$

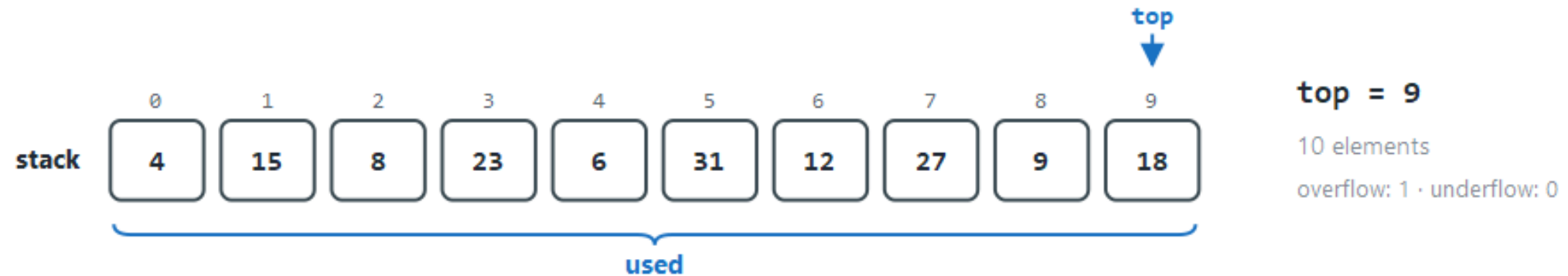
- `push` / `pop` do a **fixed number of steps**
- One condition check, one move of `top`, one array access
- True whether the stack holds 1 element or a million
- No loop ever touches the other elements

Overflow and underflow — the question

An array has a **fixed size**. What happens when you **push** a full stack, or **pop** an empty one?

A correct stack must **check first and refuse**.

Stack overflow and underflow



popped =

Done: 10 successful pushes, 0 successful pops, 1 overflow, 0 underflows. Stack (bottom to top): 4, 15, 8, 23, 6, 31, 12, 27, 9, 18. Two simple if checks catch overflow and underflow before they happen.

Common mistakes (array stack)

- Checking `top == CAP`, not `top == CAP - 1`
- Popping without checking `isEmpty` first
- Ignoring the `bool` return value of `push`

Quick question

Why is `top = -1` a better choice for "empty" than, say, `top = 0`?

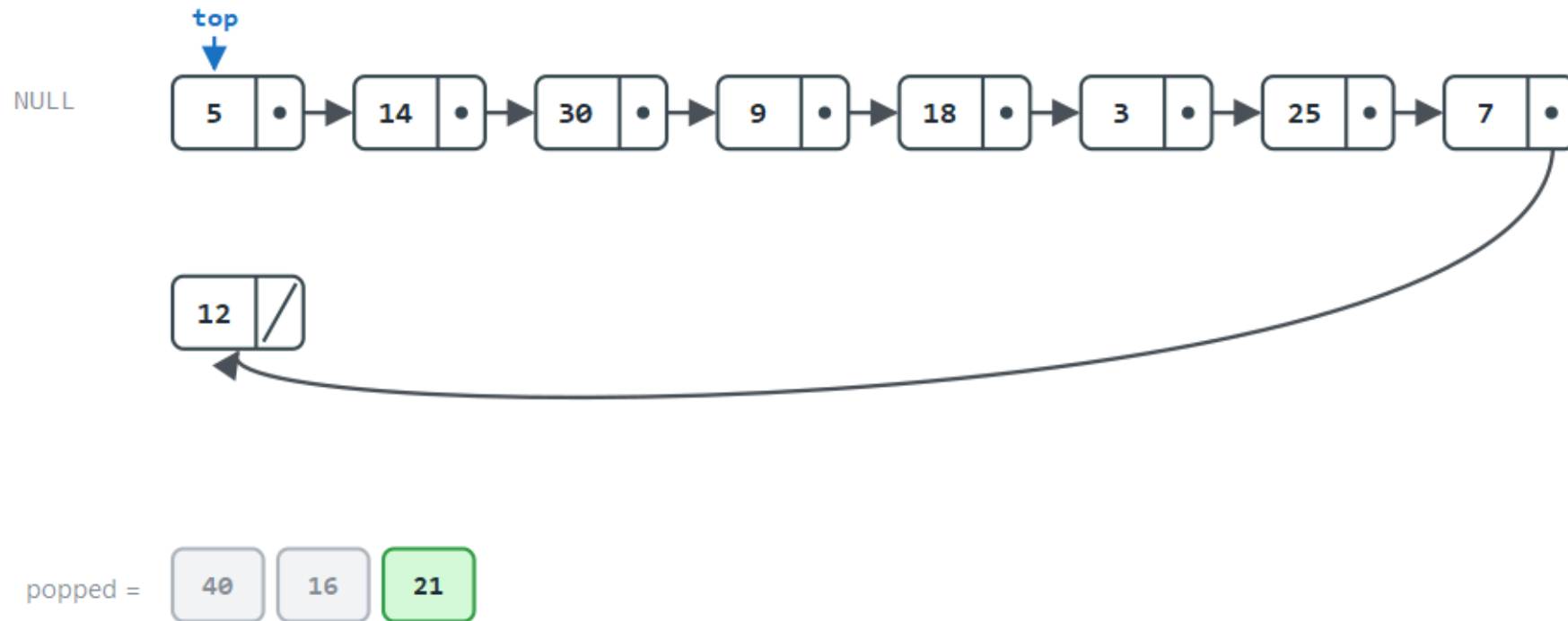
Answer

Index `0` is a **valid slot**. If `top = 0` meant "empty", you could not tell an empty stack from one holding a single element at index 0.

A stack that never overflows

- Array stack has a hard ceiling: CAP
- **Linked-list stack**: every element is its own node
- `top` is now a **pointer**, not an index
- `push` allocates; `pop` frees

Linked-list stack: push and pop



Done: 12 pushes, 3 successful pops, 0 underflows. Stack (bottom to top): 12, 7, 25, 3, 18, 9, 30, 14, 5. A linked stack never overflows until memory runs out; the cost is the extra next pointer in every node.

Code — struct Node + push()

```
typedef struct Node {  
    int data;  
    struct Node *next;  
} Node;  
Node *top = NULL;  
  
void push(int x) {  
    Node *n = malloc(sizeof(Node));  
    n->data = x;  
    n->next = top;  
    top = n;  
}
```

Code — pop()

```
bool pop(int *out) {  
    if (top == NULL) return false;  
    Node *tmp = top;  
    *out = tmp->data;  
    top = top->next;  
    free(tmp);  
    return true;  
}
```

Why push is still $O(1)$

- `malloc` for **one fixed-size node** is constant time
- Does not depend on how many nodes already exist
- No loop over existing elements
- Cost: one pointer (`next`) per node vs. array's zero

Common mistakes (linked stack)

- Forgetting `free(tmp)` in `pop` — slow memory leak
- Using a pointer after `free` ing it — undefined behavior
- Java: holding an old reference blocks garbage collection

Array vs. linked-list stack

	Array stack	Linked stack
Capacity	Fixed, can overflow	Limited by memory
Extra memory	None	One pointer/node
Cache behavior	Contiguous, fast	Scattered, slower

Quick question

`CAP = 8` . After 5 pushes and 2 pops, what is `top` ?

Answer

$-1 + 5 - 2 = 2$. Three elements remain
(indices 0, 1, 2), and $top == 2$.

2. Stack Applications: Expressions

Infix is awkward for computers

- $A + B * C$ — is $*$ computed before $+$? Needs **precedence**
- Two other notations move the operator so no precedence rule is ever needed at evaluation time

A short history

- 1920s — Polish logician **Jan Łukasiewicz** introduces prefix ("Polish") and postfix ("Reverse Polish") notation
- Famous via **HP calculators**: no parenthesis keys needed
- Evaluated by a small stack machine — what you build next

Three notations

Notation	Operator position	A + B * C
Infix	Between operands	A + B * C
Postfix (RPN)	After both operands	A B C * +
Prefix (Polish)	Before both operands	+ A * B C

Balanced brackets — the question

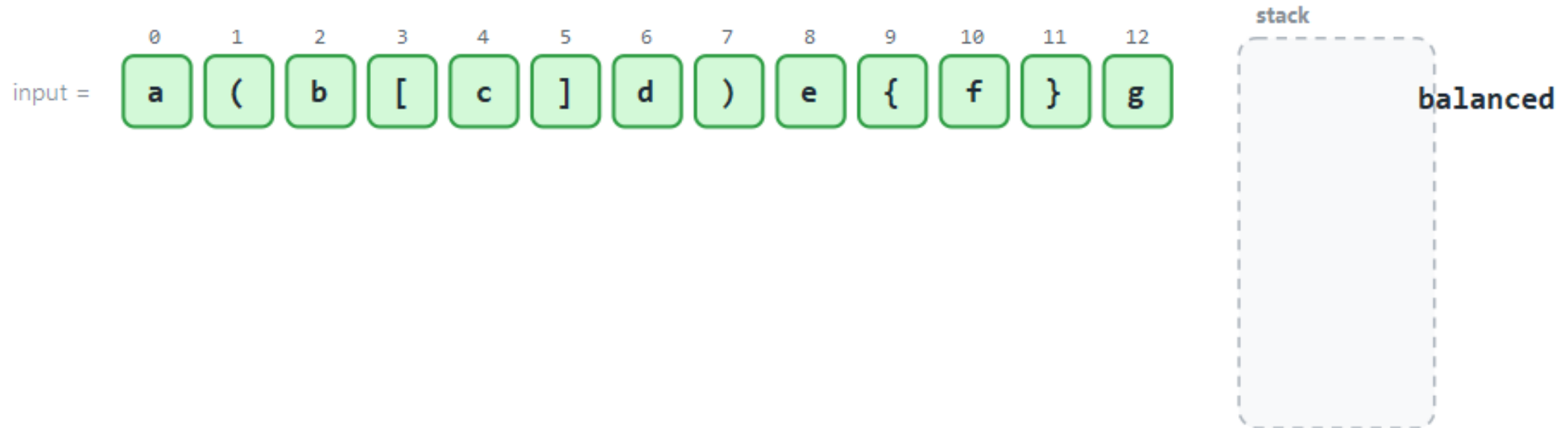
Is `{([ ])([ ])}` validly nested?

Every closer must match the **most recently opened** bracket that is still open.

The rule

- Scan the string once
- Every **opener**: push it
- Every **closer**: pop, and it must match
- If empty stack when a closer arrives → unbalanced
- If the stack is not empty at the end → unbalanced

Checking brackets with a stack



The input is over and the stack is empty: every opener found its closer → balanced. Each character is handled once: $O(n)$.

Code — matches()

```
static bool matches(char open, char close) {  
    return (open == '(' && close == ')') ||  
           (open == '[' && close == ']') ||  
           (open == '{' && close == '}');  
}
```

Code — `balanced()`

```
bool balanced(const char *s) {
    char st[100]; int top = -1;
    for (int i = 0; s[i] != '\0'; i++) {
        char c = s[i];
        if (c == '(' || c == '[' || c == '{') {
            st[++top] = c;
        } else if (c == ')' || c == ']' || c == '}') {
            if (top == -1) return false;
            char o = st[top--];
            if (!matches(o, c)) return false;
        }
    }
    return top == -1;
}
```

Complexity

Each character is looked at **exactly once**,
each stack op is $O(1)$ → **balanced** runs in **$O(n)$** .

Worst-case space: $O(n)$ — a string of all openers.

Pitfall — three ways to be unbalanced

1. Closer arrives, top does not match — `(]`
2. Closer arrives, stack already empty — `)`
3. String ends, stack **not** empty — `((`

Quick question

Is `((A+B)` accepted by `balanced` ? Why or why not?

Answer

No. Every `(` is pushed, and the string ends with the stack still holding one — `top == -1` is false, so `balanced` correctly returns false.

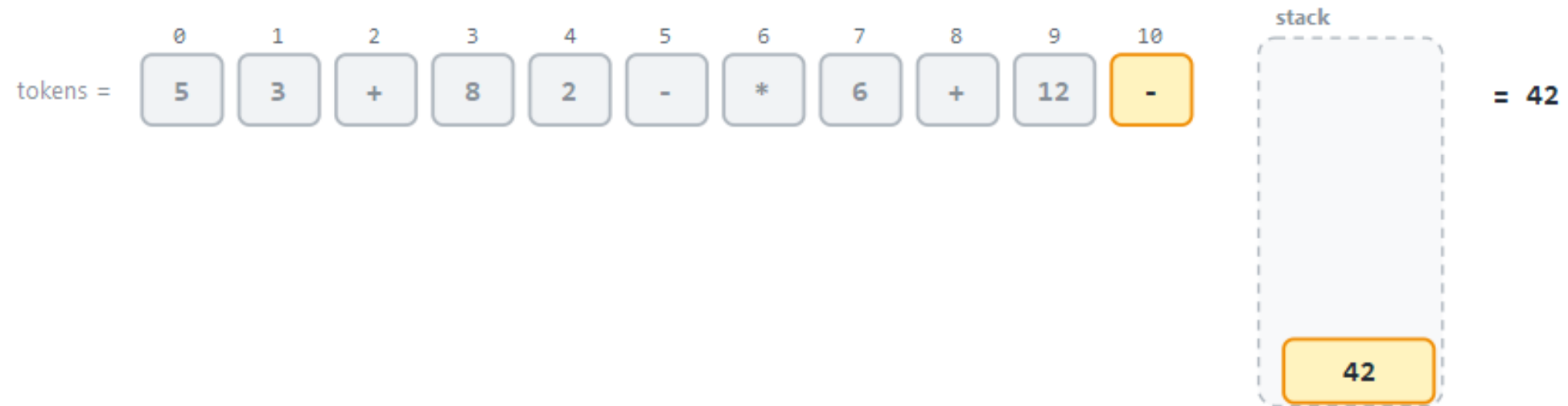
Postfix evaluation — the question

5 3 + 8 2 - * 6 + 12 - should evaluate to

$(5+3)*(8-2) = 48$, $48+6 = 54$, $54-12 = 42$.

No precedence rules needed at all — just a scan.

Evaluating a postfix expression



The input is over; one value is left on the stack: 42. Each token is handled once: $O(n)$.

Code — apply()

```
static int apply(char op, int a, int b) {  
    switch (op) {  
        case '+': return a + b;  
        case '-': return a - b;  
        case '*': return a * b;  
        case '/': return a / b;  
        default: return 0;  
    }  
}
```

Code — eval_postfix()

```
int eval_postfix(char *tok[], int n) {
    int st[100]; int top = -1;
    for (int i = 0; i < n; i++) {
        char *t = tok[i];
        if (isdigit(t[0])) {
            st[++top] = atoi(t);
        } else {
            int b = st[top--];
            int a = st[top--];
            st[++top] = apply(t[0], a, b);
        }
    }
    return st[top];
}
```

Complexity

$O(n)$ in the number of tokens: each token is pushed at most once and popped at most once.

Pitfall — operand order matters

`8 2 -` means `8 - 2`. First popped (`b`) is the **right** operand; second popped (`a`) is the **left**. Swap them and you compute `2 - 8` instead.

Quick question

Why does `eval_postfix` need only **one** stack, while `to_postfix` (coming next) needs a second output area too?

Answer

Evaluation **collapses** two operands and an operator into one number immediately — nothing extra to remember. Conversion only **reorders** symbols, so it needs a place to collect them.

Evaluating a prefix expression

- Operator comes **before** its operands: `+ A B`
- Same idea as postfix, scanned **right to left**
- First value popped is now the **left** operand

Evaluating a prefix expression



The input is over; one value is left on the stack: 56. Prefix and postfix are the same idea in two directions; both run in $O(n)$.

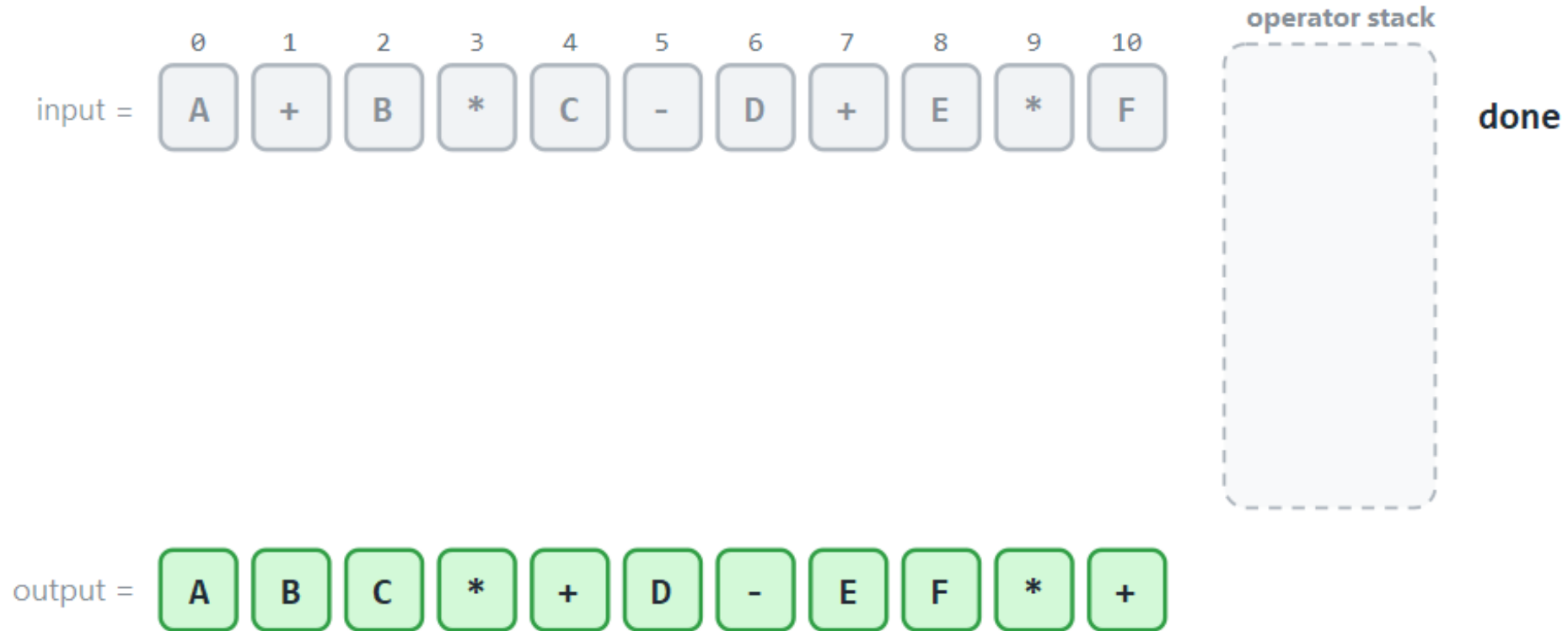
Prefix vs. postfix — the mirror

- Scan **right to left**, not left to right
- Pop the **left** operand first, then the right one
- Same **$O(n)$** complexity, same error checks (division by zero, too few/many operands)

Infix to postfix — the idea

- Humans write infix; our evaluator needs postfix
- **Shunting-yard algorithm** (Dijkstra) — after railway yards that reorder freight cars
- A second stack, holding **operators**, resolves precedence once, up front

Converting infix to postfix



The input is over: the operators left on the stack are flushed to the output in order. Result: A B C * + D - E F * +. Each character is pushed and popped once: $O(n)$.

Code — prec()

```
static int prec(char op) {  
    if (op == '+' || op == '-') return 1;  
    if (op == '*' || op == '/') return 2;  
    return 0;  
}
```

Code — to_postfix()

```
void to_postfix(const char *in, char *out) {
    char ops[100]; int top = -1, k = 0;
    for (int i = 0; in[i]; i++) {
        char c = in[i];
        if (isdigit(c)) { out[k++] = c; }
        else {
            while (top >= 0 && prec(ops[top]) >= prec(c))
                out[k++] = ops[top--];
            ops[++top] = c;
        }
    }
    while (top >= 0) out[k++] = ops[top--];
    out[k] = '\0';
}
```

Complexity

$O(n)$: every character is pushed onto the operator stack at most once, popped at most once.

Pitfall — three bugs here

- Using `>` instead of `>=` breaks left-associativity
- Forgetting the final flush loses trailing operators
- No parentheses support (left as a natural extension)

Quick question

Convert $A*B+C$ to postfix by hand.

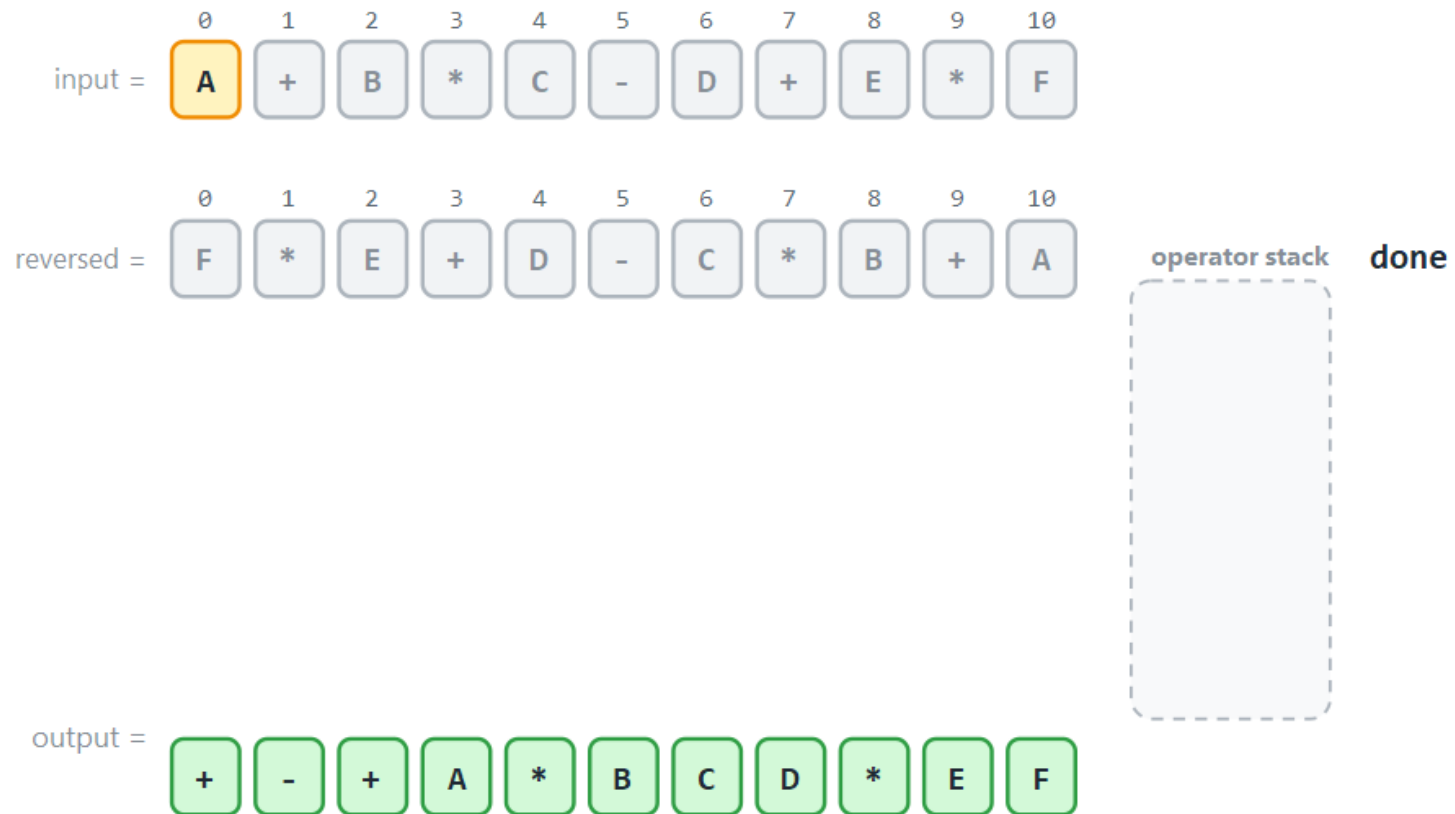
Answer

AB*C+ . A →out. * →push. B →out.
+ sees * (prec 2 ≥ 1): pop * →out, push + .
C →out. Flush + . Result: A B * C + .

Converting infix to prefix

- **Reverse** the input, swapping (↔)
- Run shunting-yard again, but pop only when a waiting operator is **strictly** stronger (> , not >=)
- **Reverse** the result

Converting infix to prefix



3) Reverse the intermediate result: + - + A * B C D * E F — that is the prefix form. Each character is pushed and popped once: $O(n)$.

Pitfall — strict `>`, not `>=`

- Prefix conversion pops only a **strictly stronger** waiting operator — using `>=` breaks the final reversal
- Forgetting to swap `( / )` while reversing breaks grouping

3. Recursion and the Call Stack

What is recursion — a question

$\text{fact}(n) = n * \text{fact}(n - 1)$, base case $\text{fact}(0) = 1$.

$\text{fact}(3) = 3 * (2 * (1 * \text{fact}(0))) = 6$

The base case is mandatory

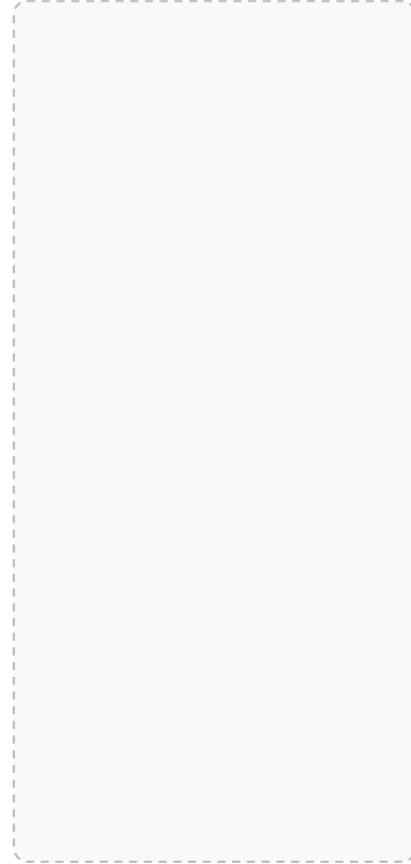
- Not optional — it is what **stops** the recursion
- Without one: the function calls itself **forever**
- "Forever" really means: until memory runs out

The simplest recursion: countdown

- Print `n`, recurse on `n - 1`, until the base case
- Base case must be `n <= 0`, **not** `n == 0`
- `n = 0` or negative `n` must stop on the very first call

Recursion: countdown

call stack



screen:

10
9
8
7
6
5
4
3
2
1
Liftoff!

Every call pushed a frame and every return popped one: recursion really is a stack. Depth 11 $\rightarrow O(11)$ memory. Printed: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, Liftoff!.

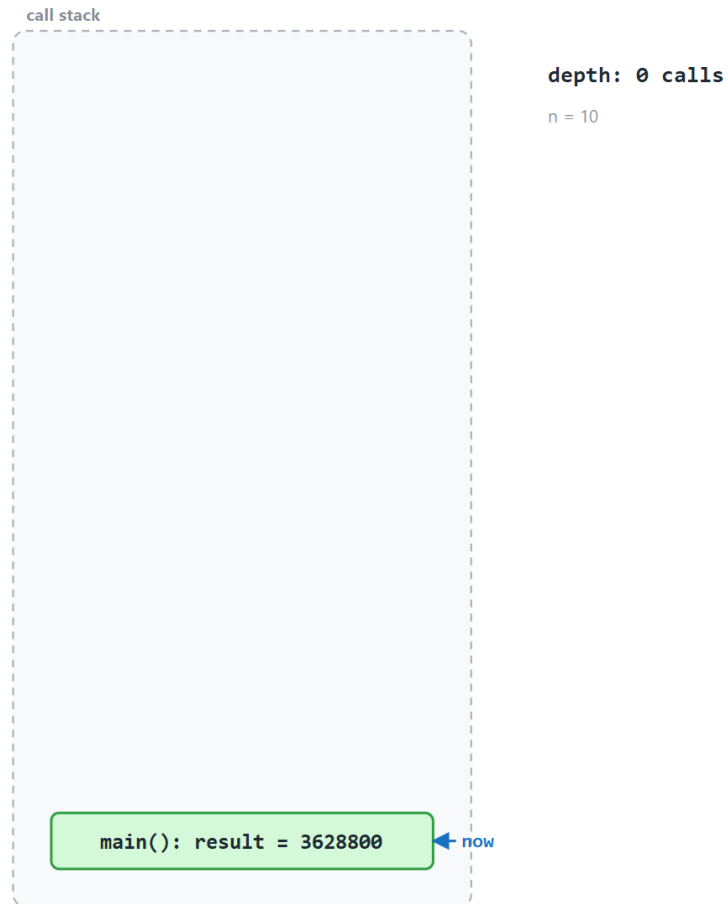
Pitfall — the wrong base case

`n == 0` only: a negative starting value **never** satisfies it — infinite recursion. `n <= 0` fixes it: negative input stops on the very first call.

The call stack: recursion is a stack

- Every function call pushes a **frame**:
local variables + the return address
- Returning from a function **pops** that frame
- This is a genuine **LIFO stack**, built into every running program, whether you write one or not

Recursion and the call stack: fact(n)



Done: fact(10) = 3628800. Every call does $O(1)$ extra work; a recursion of depth n uses $O(n)$ call-stack memory.

Code — fact()

```
int fact(int n) {  
    if (n == 0) return 1;  
    return n * fact(n - 1);  
}
```

Expected output

```
fact(10) = 3628800
```

Complexity

`fact(n)` makes `n` recursive calls: **$O(n)$ time**,
and — easy to forget — **$O(n)$ stack space**,
one frame per pending call.

A loop computing the same product uses $O(1)$ space.

Pitfall — silent `int` overflow

`13! = 6227020800` — too big for a 32-bit `int`
(max `2147483647`). The program does **not** crash;
it silently wraps to `1932053504`.

Pitfall — missing base case

Without a reachable base case, a recursive function pushes a **new frame every call**, forever — until the call stack itself overflows.

Pitfall — a problem that doesn't shrink

`fact(n)` calls itself with `n - 1`, strictly smaller. Every recursive call **must** move measurably closer to the base case.

Quick question

`fact(3)` is the original call. How many frames sit on the call stack the instant `fact(0)` starts?

Answer

Four: `fact(3)` , `fact(2)` , `fact(1)` , `fact(0)` ,
each still waiting for the one below it to return.

Quick question

If the base case were `if (n == 1) return 1;`
and someone called `fact(-1)`, what happens?

Answer

-1 never equals 1 ; the function recurses on
-2, -3, -4, ... forever, eventually
overflowing the call stack.

4. The Tower of Hanoi

The puzzle

- Three rods; disks stacked large-to-small on rod A
- Move the whole stack to rod C, one disk at a time
- Never place a larger disk on a smaller one
- Invented by Édouard Lucas, 1883 (monks legend)

The recursive idea

To move `n` disks from `from` to `to`, via `via`:

1. Move top `n - 1` disks to `via`
2. Move the largest disk to `to`
3. Move the `n - 1` disks from `via` onto it

Tower of Hanoi

move 15 / 15



moves =

1A→B	2A→C	1B→C	3A→B	1C→A	2C→B	1A→B	4A→C	1B→C	2B→A	1C→A	3B→C	1A→B	2A→C
1B→C													

Done: 4 disks moved in 15 moves ($2^4 - 1 = 15$) — the move count grows exponentially.

Code — hanoi()

```
void hanoi(int n, char from, char to, char via) {  
    if (n == 0) return;  
    hanoi(n - 1, from, via, to);  
    move_disk(n, from, to);  
    hanoi(n - 1, via, to, from);  
}
```

Expected output

```
move 1: disk 1 from A to B
move 2: disk 2 from A to C
move 3: disk 1 from B to C
move 4: disk 3 from A to B
...
total moves = 15
```

Complexity

$$T(n) = 2 * T(n-1) + 1, \quad T(0) = 0$$

→ $T(n) = 2^n - 1$ — **exponential** growth.

$n = 4$: $2^4 - 1 = 15$, matching the program above.

The legend's 64 disks

$2^{64} - 1 \approx 1.8 \times 10^{19}$ moves.

At one move per second: **~585 billion years** —
tens of times the current age of the universe.

Looking further ahead

"Solve a smaller version, then combine" is exactly how **depth-first search (DFS)** explores a tree or graph, coming in the next two weeks.

Quick question

How many moves does a 5-disk Tower of Hanoi need?

Answer

$2^5 - 1 = 31$ moves.

Quick question

In `hanoi(n - 1, from, via, to)`, why are the last two arguments swapped vs. the outer call?

Answer

For moving the top $n - 1$ disks out of the way, the *destination* is the spare rod, and the *spare* is the original destination — roles rotate at every level of recursion.

5. The Queue: First In, First Out

Intuition — a waiting line

- A checkout line, a print queue, a web request queue
- New arrivals join at the **back**
- The longest-waiting item leaves from the **front**

FIFO — the mirror of LIFO

First In, First Out. Same two underlying operations as a stack, different names, and now they touch **different ends**.

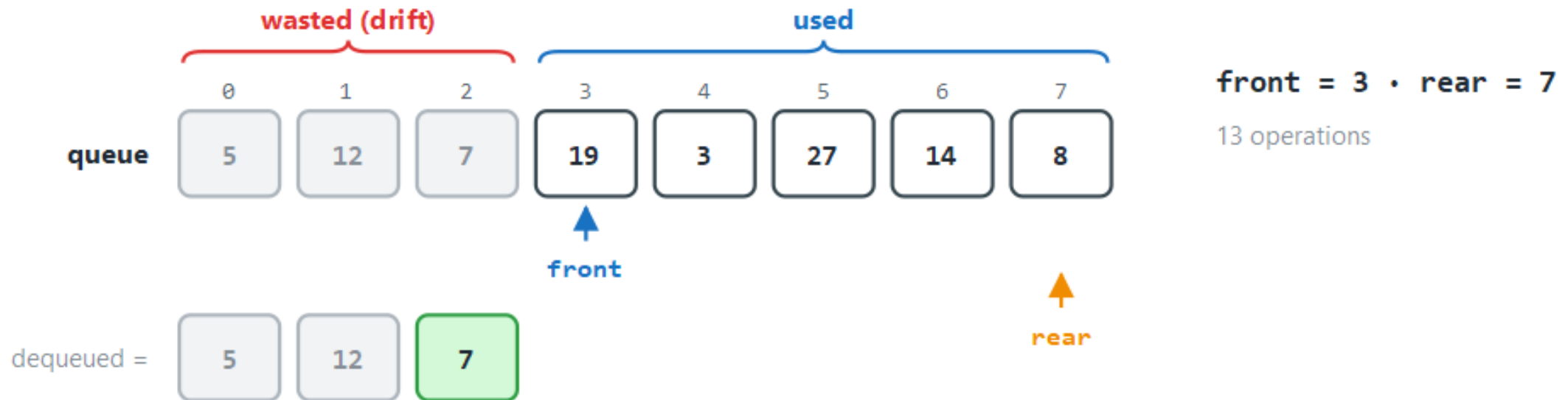
The Queue ADT

Operation	What it does	Precondition	Complexity
<code>enqueue(x)</code>	Add at the back	not full (array)	O(1)
<code>dequeue()</code>	Remove/return front	not empty	O(1)
<code>peek()</code>	Return front, keep it	not empty	O(1)
<code>isEmpty()</code>	zero elements?	none	O(1)

Array queue — the idea

- Keep two indices: `front` (oldest) and `rear` (newest)
- `enqueue`: advance `rear`, write there
- `dequeue`: read at `front`, advance `front`
- No elements shifted — but watch the space at index 0

Queue in a plain array



Done: 8 enqueues, 3 dequeues, 2 overflows. In the queue: 19, 3, 27, 14, 8. front = 3 — 3 cells can never be used again (38% of the capacity wasted).

Code — enqueue()/dequeue() (naive)

```
bool enqueue(int x) {  
    if (rear == CAP - 1) return false;  
    q[++rear] = x;  
    return true;  
}  
  
bool dequeue(int *out) {  
    if (front > rear) return false;  
    *out = q[front++];  
    return true;  
}
```

Expected output

```
enqueue(5,12,7,19,3,27,14,8): front=0, rear=7  
dequeue() x3:    front=3, rear=7  
enqueue(99) -> false  
enqueue(42) -> false
```

Pitfall — the drift problem

`rear` only ever moves **forward**. The queue reports "full" while cells sit empty at the front — it has **drifted** off the array.

Quick question

Why does this queue say "full" even with three empty cells at the beginning?

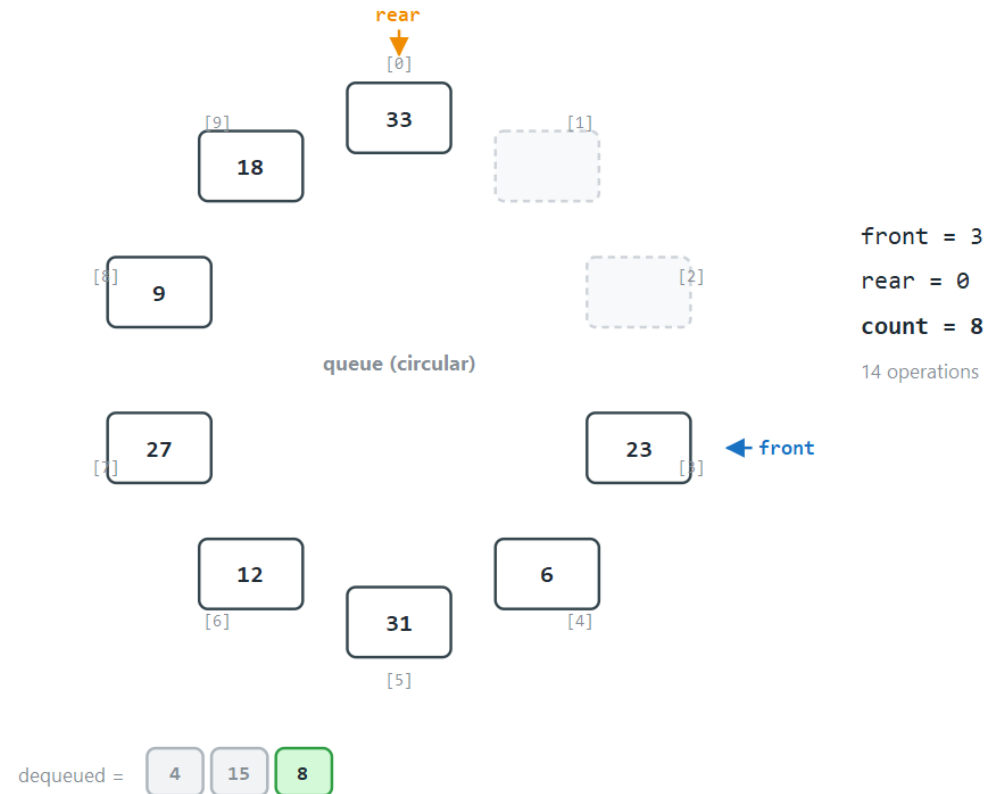
Answer

`enqueue` only checks `rear == CAP - 1`; it never looks at whether cells before `front` are free. The real fix is next: think of the array as a **ring**.

The circular queue — think ring

- After the last index comes the first one again
- $(\text{index} + 1) \% \text{CAP}$ walks forward, wraps around
- $\text{front} == \text{rear}$ is now ambiguous (empty? full?)
- Keep one more field, `count`, to resolve it

Circular queue



Done: 11 enqueues, 3 dequeues. In the queue: 23, 6, 31, 12, 27, 9, 18, 33 (count = 8). front == rear here only ever means "exactly one element"; the real trap is $\text{front} == (\text{rear} + 1) \% \text{CAP}$ — that looks identical whether the queue is empty or full, only count tells them apart.

Code — enqueue() (circular)

```
bool enqueue(int x) {  
    if (count == CAP) return false;  
    rear = (rear + 1) % CAP;  
    q[rear] = x;  
    count++;  
    return true;  
}
```

Code — dequeue() (circular)

```
bool dequeue(int *out) {  
    if (count == 0) return false;  
    *out = q[front];  
    front = (front + 1) % CAP;  
    count--;  
    return true;  
}
```

Why we also need **count**

`front == rear` could mean "one element",
"empty", **or** "completely full" — a plain
index comparison can no longer tell these apart.

Complexity — still $O(1)$, no drift

All five cells are used before the queue reports "full". The drift problem is completely gone, and every operation is still $O(1)$.

Common mistakes (circular queue)

- Using `front == rear` alone to mean "empty"
- Forgetting the modulo on **one** of the two indices
- Either mistake corrupts the ring after one wrap

Quick question

After `count` reaches `CAP`, what does
`front == rear` mean now, vs. when empty?

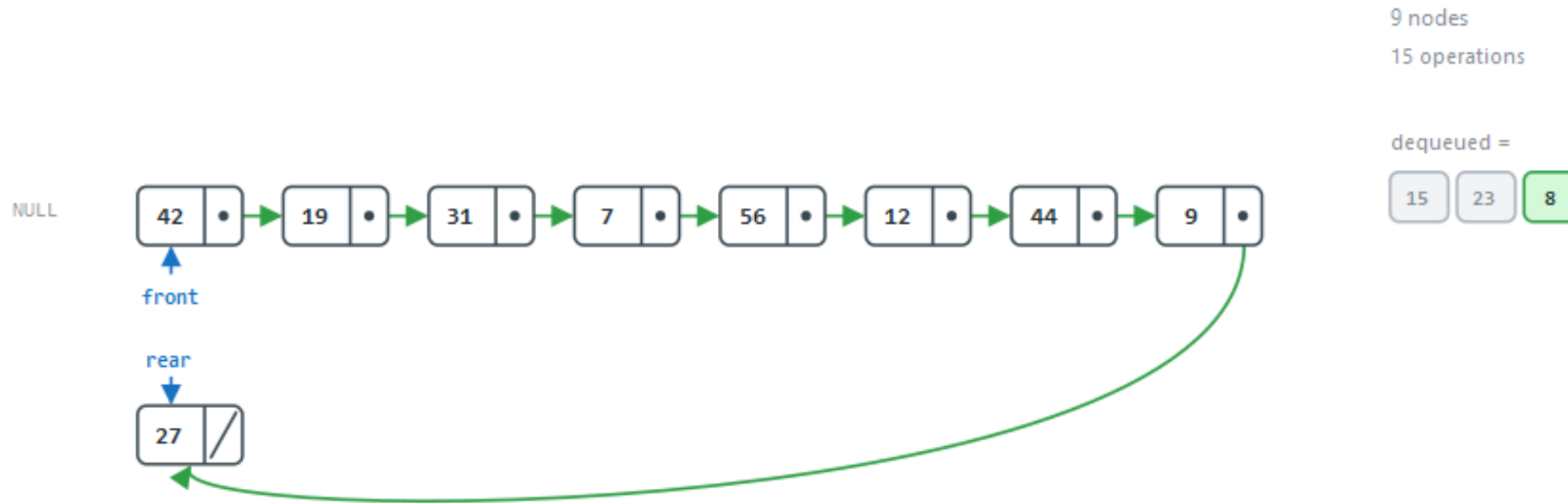
Answer

Both states can show `front == rear ; count` is exactly what tells them apart, since the indices alone are ambiguous.

A queue that never overflows

- Linked-list queue: one node per element, no CAP
- Needs **two** pointers: `front` (remove), `rear` (add) — so both stay $O(1)$
- Without `rear`, `enqueue` would walk the whole list

Linked-list queue



Done: 12 enqueue calls, 3 dequeue calls. Queue (front → rear): 42, 19, 31, 7, 56, 12, 44, 9, 27. Thanks to the two pointers every operation is $O(1)$; there is no capacity limit.

Code — enqueue() (linked)

```
void enqueue(int x) {  
    QNode *n = malloc(sizeof(QNode));  
    n->data = x; n->next = NULL;  
    if (rear == NULL) {  
        front = rear = n;  
    } else {  
        rear->next = n;  
        rear = n;  
    }  
}
```

Code — dequeue() (linked)

```
bool dequeue(int *out) {  
    if (front == NULL) return false;  
    QNode *tmp = front;  
    *out = tmp->data;  
    front = front->next;  
    if (front == NULL) rear = NULL;  
    free(tmp);  
    return true;  
}
```

Common mistakes (linked queue)

- Forgetting to reset `rear` to `NULL` when empty
- Updating only one of `front` / `rear` with one node
- Both lead to a **dangling pointer** on the next enqueue

Quick question

Why does the linked queue need a `rear` pointer, while the linked stack does not?

Answer

The stack adds/removes at the **same** end (`top`) — one pointer suffices. The queue adds at one end, removes at the other — it needs a pointer to **each** end to stay $O(1)$.

The deque — both ends

A **deque** (double-ended queue) drops the "one end only" rule: it allows push/pop at **both** the front and the back.

The Deque ADT

Operation	What it does	Complexity
<code>push_back(x)</code>	Add at the back	$O(1)$
<code>push_front(x)</code>	Add at the front	$O(1)$
<code>pop_back()</code>	Remove the back	$O(1)$
<code>pop_front()</code>	Remove the front	$O(1)$

Double-ended queue (deque)



Done: 4 push_back, 3 push_front, 2 pop_back, 1 pop_front calls. Deque (front → back): 8, 5, 10, 40. All four operations are $O(1)$: a deque does everything a stack and a queue can do.

Code — push_back()

```
void push_back(Deque *d, int x) {
    DNode *n = malloc(sizeof(DNode));
    n->data = x;
    n->next = NULL;
    n->prev = d->back;
    if (d->back != NULL)
        d->back->next = n;
    else
        d->front = n;
    d->back = n;
}
```

Code — pop_back()

```
bool pop_back(Deque *d) {  
    if (d->back == NULL) return false;  
    DNode *tmp = d->back;  
    d->back = tmp->prev;  
    if (d->back != NULL)  
        d->back->next = NULL;  
    else  
        d->front = NULL;  
    free(tmp);  
    return true;  
}
```

Note — a deque can be a stack or a queue

- Only `push_back` / `pop_back` → behaves like a **stack**
- Only `push_back` / `pop_front` → behaves like a **queue**
- Java's `ArrayDeque` is recommended as a faster general replacement for both

Quick question

Empty deque: `push_front(1)` , `push_back(2)` ,
`push_front(3)` . Front to back — what is it?

Answer

3 1 2 . push_front(1) → [1] .

push_back(2) → [1, 2] .

push_front(3) → [3, 1, 2] .

The multilevel queue

- Real schedulers rarely treat all work equally
- OS scheduler: keeps **several** FIFO queues at once
- Interactive first, then background, then batch
- Each level is an **ordinary queue**; a policy picks one

Multilevel queue — levels

Level	Kind of work	Priority
Queue 1	Interactive	Highest
Queue 2	Background	Medium
Queue 3	Batch	Lowest

Multilevel queue scheduling



Done: 12 processes, service order — P3, P5, P7, P10, P1, P4, P8, P11, P2, P6, P9, P12.

Pitfall — starvation risk

One early **batch** process among nine system/interactive ones is served **last** — plain priority order alone cannot prevent starvation.

Quick question

Why is a multilevel queue not, strictly speaking, a new data structure?

Answer

Each level is an **ordinary queue**; "multilevel" describes a *scheduling policy* for choosing among several existing queues — not a new way to store data.

Summary — stack-based structures

Structure	Rule	Add/remove	Typical use
Array stack	LIFO	$O(1)$, fixed cap.	Small, bounded stacks
Linked stack	LIFO	$O(1)$, unbounded	Unpredictable size
Call stack	LIFO	Automatic	Every running program

Summary — queue-based structures (I)

Structure	Rule	Add/remove
Array queue (naive)	FIFO	$O(1)$, but drifts
Circular queue	FIFO	$O(1)$, no drift
Linked-list queue	FIFO	$O(1)$, unbounded

Summary — queue-based structures (II)

Structure	Rule	Typical use
Deque	Both ends	Sliding window, undo/redo
Multilevel queue	Several FIFOs + policy	OS scheduling

Summary — stack applications

Application	Structure	Key idea
Balanced brackets	Stack	Most recent, first closed
Postfix evaluation	Stack	Pop two, apply, push result
Infix → postfix	Operator stack + output	Resolve precedence once
Tower of Hanoi	Call stack	Solve n-1, act, solve n-1

The big picture

One rule — **which end(s) you may touch** — separates every structure in this lecture:
one end (stack), two ends (queue/deque),
or several parallel queues (multilevel).

Practice

- Exercises for all of today's topics are in the week notes: `docs/week-3/cen207-week-3.en.md`
- Peek, unbalanced-parens detection, prefix-free infix-to-prefix, count-free circular queue, palindrome check, fairer multilevel scheduling

Self-check round

Ten short questions. Think before the answer appears on the next slide.

1. What does LIFO stand for?

Last In, First Out — the stack.

2. What does FIFO stand for?

First In, First Out — the queue.

3. Why does array **push** check **$top == CAP - 1$** , not **$top == CAP$** ?

Valid indices run $0..CAP-1$; the stack is full the moment **top** reaches $CAP-1$.

4. In postfix evaluation, which operand is popped first?

The right-hand operand — it was pushed most recently.

5. Why does a plain array queue eventually report "full" even with empty cells at the front?

rear only increases and never reuses cells **dequeue** frees near the front.

6. What extra state does a circular queue need beyond `front` / `rear` ?

A **count** of current elements — indices alone cannot tell empty from full.

7. What is the base case of $\text{fact}(n) = n * \text{fact}(n - 1)$?

`fact(0) = 1` . Without it, calls never stop and the call stack overflows.

8. Why does the call stack qualify as a genuine stack?

It obeys LIFO exactly: the most recent call is always the next one to finish.

9. For n disks, how many Hanoi moves are needed?

$2^n - 1$ moves — exponential growth in n .

10. Name one thing a deque can do that a plain stack or queue cannot.

Add or remove at **both** the front and the back, each in $O(1)$.

Next week

Week 4 — Trees: Binary Trees, Traversals, Heaps

Every stack and queue today stored elements in a strict, unbranching sequence. Trees let a node have several children — traversed with the very same recursive pattern used for the Tower of Hanoi.

References (1/2)

- Course syllabus, Week 3: `docs/syllabus/syllabus.en.md`
- Cormen, Leiserson, Rivest, Stein. *Introduction to Algorithms*, 3rd ed. MIT Press
- Sedgewick, Wayne. *Algorithms*, 4th ed. Addison-Wesley, 2011

References (2/2)

- Deitel & Deitel. *C How to Program*, 7th ed.
- Y. D. Liang. *Introduction to Java Programming*, 10th ed.
- É. Lucas, *Récréations mathématiques*, 1883
- J. Łukasiewicz, Polish/Reverse Polish notation, 1920s