Chapter 7: Stacks

Data Structures

LIFO and basic operations | Array-based stack | Linked stack | Bracket matching | Next greater element | Towers of Hanoi | Stock span | Infix → postfix / prefix | Postfix evaluation

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/data_structures/slides/Chapter_07_Stacks.html

INTRODUCTION

Learning Outcomes

The LIFO model

Explain the last-in, first-out behavior of a stack.

Implementation

Implement a stack with an array and with a linked list.

Applications

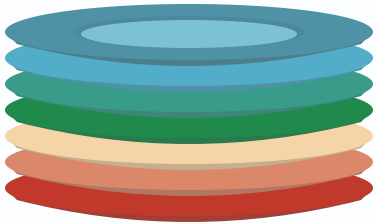
Use stacks to parse expressions and manage method calls.

Cost analysis

Analyze the time and space cost of the basic stack operations.

The Stack

- Items are added and removed at one end only: the **top**.
- Think of a pile of plates or boxes.
- To reach the bottom item, you must remove everything above it, one at a time.
- The rule is **LIFO** (last in, first out), or equivalently **FILO** (first in, last out).



A new plate goes on top, and you take plates from the top.

Undo
The most recent action is undone first.

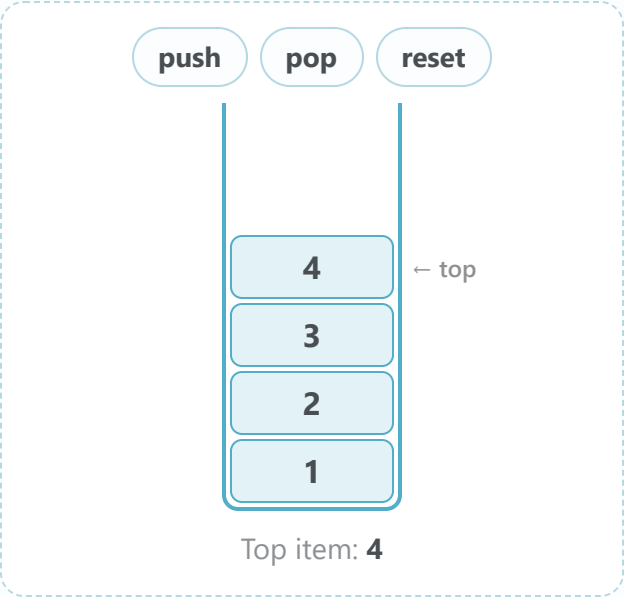
Browser "Back"
Returns to the most recently visited page.

Function calls
The most recently called function finishes first.

Basic Operations

Operation	Description
`push(x)`	Adds a new item x on top.
`pop()`	Removes and returns the top item.
`top()`	Returns the top item **without removing** it (Java: `peek()`).
`isEmpty()`	Is the stack empty?
`size()`	Returns the number of items.

Users see only the top item; there is **no** direct access to the items below it.



push and pop, Step by Step

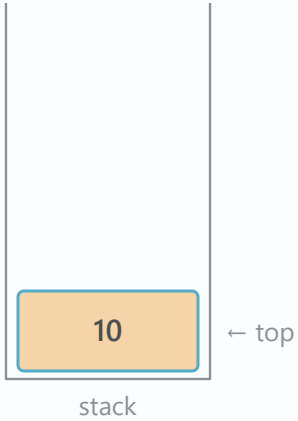
Operation	Returned	size



The stack starts empty: **size = 0**, nothing on top.

push and pop, Step by Step

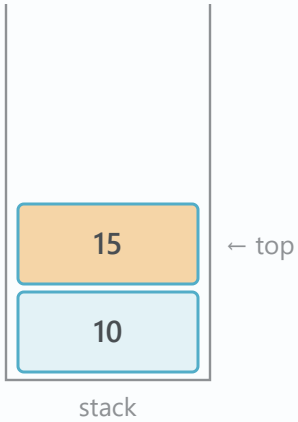
Operation	Returned	size
`push(10)`	–	1



push(10): 10 goes onto the empty stack; the top is now 10.

push and pop, Step by Step

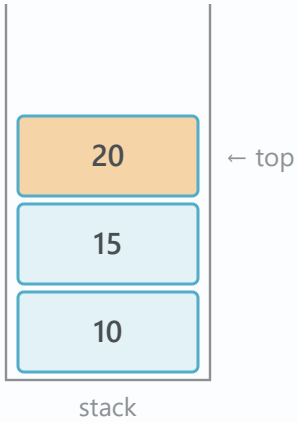
Operation	Returned	size
`push(10)`	–	1
`push(15)`	–	2



push(15): 15 goes on top of 10.

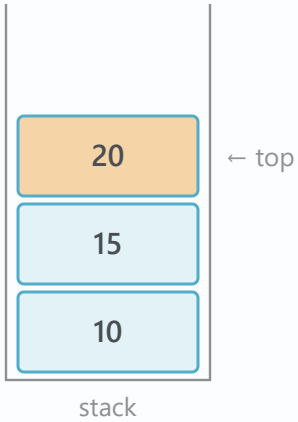
push and pop, Step by Step

Operation	Returned	size
`push(10)`	–	1
`push(15)`	–	2
`push(20)`	–	3



push(20): 20 goes on top; **size = 3**.

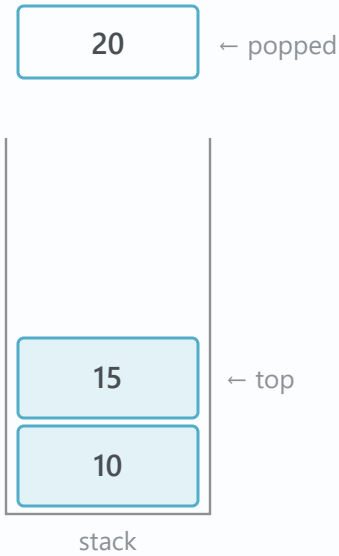
push and pop, Step by Step



Operation	Returned	size
`push(10)`	–	1
`push(15)`	–	2
`push(20)`	–	3
`top()`	20	3

top(): reads the top item, 20, but does **not remove** it.

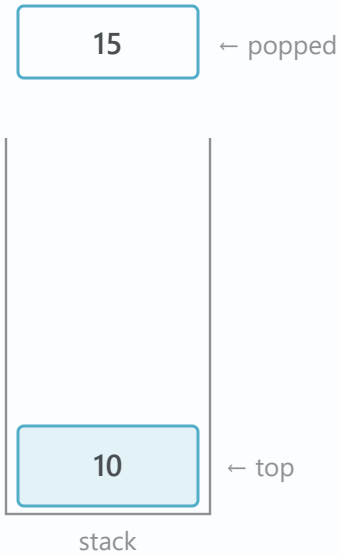
push and pop, Step by Step



Operation	Returned	size
`push(10)`	–	1
`push(15)`	–	2
`push(20)`	–	3
`top()`	20	3
`pop()`	20	2

pop(): 20, the last in, is the first out (LIFO).

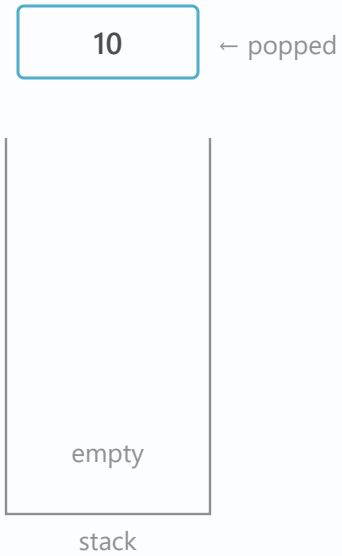
push and pop, Step by Step



Operation	Returned	size
`push(10)`	–	1
`push(15)`	–	2
`push(20)`	–	3
`top()`	20	3
`pop()`	20	2
`pop()`	15	1

pop(): removes 15; the top is now 10.

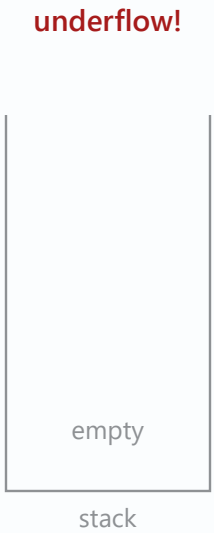
push and pop, Step by Step



Operation	Returned	size
`push(10)`	–	1
`push(15)`	–	2
`push(20)`	–	3
`top()`	20	3
`pop()`	20	2
`pop()`	15	1
`pop()`	10	0

pop(): removes 10; the stack is empty again.

push and pop, Step by Step



Operation	Returned	size
`push(10)`	–	1
`push(15)`	–	2
`push(20)`	–	3
`top()`	20	3
`pop()`	20	2
`pop()`	15	1
`pop()`	10	0
`pop()`	**error!**	0

pop() on an empty stack → **underflow** error.

The Stack ADT as a Java Interface

```
public interface Stack<E> {
  int size();
  boolean isEmpty();
  void push(E e);
  E top();     // null if empty
  E pop();     // throws if empty
}
```

- **ADT = what?** The interface states the operations and what they mean, not how the data is stored.
- **Implementation = how?** `ArrayStack<E> implements Stack<E>`, or a linked-list class.
- Code written against the interface lets you **swap** the implementation without touching the callers.

On an empty stack	`pop()`	`top()` / `peek()`
Our `Stack<E>` (above)	throws an exception	returns `null`
`java.util.Stack`	`EmptyStackException`	`EmptyStackException`
`java.util.ArrayDeque`	`NoSuchElementException`	returns `null`

The same abstract operation handles "empty" differently from library to library, so check `isEmpty()` first. Our interface shares its name with `java.util.Stack`; don't `import` both in the same file.

Algorithms for the Operations (Array)

The stack lives in an array; `top` is the index of the top item, and an empty stack has **top = −1**.

push(x)

```
begin
  if stack is full   // top = cap−1
    error: overflow
  else
    increment top by 1
    stack[top] ← x
end
```

pop()

```
begin
  if stack is empty    // top = −1
    error: underflow
  else
    x ← stack[top]; decrement top by 1
    return x
end
```

top()

```
begin
  if stack is empty return null
  else return stack[top]
end
```

isEmpty() · size()

```
isEmpty: true if top = −1, else false
size:    return top + 1
```

Time and Space Complexity

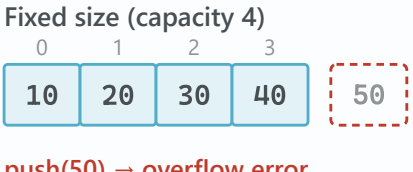
Operation	Time	Why?
`push`	O(1)	only the top changes
`pop`	O(1)	only the top changes
`top`	O(1)	the top is known directly
`isEmpty`, `size`	O(1)	a counter / index is kept
searching for an item	O(n)	items must be removed one by one

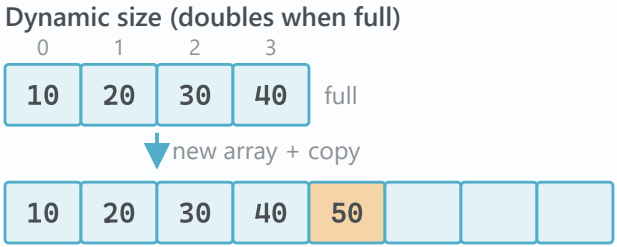
- The basic operations take the same time **no matter** how many items the stack holds.
- A stack of n items uses **O(n)** memory; the operations need only O(1) extra.
- A dynamic array grows when it fills up, so push is **amortized O(1)**.

A stack buys speed by restricting access: because of the “top only” rule, no basic operation ever shifts or scans the items.

IMPLEMENTATION

Kinds of Stacks





Fixed-Size Stack

- Capacity is set up front; it can neither grow nor shrink.
- Pushing onto a full stack → **overflow**.
- Popping an empty stack → **underflow**.
- Backed by a fixed-size **array**.

Dynamic-Size Stack

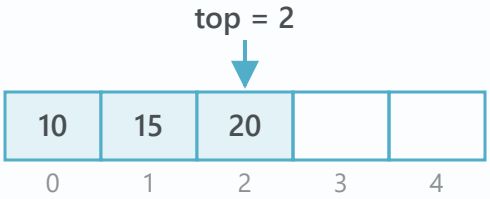
- Grows and shrinks at run time.
- Increases its capacity automatically when full (e.g., doubles it).
- May shrink when it holds far fewer items than its capacity.
- Implemented with a **dynamic array** or a **linked list**.

Popping an empty stack (underflow) is an error in both kinds; overflow happens only in a fixed-size stack.

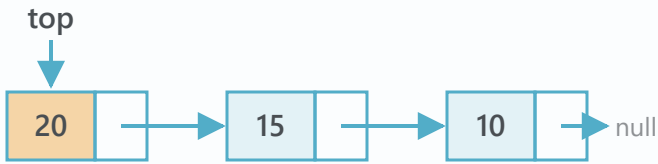
IMPLEMENTATION

Ways to Implement a Stack

The stack ADT can be implemented with an **array** or a **linked list**.



array: push → top++ , pop → top−−



linked list: push/pop at the head

Array-Based

`push` : increment top, then write the item to `arr[top]` .
`pop` : read `arr[top]` , then decrement top.

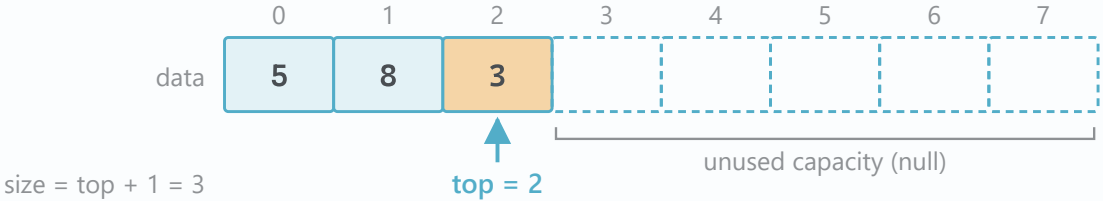
Linked-List-Based

`push` : add a new node at the **head** of the list.
`pop` : remove the head node and return its value.

IMPLEMENTATION

Array-Based Stack

This is also called a **bounded stack**: its capacity is fixed when it is created and never changes.



Advantages

+ Simple and easy to implement.
+ No per-node references → saves memory.
+ Memory use is tightly bounded, which makes it safer against attacks that try to exhaust memory.
+ Items are contiguous → cache-friendly.

Watch out for

– Check that the stack is not full before pushing.
– Too small a capacity overflows; too large wastes memory.
– In an array of objects, set a popped cell to `null`; otherwise the object lingers in memory.

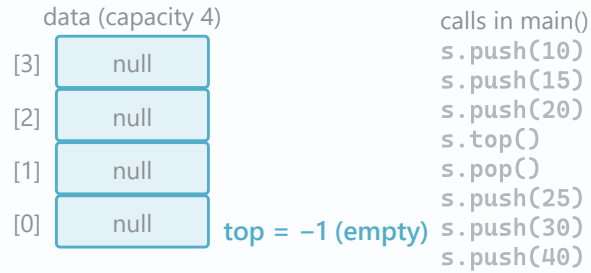
IMPLEMENTATION

Array-Based Stack: Java

```
public class ArrayStack<E> implements Stack<E> {
    private E[] data;
    private int top = -1;                    // -1 → empty stack
    public ArrayStack(int capacity) { data = (E[]) new Object[capacity]; }
    public boolean isEmpty() { return top == -1; }
    public int size()        { return top + 1; }
    public E top()           { return isEmpty() ? null : data[top]; }
    public void push(E x) {
        if (top == data.length - 1) throw new IllegalStateException("Overflow");
        data[++top] = x;
    }
    public E pop() {
        if (isEmpty()) throw new IllegalStateException("Underflow");
        E x = data[top];
        data[top--] = null;                  // drop the reference
        return x;
    }
}
```

Java cannot create a generic array directly, so we cast an `Object[]` (the compiler issues an "unchecked" warning). Every operation is O(1).

Code Trace: Array-Based Stack



`new ArrayStack<>(4)` : four cells, all **null**; an empty stack has **top = −1**.

Step 1 / 18

```
1  public void push(E x) {
2    if (top == data.length - 1)
3      throw new IllegalStateException("Overflow");
4    data[++top] = x;
5  }
6  public E pop() {
7    if (isEmpty())
8      throw new IllegalStateException("Underflow");
9    E x = data[top];
10   data[top--] = null;          // drop the reference
11   return x;
12 }
13 public E top() { return isEmpty() ? null : data[top]; }
```

VARIABLES

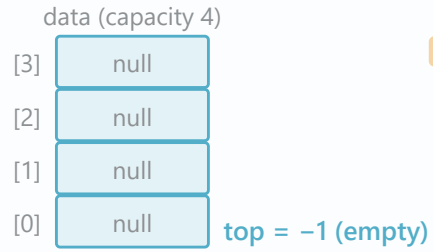
`top = −1`

`data = [null, null, null, null]`

CALL STACK

`main()`

Code Trace: Array-Based Stack



calls in main()

```
s.push(10)
s.push(15)
s.push(20)
s.top()
s.pop()
s.push(25)
s.push(30)
s.push(40)
```

push(10): full? top = -1, data.length − 1 = 3 → no, continue.

Step 2 / 18

```
1  public void push(E x) {
▶    if (top == data.length - 1)
3      throw new IllegalStateException("Overflow");
4    data[++top] = x;
5  }
6  public E pop() {
7    if (isEmpty())
8      throw new IllegalStateException("Underflow");
9    E x = data[top];
10   data[top--] = null;          // drop the reference
11   return x;
12 }
13 public E top() { return isEmpty() ? null : data[top]; }
```

VARIABLES

`top = -1`

`data = [null, null, null, null]`

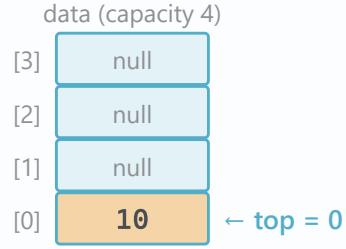
`x = 10`

CALL STACK

`s.push(10)`

`main()`

Code Trace: Array-Based Stack



calls in main()

```
s.push(10)     ✓
s.push(15)
s.push(20)
s.top()
s.pop()
s.push(25)
s.push(30)
s.push(40)
```

`data[++top] = x` : top goes -1 → **0** first, then data[0] = 10. One assignment → O(1).

Step 3 / 18

```
1  public void push(E x) {
2    if (top == data.length - 1)
3      throw new IllegalStateException("Overflow");
▶    data[++top] = x;
5  }
6  public E pop() {
7    if (isEmpty())
8      throw new IllegalStateException("Underflow");
9    E x = data[top];
10   data[top--] = null;          // drop the reference
11   return x;
12 }
13 public E top() { return isEmpty() ? null : data[top]; }
```

VARIABLES

`top = 0`

`data = [10, null, null, null]`

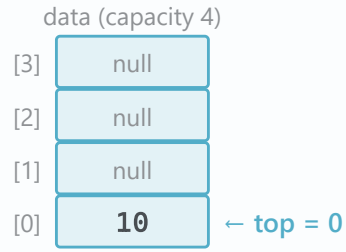
`x = 10`

CALL STACK

`s.push(10)`

`main()`

Code Trace: Array-Based Stack



calls in main()

```
s.push(10)     ✓
s.push(15)
s.push(20)
s.top()
s.pop()
s.push(25)
s.push(30)
s.push(40)
```

push(15): full? top = 0, data.length − 1 = 3 → no, continue.

Step 4 / 18

```
1  public void push(E x) {
▶    if (top == data.length - 1)
3      throw new IllegalStateException("Overflow");
4    data[++top] = x;
5  }
6  public E pop() {
7    if (isEmpty())
8      throw new IllegalStateException("Underflow");
9    E x = data[top];
10   data[top--] = null;           // drop the reference
11   return x;
12 }
13 public E top() { return isEmpty() ? null : data[top]; }
```

VARIABLES

`top = 0`

`data = [10, null, null, null]`

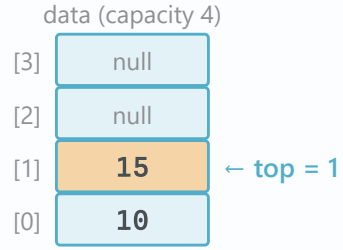
`x = 15`

CALL STACK

`s.push(15)`

`main()`

Code Trace: Array-Based Stack



calls in main()

```
s.push(10)      ✓
s.push(15)      ✓
s.push(20)
s.top()
s.pop()
s.push(25)
s.push(30)
s.push(40)
```

`data[++top] = x` : top goes 0 → **1** first, then data[1] = 15. One assignment → O(1).

Step 5 / 18

```
1  public void push(E x) {
2    if (top == data.length - 1)
3      throw new IllegalStateException("Overflow");
▶    data[++top] = x;
5  }
6  public E pop() {
7    if (isEmpty())
8      throw new IllegalStateException("Underflow");
9    E x = data[top];
10   data[top--] = null;          // drop the reference
11   return x;
12 }
13 public E top() { return isEmpty() ? null : data[top]; }
```

VARIABLES

`top = 1`

`data = [10, 15, null, null]`

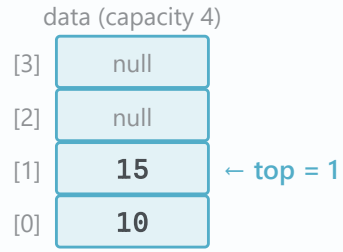
`x = 15`

CALL STACK

`s.push(15)`

`main()`

Code Trace: Array-Based Stack



calls in main()

```
s.push(10)      ✓
s.push(15)      ✓
s.push(20)
s.top()
s.pop()
s.push(25)
s.push(30)
s.push(40)
```

push(20): full? top = 1, data.length − 1 = 3 → no, continue.

Step 6 / 18

```
1  public void push(E x) {
▶    if (top == data.length - 1)
3      throw new IllegalStateException("Overflow");
4    data[++top] = x;
5  }
6  public E pop() {
7    if (isEmpty())
8      throw new IllegalStateException("Underflow");
9    E x = data[top];
10   data[top--] = null;          // drop the reference
11   return x;
12 }
13 public E top() { return isEmpty() ? null : data[top]; }
```

VARIABLES

`top = 1`

`data = [10, 15, null, null]`

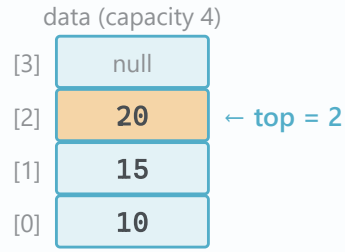
`x = 20`

CALL STACK

`s.push(20)`

`main()`

Code Trace: Array-Based Stack



calls in main()

```
s.push(10)      ✓
s.push(15)      ✓
s.push(20)      ✓
s.top()
s.pop()
s.push(25)
s.push(30)
s.push(40)
```

`data[++top] = x` : top goes 1 → **2** first, then data[2] = 20. One assignment → O(1).

Step 7 / 18

```
1  public void push(E x) {
2    if (top == data.length - 1)
3      throw new IllegalStateException("Overflow");
▶    data[++top] = x;
5  }
6  public E pop() {
7    if (isEmpty())
8      throw new IllegalStateException("Underflow");
9    E x = data[top];
10   data[top--] = null;          // drop the reference
11   return x;
12 }
13 public E top() { return isEmpty() ? null : data[top]; }
```

VARIABLES

`top = 2`

`data = [10, 15, 20, null]`

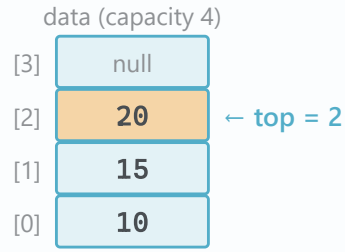
`x = 20`

CALL STACK

`s.push(20)`

`main()`

Code Trace: Array-Based Stack



calls in main()

```
s.push(10)      ✓
s.push(15)      ✓
s.push(20)      ✓
s.top()         → 20
s.pop()
s.push(25)
s.push(30)
s.push(40)
```

top(): the stack is not empty → returns **data[2] = 20**. The array and top **do not change**.

Step 8 / 18

```
1  public void push(E x) {
2    if (top == data.length - 1)
3      throw new IllegalStateException("Overflow");
4    data[++top] = x;
5  }
6  public E pop() {
7    if (isEmpty())
8      throw new IllegalStateException("Underflow");
9    E x = data[top];
10   data[top--] = null;          // drop the reference
11   return x;
12 }
▶ public E top() { return isEmpty() ? null : data[top]; }
```

VARIABLES

`top = 2`

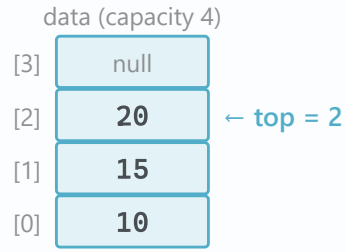
`data = [10, 15, 20, null]`

CALL STACK

`s.top()`

`main()`

Code Trace: Array-Based Stack



calls in main()

```
s.push(10)      ✓
s.push(15)      ✓
s.push(20)      ✓
s.top()         → 20
s.pop()
s.push(25)
s.push(30)
s.push(40)
```

pop(): isEmpty()? top = 2 ≠ −1 → not empty, continue.

Step 9 / 18

```
1  public void push(E x) {
2    if (top == data.length - 1)
3      throw new IllegalStateException("Overflow");
4    data[++top] = x;
5  }
6  public E pop() {
▶    if (isEmpty())
8      throw new IllegalStateException("Underflow");
9    E x = data[top];
10   data[top--] = null;          // drop the reference
11   return x;
12 }
13 public E top() { return isEmpty() ? null : data[top]; }
```

VARIABLES

`top = 2`

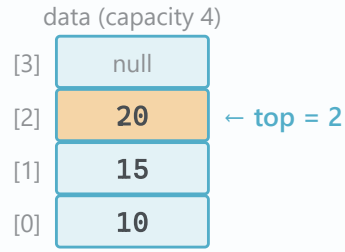
`data = [10, 15, 20, null]`

CALL STACK

`s.pop()`

`main()`

Code Trace: Array-Based Stack



calls in main()

```
s.push(10)      ✓
s.push(15)      ✓
s.push(20)      ✓
s.top()         → 20
s.pop()
s.push(25)
s.push(30)
s.push(40)
```

`E x = data[top]` → x = 20
(save the top item).

Step 10 / 18

```
1  public void push(E x) {
2    if (top == data.length - 1)
3      throw new IllegalStateException("Overflow");
4    data[++top] = x;
5  }
6  public E pop() {
7    if (isEmpty())
8      throw new IllegalStateException("Underflow");
▶    E x = data[top];
10   data[top--] = null;          // drop the reference
11   return x;
12 }
13 public E top() { return isEmpty() ? null : data[top]; }
```

VARIABLES

`top = 2`

`data = [10, 15, 20, null]`

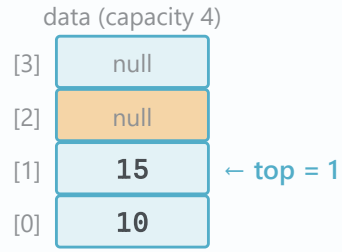
`x = 20`

CALL STACK

`s.pop()`

`main()`

Code Trace: Array-Based Stack



calls in main()

```
s.push(10)      ✓
s.push(15)      ✓
s.push(20)      ✓
s.top()         → 20
s.pop()
s.push(25)
s.push(30)
s.push(40)
```

`data[top--] = null` : clears the cell (the object is left to the garbage collector); top → **1**.

Step 11 / 18

```
1  public void push(E x) {
2    if (top == data.length - 1)
3      throw new IllegalStateException("Overflow");
4    data[++top] = x;
5  }
6  public E pop() {
7    if (isEmpty())
8      throw new IllegalStateException("Underflow");
9    E x = data[top];
▶    data[top--] = null;          // drop the reference
11   return x;
12 }
13 public E top() { return isEmpty() ? null : data[top]; }
```

VARIABLES

`top = 1`

`data = [10, 15, null, null]`

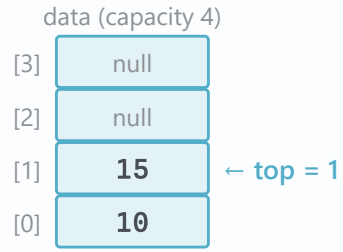
`x = 20`

CALL STACK

`s.pop()`

`main()`

Code Trace: Array-Based Stack



calls in main()

```
s.push(10)      ✓
s.push(15)      ✓
s.push(20)      ✓
s.top()         → 20
s.pop()         → 20
s.push(25)
s.push(30)
s.push(40)
```

return 20: last in, first out (LIFO).

Step 12 / 18

```
1  public void push(E x) {
2    if (top == data.length - 1)
3      throw new IllegalStateException("Overflow");
4    data[++top] = x;
5  }
6  public E pop() {
7    if (isEmpty())
8      throw new IllegalStateException("Underflow");
9    E x = data[top];
10   data[top--] = null;          // drop the reference
▶    return x;
12 }
13 public E top() { return isEmpty() ? null : data[top]; }
```

VARIABLES

`top = 1`

`data = [10, 15, null, null]`

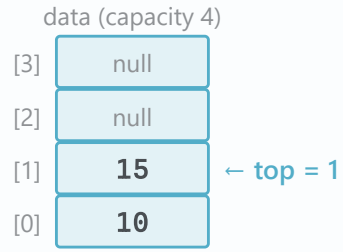
`x = 20`

CALL STACK

`s.pop()`

`main()`

Code Trace: Array-Based Stack



calls in main()

```
s.push(10)      ✓
s.push(15)      ✓
s.push(20)      ✓
s.top()         → 20
s.pop()         → 20
s.push(25)
s.push(30)
s.push(40)
```

push(25): full? top = 1, data.length − 1 = 3 → no, continue.

Step 13 / 18

```
1  public void push(E x) {
▶    if (top == data.length - 1)
3      throw new IllegalStateException("Overflow");
4    data[++top] = x;
5  }
6  public E pop() {
7    if (isEmpty())
8      throw new IllegalStateException("Underflow");
9    E x = data[top];
10   data[top--] = null;          // drop the reference
11   return x;
12 }
13 public E top() { return isEmpty() ? null : data[top]; }
```

VARIABLES

`top = 1`

`data = [10, 15, null, null]`

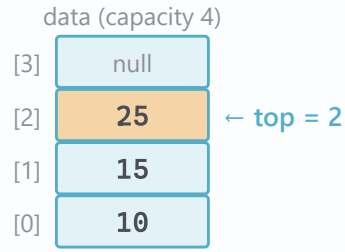
`x = 25`

CALL STACK

`s.push(25)`

`main()`

Code Trace: Array-Based Stack



calls in main()

```
s.push(10)      ✓
s.push(15)      ✓
s.push(20)      ✓
s.top()         → 20
s.pop()         → 20
s.push(25)      ✓
s.push(30)
s.push(40)
```

`data[++top] = x` : top goes 1 → **2** first, then data[2] = 25. One assignment → O(1).

Step 14 / 18

```
1  public void push(E x) {
2    if (top == data.length - 1)
3      throw new IllegalStateException("Overflow");
▶    data[++top] = x;
5  }
6  public E pop() {
7    if (isEmpty())
8      throw new IllegalStateException("Underflow");
9    E x = data[top];
10   data[top--] = null;          // drop the reference
11   return x;
12 }
13 public E top() { return isEmpty() ? null : data[top]; }
```

VARIABLES

`top = 2`

`data = [10, 15, 25, null]`

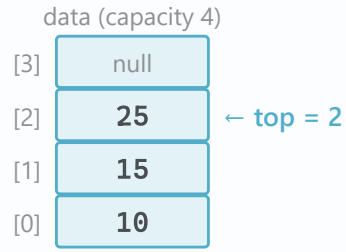
`x = 25`

CALL STACK

`s.push(25)`

`main()`

Code Trace: Array-Based Stack



calls in main()

```
s.push(10)     ✓
s.push(15)     ✓
s.push(20)     ✓
s.top()        → 20
s.pop()        → 20
s.push(25)     ✓
s.push(30)
s.push(40)
```

push(30): full? top = 2, data.length − 1 = 3 → no, continue.

Step 15 / 18

```
1  public void push(E x) {
▶    if (top == data.length - 1)
3      throw new IllegalStateException("Overflow");
4    data[++top] = x;
5  }
6  public E pop() {
7    if (isEmpty())
8      throw new IllegalStateException("Underflow");
9    E x = data[top];
10   data[top--] = null;          // drop the reference
11   return x;
12 }
13 public E top() { return isEmpty() ? null : data[top]; }
```

VARIABLES

`top = 2`

`data = [10, 15, 25, null]`

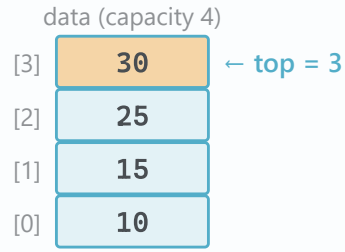
`x = 30`

CALL STACK

`s.push(30)`

`main()`

Code Trace: Array-Based Stack



calls in main()

```
s.push(10)      ✓
s.push(15)      ✓
s.push(20)      ✓
s.top()         → 20
s.pop()         → 20
s.push(25)      ✓
s.push(30)      ✓
s.push(40)
```

`data[++top] = x` : top goes 2 → **3** first, then data[3] = 30. One assignment → O(1).

Step 16 / 18

```
1  public void push(E x) {
2    if (top == data.length - 1)
3      throw new IllegalStateException("Overflow");
▶    data[++top] = x;
5  }
6  public E pop() {
7    if (isEmpty())
8      throw new IllegalStateException("Underflow");
9    E x = data[top];
10   data[top--] = null;          // drop the reference
11   return x;
12 }
13 public E top() { return isEmpty() ? null : data[top]; }
```

VARIABLES

`top = 3`

`data = [10, 15, 25, 30]`

`x = 30`

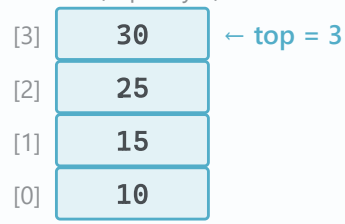
CALL STACK

`s.push(30)`

`main()`

Code Trace: Array-Based Stack

data (capacity 4)



calls in main()

```
s.push(10)      ✓
s.push(15)      ✓
s.push(20)      ✓
s.top()         → 20
s.pop()         → 20
s.push(25)      ✓
s.push(30)      ✓
s.push(40)
```

push(40): full? top = 3, data.length − 1 = 3 → **yes, full!**

Step 17 / 18

```
1  public void push(E x) {
▶    if (top == data.length - 1)
3      throw new IllegalStateException("Overflow");
4    data[++top] = x;
5  }
6  public E pop() {
7    if (isEmpty())
8      throw new IllegalStateException("Underflow");
9    E x = data[top];
10   data[top--] = null;           // drop the reference
11   return x;
12 }
13 public E top() { return isEmpty() ? null : data[top]; }
```

VARIABLES

`top = 3`

`data = [10, 15, 25, 30]`

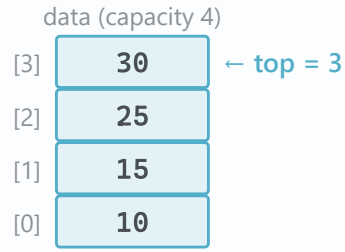
`x = 40`

CALL STACK

`s.push(40)`

`main()`

Code Trace: Array-Based Stack



calls in main()

```
s.push(10)      ✓
s.push(15)      ✓
s.push(20)      ✓
s.top()         → 20
s.pop()         → 20
s.push(25)      ✓
s.push(30)      ✓
s.push(40)      ✗ Overflow
```

The array is full: **IllegalStateException("Overflow")** is thrown. This is **overflow** in a fixed-capacity stack; a dynamic array would grow here instead.

Step 18 / 18

```
1  public void push(E x) {
2    if (top == data.length - 1)
▶      throw new IllegalStateException("Overflow");
4    data[++top] = x;
5  }
6  public E pop() {
7    if (isEmpty())
8      throw new IllegalStateException("Underflow");
9    E x = data[top];
10   data[top--] = null;          // drop the reference
11   return x;
12 }
13 public E top() { return isEmpty() ? null : data[top]; }
```

VARIABLES

`top = 3`

`data = [10, 15, 25, 30]`

`x = 40`

CALL STACK

`s.push(40)`

`main()`

IMPLEMENTATION

Dynamic Array: Doubling When Full

```
public void push(E x) {
  if (top == data.length - 1)
    resize(2 * data.length);
  data[++top] = x;
}
private void resize(int newCap) {
  E[] bigger = (E[]) new Object[newCap];
  for (int i = 0; i <= top; i++)
    bigger[i] = data[i];
  data = bigger;
}
```

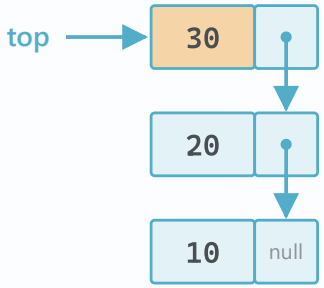
- When the array is full, we allocate one twice as large and copy the items over, so a push that grows the array takes **O(n)**.
- Growth is rare: over n pushes the copies total 2 + 4 + 8 + … < 2n, and the total cost is < 3n → push is **amortized O(1)**.
- Adding a fixed number of cells (+c) each time gives $O(n^2)$ in total, O(n) per push; capacity must grow **proportionally**.

Amortized cost is an average over a long sequence of operations, not the cost of any single one.

IMPLEMENTATION

Linked-List Stack

- Items live in a chain of linked nodes → the size is flexible.
- The size changes at run time as needed; **no overflow** (until memory runs out).
- Each node stores a reference (next), which costs extra memory.
- No random access, but reaching the top item (the head of the list) is quick and simple.
- Insertion and removal happen at the head, so both are O(1).



push / pop only
at the head (top) node

```
class Node {
    int data;
    Node next;
    Node(int data) { this.data = data; }   // next = null
}
```

Why insert at the head? Deleting from the tail of a singly linked list requires finding the node before the tail → O(n).

IMPLEMENTATION · STEP THROUGH THE CODE

Code Trace: `push` on a Linked List

top = null

Start: **top = null**, size = 0 → empty stack.

node = [data | next]

Step 1 / 10

```
▶ Node top;          // top of the stack
▶ int size;
3
4 public void push(int data) {
5   Node temp = new Node(data);
6   temp.next = top;
7   top = temp;
8   size++;
9 }
```

VARIABLES

`top = null` `size = 0`

CALL STACK

`main()`

Code Trace: `push` on a Linked List



push(10): creates `new Node(10)`; **temp** points to it, and its next field defaults to null.

node = [data | next]

Step 2 / 10

```
1 Node top;          // top of the
  stack
2 int size;
3
4 public void push(int data) {
▶   Node temp = new Node(data);
6   temp.next = top;
7   top = temp;
8   size++;
9 }
```

VARIABLES

`data = 10` `temp = node(10)` `top = null` `size = 0`

CALL STACK

`push(10)`

`main()`

Code Trace: `push` on a Linked List



`temp.next = top`: top is null, so next stays **null** (the new node stands alone).

node = [data | next]

Step 3 / 10

```
1 Node top;          // top of the
  stack
2 int size;
3
4 public void push(int data) {
5   Node temp = new Node(data);
▶   temp.next = top;
7   top = temp;
8   size++;
9 }
```

VARIABLES

`data = 10` `temp = node(10)` `temp.next = null` `top = null` `size = 0`

CALL STACK

`push(10)`

`main()`

Code Trace: `push` on a Linked List



`top = temp` : the new node is now the top of the stack.

node = [data | next]

Step 4 / 10

```
1 Node top;          // top of the
  stack
2 int size;
3
4 public void push(int data) {
5   Node temp = new Node(data);
6   temp.next = top;
▶   top = temp;
8   size++;
9 }
```

VARIABLES

`data = 10` `temp = node(10)` `temp.next = null`

`top = node(10)` `size = 0`

CALL STACK

`push(10)`

`main()`

Code Trace: `push` on a Linked List



`size++` → 1. The method returns; the local **temp** goes away, and the node is reached through top.

node = [data | next]

Step 5 / 10

```
1 Node top;          // top of the
  stack
2 int size;
3
4 public void push(int data) {
5   Node temp = new Node(data);
6   temp.next = top;
7   top = temp;
▶   size++;
9 }
```

VARIABLES

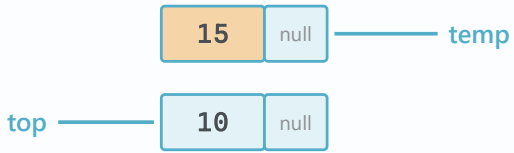
`data = 10` `temp = node(10)` `temp.next = null` `top = node(10)` `size = 1`

CALL STACK

`push(10)`

`main()`

Code Trace: `push` on a Linked List



push(15): creates node 15. It is **not linked** to the list yet; top still points to 10.

node = [data | next]

Step 6 / 10

```
1 Node top;          // top of the
  stack
2 int size;
3
4 public void push(int data) {
▶   Node temp = new Node(data);
6   temp.next = top;
7   top = temp;
8   size++;
9 }
```

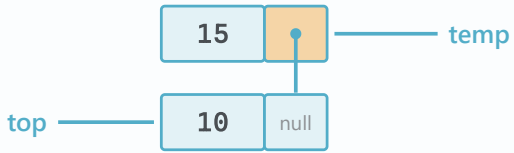
VARIABLES

`data = 15` `temp = node(15)` `top = node(10)` `size = 1`

CALL STACK

- **push(15)**
- main()

Code Trace: `push` on a Linked List



`temp.next = top`: 15 links to 10, the old top. We link first so the old list isn't lost.

node = [data | next]

Step 7 / 10

```
1 Node top;           // top of the
  stack
2 int size;
3
4 public void push(int data) {
5   Node temp = new Node(data);
▶   temp.next = top;
7   top = temp;
8   size++;
9 }
```

VARIABLES

`data = 15` `temp = node(15)`

`temp.next = node(10)` `top = node(10)`

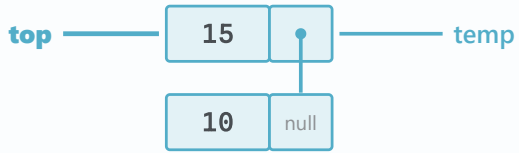
`size = 1`

CALL STACK

`push(15)`

`main()`

Code Trace: `push` on a Linked List



node = [data | next]

`top = temp` : the new top is 15. Insertion at the **head**, no shifting → O(1).

Step 8 / 10

```
1 Node top;          // top of the
  stack
2 int size;
3
4 public void push(int data) {
5   Node temp = new Node(data);
6   temp.next = top;
▶   top = temp;
8   size++;
9 }
```

VARIABLES

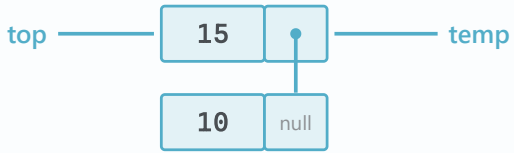
`data = 15` `temp = node(15)` `temp.next = node(10)` `top = node(15)` `size = 1`

CALL STACK

`push(15)`

`main()`

Code Trace: `push` on a Linked List



`size++` → 2.

node = [data | next]

Step 9 / 10

```
1 Node top;          // top of the
  stack
2 int size;
3
4 public void push(int data) {
5   Node temp = new Node(data);
6   temp.next = top;
7   top = temp;
▶   size++;
9 }
```

VARIABLES

`data = 15` `temp = node(15)`

`temp.next = node(10)` `top = node(15)`

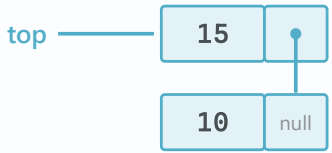
`size = 2`

CALL STACK

push(15)

main()

Code Trace: `push` on a Linked List



node = [data | next]

push returns. Stack (top to bottom): **15** → **10**. The last item pushed is on top.

Step 10 / 10

```
1 Node top;           // top of the
  stack
2 int size;
3
4 public void push(int data) {
5   Node temp = new Node(data);
6   temp.next = top;
7   top = temp;
8   size++;
▶ }
```

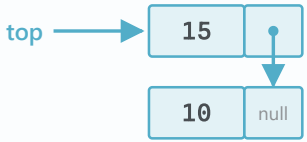
VARIABLES

`top = node(15)` `size = 2`

CALL STACK

`main()`

Code Trace: `pop` on a Linked List



The stack holds 15 and 10. The loop will call pop() **3 times**.

node = [data | next]

Step 1 / 15

```
1  public int pop() {
2    if (top == null)
3      throw new RuntimeException("empty");
4    int result = top.data;
5    top = top.next;
6    size--;
7    return result;
8  }
9  // in main (stack: 15 → 10):
▶ for (int k = 0; k < 3; k++)
11   System.out.println(pop());
```

VARIABLES

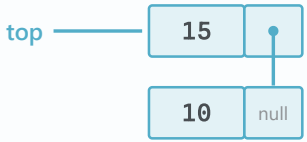
`k = 0` `top = node(15)` `size = 2`

CALL STACK

`main()`

OUTPUT

Code Trace: `pop` on a Linked List



node = [data | next]

pop(): check for empty first; top is not null → continue.

Step 2 / 15

```
1  public int pop() {
▶    if (top == null)
3      throw new RuntimeException("empty");
4    int result = top.data;
5    top = top.next;
6    size--;
7    return result;
8  }
9  // in main (stack: 15 → 10):
10 for (int k = 0; k < 3; k++)
11   System.out.println(pop());
```

VARIABLES

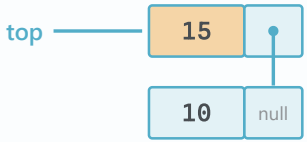
`k = 0` `top = node(15)` `size = 2`

CALL STACK

- `pop()`
- `main()`

OUTPUT

Code Trace: `pop` on a Linked List



node = [data | next]

`result = top.data` → save **15** (before the node is unlinked).

Step 3 / 15

```
1  public int pop() {
2    if (top == null)
3      throw new RuntimeException("empty");
▶    int result = top.data;
5    top = top.next;
6    size--;
7    return result;
8  }
9  // in main (stack: 15 → 10):
10 for (int k = 0; k < 3; k++)
11   System.out.println(pop());
```

VARIABLES

`k = 0` `top = node(15)` `result = 15` `size = 2`

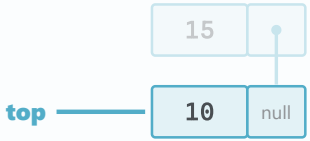
CALL STACK

`pop()`

`main()`

OUTPUT

Code Trace: `pop` on a Linked List



node = [data | next]

`top = top.next` : top moves down one → **node(10)**. Nothing refers to the old top node anymore.

Step 4 / 15

```
1  public int pop() {
2    if (top == null)
3      throw new RuntimeException("empty");
4    int result = top.data;
▶    top = top.next;
6    size--;
7    return result;
8  }
9  // in main (stack: 15 → 10):
10 for (int k = 0; k < 3; k++)
11   System.out.println(pop());
```

VARIABLES

`k = 0` `top = node(10)` `result = 15` `size = 2`

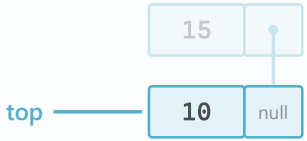
CALL STACK

`pop()`

`main()`

OUTPUT

Code Trace: `pop` on a Linked List



`size--` → 1.

node = [data | next]

Step 5 / 15

```
1  public int pop() {
2    if (top == null)
3      throw new RuntimeException("empty");
4    int result = top.data;
5    top = top.next;
▶    size--;
7    return result;
8  }
9  // in main (stack: 15 → 10):
10 for (int k = 0; k < 3; k++)
11   System.out.println(pop());
```

VARIABLES

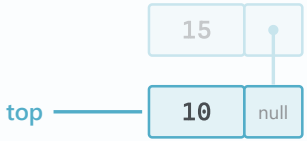
`k = 0` `top = node(10)` `result = 15` `size = 1`

CALL STACK

- `pop()`
- `main()`

OUTPUT

Code Trace: `pop` on a Linked List



node = [data | next]

return 15. The garbage collector reclaims the unreachable node.

Step 6 / 15

```
1  public int pop() {
2    if (top == null)
3      throw new RuntimeException("empty");
4    int result = top.data;
5    top = top.next;
6    size--;
▶    return result;
8  }
9  // in main (stack: 15 → 10):
10 for (int k = 0; k < 3; k++)
11   System.out.println(pop());
```

VARIABLES

`k = 0` `top = node(10)` `result = 15` `size = 1`

CALL STACK

`pop()`

`main()`

OUTPUT

Code Trace: `pop` on a Linked List



main prints the returned value, **15**.

node = [data | next]

Step 7 / 15

```
1  public int pop() {
2    if (top == null)
3      throw new RuntimeException("empty");
4    int result = top.data;
5    top = top.next;
6    size--;
7    return result;
8  }
9  // in main (stack: 15 → 10):
10 for (int k = 0; k < 3; k++)
▶    System.out.println(pop());
```

VARIABLES

`k = 0` `top = node(10)` `size = 1`

CALL STACK

`main()`

OUTPUT

```
15
```

Code Trace: `pop` on a Linked List



pop(): check for empty first; top is not null → continue.

node = [data | next]

Step 8 / 15

```
1  public int pop() {
▶    if (top == null)
3      throw new RuntimeException("empty");
4    int result = top.data;
5    top = top.next;
6    size--;
7    return result;
8  }
9  // in main (stack: 15 → 10):
10 for (int k = 0; k < 3; k++)
11   System.out.println(pop());
```

VARIABLES

`k = 1` `top = node(10)` `size = 1`

CALL STACK

- `pop()`
- `main()`

OUTPUT

```
15
```

Code Trace: `pop` on a Linked List



`result = top.data` → save **10** (before the node is unlinked).

node = [data | next]

Step 9 / 15

```
1  public int pop() {
2    if (top == null)
3      throw new RuntimeException("empty");
▶    int result = top.data;
5    top = top.next;
6    size--;
7    return result;
8  }
9  // in main (stack: 15 → 10):
10 for (int k = 0; k < 3; k++)
11   System.out.println(pop());
```

VARIABLES

`k = 1` `top = node(10)` `result = 10` `size = 1`

CALL STACK

- `pop()`
- `main()`

OUTPUT

```
15
```

Code Trace: `pop` on a Linked List



`top = top.next` : top moves down one → **null**. Nothing refers to the old top node anymore.

node = [data | next]

Step 10 / 15

```
1  public int pop() {
2    if (top == null)
3      throw new RuntimeException("empty");
4    int result = top.data;
▶    top = top.next;
6    size--;
7    return result;
8  }
9  // in main (stack: 15 → 10):
10 for (int k = 0; k < 3; k++)
11   System.out.println(pop());
```

VARIABLES

`k = 1` `top = null` `result = 10` `size = 1`

CALL STACK

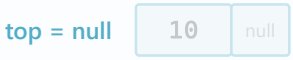
`pop()`

`main()`

OUTPUT

```
15
```

Code Trace: `pop` on a Linked List



`size--` → 0.

node = [data | next]

Step 11 / 15

```
1  public int pop() {
2    if (top == null)
3      throw new RuntimeException("empty");
4    int result = top.data;
5    top = top.next;
▶    size--;
7    return result;
8  }
9  // in main (stack: 15 → 10):
10 for (int k = 0; k < 3; k++)
11   System.out.println(pop());
```

VARIABLES

`k = 1` `top = null` `result = 10` `size = 0`

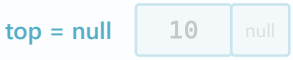
CALL STACK

- `pop()`
- `main()`

OUTPUT

```
15
```

Code Trace: `pop` on a Linked List



return 10. The garbage collector reclaims the unreachable node.

node = [data | next]

Step 12 / 15

```
1  public int pop() {
2    if (top == null)
3      throw new RuntimeException("empty");
4    int result = top.data;
5    top = top.next;
6    size--;
▶    return result;
8  }
9  // in main (stack: 15 → 10):
10 for (int k = 0; k < 3; k++)
11   System.out.println(pop());
```

VARIABLES

`k = 1` `top = null` `result = 10` `size = 0`

CALL STACK

`pop()`

`main()`

OUTPUT

```
15
```

Code Trace: `pop` on a Linked List

top = null

main prints the returned value, **10**.

node = [data | next]

Step 13 / 15

```
1  public int pop() {
2    if (top == null)
3      throw new RuntimeException("empty");
4    int result = top.data;
5    top = top.next;
6    size--;
7    return result;
8  }
9  // in main (stack: 15 → 10):
10 for (int k = 0; k < 3; k++)
▶    System.out.println(pop());
```

VARIABLES

`k = 1` `top = null` `size = 0`

CALL STACK

`main()`

OUTPUT

```
15
10
```

Code Trace: `pop` on a Linked List

top = null

pop(): **top == null** → the stack is empty!

node = [data | next]

Step 14 / 15

```
1  public int pop() {
▶    if (top == null)
3      throw new RuntimeException("empty");
4    int result = top.data;
5    top = top.next;
6    size--;
7    return result;
8  }
9  // in main (stack: 15 → 10):
10 for (int k = 0; k < 3; k++)
11   System.out.println(pop());
```

VARIABLES

`k = 2` `top = null` `size = 0`

CALL STACK

- `pop()`
- `main()`

OUTPUT

```
15
10
```

Code Trace: `pop` on a Linked List

top = null

node = [data | next]

An **underflow** exception stops the program. Without the check, `top.data` would throw a NullPointerException.

Step 15 / 15

```
1  public int pop() {
2    if (top == null)
▶      throw new RuntimeException("empty");
4    int result = top.data;
5    top = top.next;
6    size--;
7    return result;
8  }
9  // in main (stack: 15 → 10):
10 for (int k = 0; k < 3; k++)
11   System.out.println(pop());
```

VARIABLES

`k = 2` `top = null` `size = 0`

CALL STACK

- `pop()`
- `main()`

OUTPUT

```
15
10
RuntimeException: empty
```

IMPLEMENTATION

Array or Linked List?

Criterion	Array-based	Linked-list-based
Size	fixed capacity (a dynamic array grows)	flexible, changes at run time
push / pop	O(1) (amortized O(1) with growth)	O(1)
Overflow	possible	only if memory runs out
Memory per item	data only	data + next reference
Wasted space	unused capacity	none
Java	`ArrayDeque`, `Stack` (Vector-based)	`LinkedList` (as a Deque)

In new Java code, use `Deque<Integer> s = new ArrayDeque<>();` with `push` / `pop` / `peek`; the legacy `Stack` class is synchronized and therefore slower.

APPLICATIONS

Where Stacks Are Used

Expressions
Converting and evaluating infix, prefix, and postfix; bracket matching.

Undo / redo
Undo and redo in text editors.

Browser history
Two stacks drive the “Back” and “Forward” buttons.

The call stack
Function calls, local variables, and return addresses.

Backtracking
Maze solving, n-queens, the knight's tour.

Algorithms
Towers of Hanoi, tree traversal, stock span, histogram problems.

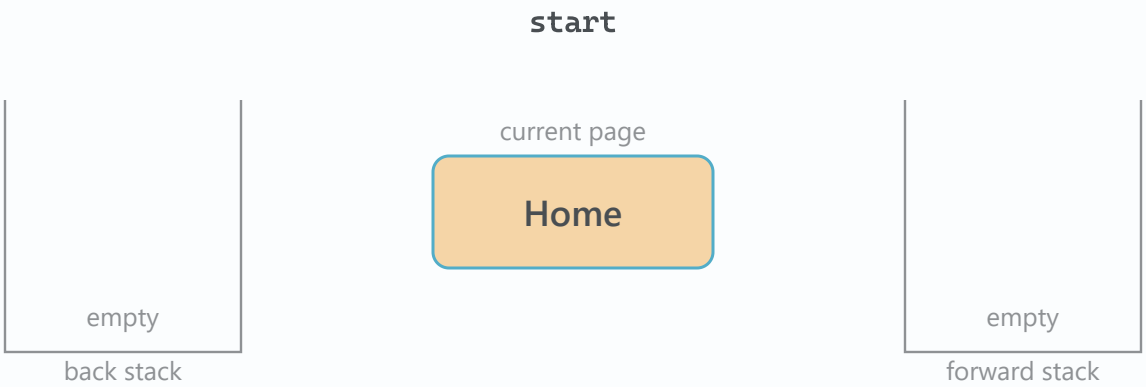
Graph algorithms
Depth-first search, topological sort, strongly connected components.

Reversing a string
Characters go in one order and come out in reverse.

Next greater element
For each item, the first larger value to its right.

APPLICATIONS

Browser Back and Forward: Two Stacks



The browser opens on **Home**; both stacks are empty.

Step 1 / 8

Rules: **visit** → push onto back, clear forward · **back** → push onto forward, pop back · **forward** → push onto back, pop forward. Each is O(1). **Undo / redo** in a text editor follows the same pattern.

APPLICATIONS

Browser Back and Forward: Two Stacks



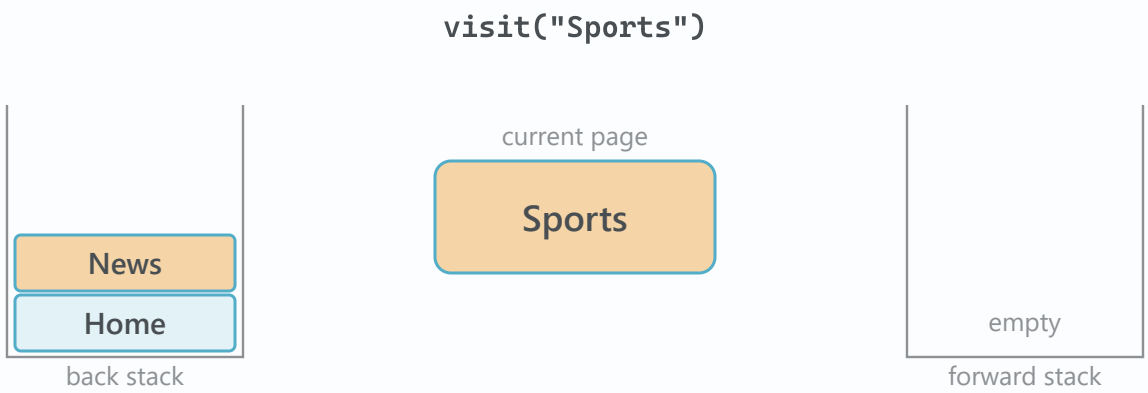
visit(News): **push** the old page onto the back stack and open the new one.

Step 2 / 8

Rules: **visit** → push onto back, clear forward · **back** → push onto forward, pop back · **forward** → push onto back, pop forward. Each is O(1). **Undo / redo** in a text editor follows the same pattern.

APPLICATIONS

Browser Back and Forward: Two Stacks



visit(Sports): **push** the old page onto the back stack and open the new one.

Step 3 / 8

Rules: **visit** → push onto back, clear forward · **back** → push onto forward, pop back · **forward** → push onto back, pop forward. Each is O(1). **Undo / redo** in a text editor follows the same pattern.

APPLICATIONS

Browser Back and Forward: Two Stacks



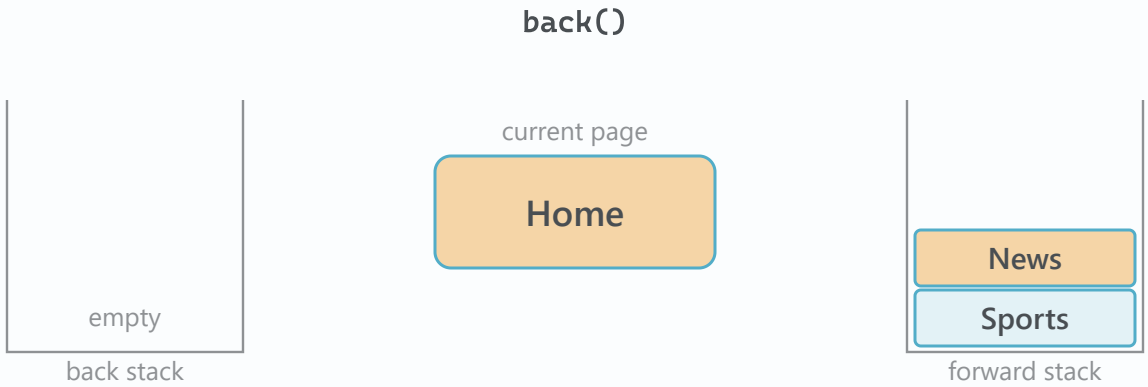
back(): push the current page onto the forward stack; **pop** the back stack → News.

Step 4 / 8

Rules: **visit** → push onto back, clear forward · **back** → push onto forward, pop back · **forward** → push onto back, pop forward. Each is O(1). **Undo / redo** in a text editor follows the same pattern.

APPLICATIONS

Browser Back and Forward: Two Stacks



back(): push the current page onto the forward stack; **pop** the back stack → Home.

Step 5 / 8

Rules: **visit** → push onto back, clear forward · **back** → push onto forward, pop back · **forward** → push onto back, pop forward. Each is O(1). **Undo / redo** in a text editor follows the same pattern.

APPLICATIONS

Browser Back and Forward: Two Stacks

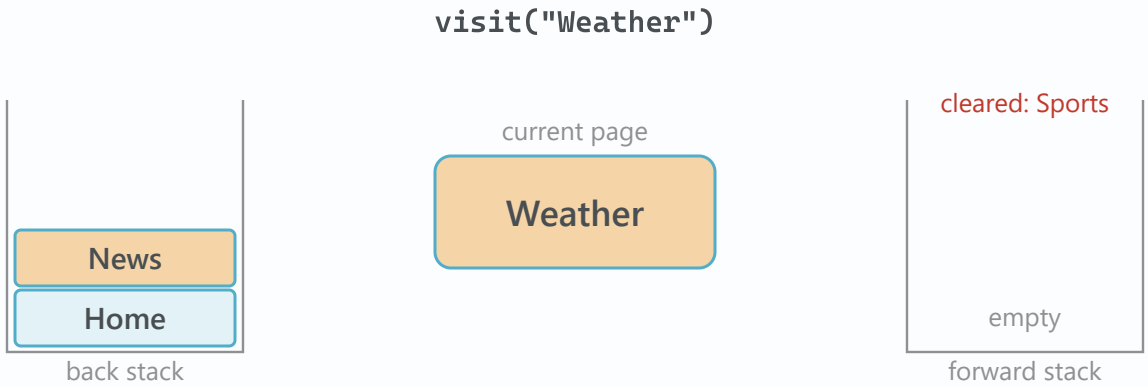


forward(): push the current page onto the back stack; pop the forward stack → News.

Rules: **visit** → push onto back, clear forward · **back** → push onto forward, pop back · **forward** → push onto back, pop forward. Each is O(1). **Undo / redo** in a text editor follows the same pattern.

APPLICATIONS

Browser Back and Forward: Two Stacks



visit(Weather): **push** the old page onto the back stack and open the new one. A new path begins, so the **forward stack is cleared**.

Rules: **visit** → push onto back, clear forward · **back** → push onto forward, pop back · **forward** → push onto back, pop forward. Each is O(1). **Undo / redo** in a text editor follows the same pattern.

APPLICATIONS

Browser Back and Forward: Two Stacks

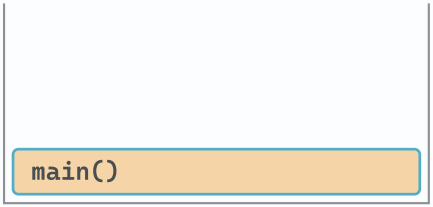


back(): push the current page onto the forward stack; **pop** the back stack → News.

Step 8 / 8

Rules: **visit** → push onto back, clear forward · **back** → push onto forward, pop back · **forward** → push onto back, pop forward. Each is O(1). **Undo / redo** in a text editor follows the same pattern.

The Call Stack: `fact(3)` Step by Step



main() is running

frame = parameters
+ local variables
+ return address

main is running and is about to call `fact(3)`. Every call pushes a new **stack frame**.

Step 1 / 10

```
1 int fact(int n) {
2   if (n <= 1)
3     return 1;
4   return n * fact(n - 1);
5 }
6 // in main:
▶ System.out.println(fact(3));
```

VARIABLES

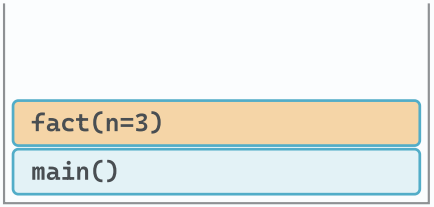
—

CALL STACK

`main()`

OUTPUT

The Call Stack: `fact(3)` Step by Step



call stack

push: fact(3)

stack depth = 1

The frame for `fact(3)` is **pushed**. `n <= 1` ? 3 <= 1 → **false**.

Step 2 / 10

```
1 int fact(int n) {
▶   if (n <= 1)
3     return 1;
4   return n * fact(n - 1);
5 }
6 // in main:
7 System.out.println(fact(3));
```

VARIABLES

`n = 3`

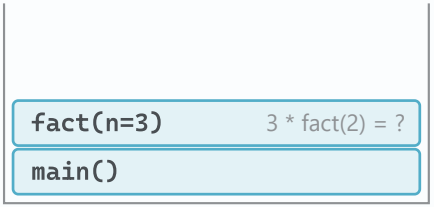
CALL STACK

`fact(n=3)`

`main()`

OUTPUT

The Call Stack: `fact(3)` Step by Step



call stack

fact(3) suspended

fact(2) must finish first

`3 * fact(2)` can't be computed until fact(2) finishes. This frame waits, **suspended**.

Step 3 / 10

```
1 int fact(int n) {
2   if (n <= 1)
3     return 1;
▶   return n * fact(n - 1);
5 }
6 // in main:
7 System.out.println(fact(3));
```

VARIABLES

`n = 3`

CALL STACK

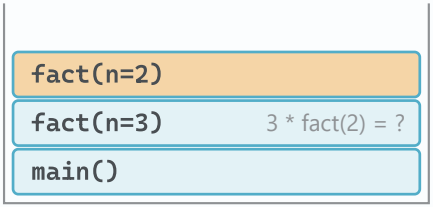
`fact(n=3)`

`main()`

OUTPUT

APPLICATIONS · STEP THROUGH THE CODE

The Call Stack: `fact(3)` Step by Step



call stack

push: fact(2)

stack depth = 2

The frame for `fact(2)` is **pushed**. `n <= 1` ? 2 <= 1 → **false**.

Step 4 / 10

```
1 int fact(int n) {
▶   if (n <= 1)
3     return 1;
4   return n * fact(n - 1);
5 }
6 // in main:
7 System.out.println(fact(3));
```

VARIABLES

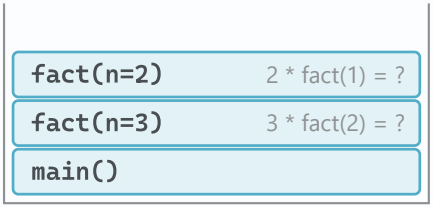
`n = 2`

CALL STACK

- `fact(n=2)`
- `fact(n=3)`
- `main()`

OUTPUT

The Call Stack: `fact(3)` Step by Step



call stack

fact(2) suspended

fact(1) must finish first

`2 * fact(1)` can't be computed until fact(1) finishes. This frame waits, **suspended**.

Step 5 / 10

```
1 int fact(int n) {
2   if (n <= 1)
3     return 1;
▶   return n * fact(n - 1);
5 }
6 // in main:
7 System.out.println(fact(3));
```

VARIABLES

`n = 2`

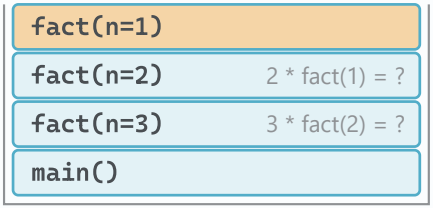
CALL STACK

- `fact(n=2)`
- `fact(n=3)`
- `main()`

OUTPUT

APPLICATIONS · STEP THROUGH THE CODE

The Call Stack: `fact(3)` Step by Step



call stack

push: fact(1)

stack depth = 3

The frame for `fact(1)` is **pushed**. `n <= 1` ? 1 <= 1 → **true**.

Step 6 / 10

```
1 int fact(int n) {
▶   if (n <= 1)
3     return 1;
4   return n * fact(n - 1);
5 }
6 // in main:
7 System.out.println(fact(3));
```

VARIABLES

`n = 1`

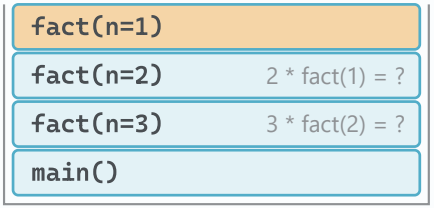
CALL STACK

- `fact(n=1)`
- `fact(n=2)`
- `fact(n=3)`
- `main()`

OUTPUT

APPLICATIONS · STEP THROUGH THE CODE

The Call Stack: `fact(3)` Step by Step



call stack

base case

return 1

Base case: **return 1**. Now the stack starts **unwinding**.

Step 7 / 10

```
1 int fact(int n) {
2   if (n <= 1)
▶     return 1;
4   return n * fact(n - 1);
5 }
6 // in main:
7 System.out.println(fact(3));
```

VARIABLES

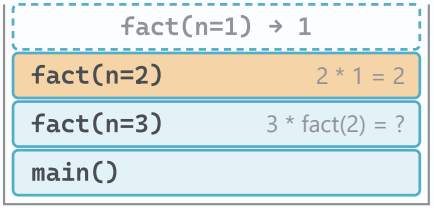
`n = 1`

CALL STACK

- `fact(n=1)`
- `fact(n=2)`
- `fact(n=3)`
- `main()`

OUTPUT

The Call Stack: `fact(3)` Step by Step



call stack

pop: fact(1) → 1

2 * 1 = 2

fact(1) is **popped** and returns 1.
fact(2) resumes: 2 * 1 = **2**.

Step 8 / 10

```
1 int fact(int n) {
2   if (n <= 1)
3     return 1;
▶   return n * fact(n - 1);
5 }
6 // in main:
7 System.out.println(fact(3));
```

VARIABLES

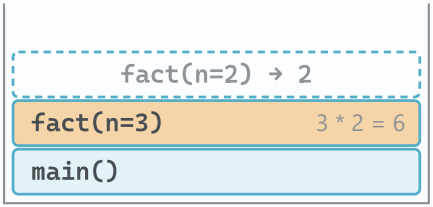
`n = 2` `fact(1) = 1`

CALL STACK

- `fact(n=2)`
- `fact(n=3)`
- `main()`

OUTPUT

The Call Stack: `fact(3)` Step by Step



call stack

pop: fact(2) → 2

3 * 2 = 6

fact(2) is **popped** and returns 2.
fact(3) resumes: 3 * 2 = **6**.

Step 9 / 10

```
1 int fact(int n) {
2   if (n <= 1)
3     return 1;
▶   return n * fact(n - 1);
5 }
6 // in main:
7 System.out.println(fact(3));
```

VARIABLES

`n = 3` `fact(2) = 2`

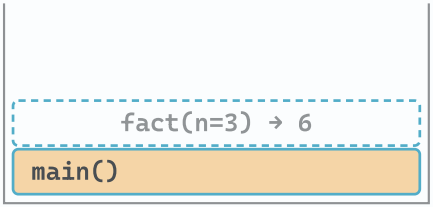
CALL STACK

`fact(n=3)`

`main()`

OUTPUT

The Call Stack: `fact(3)` Step by Step



call stack

pop: fact(3) → 6

println(6)

fact(3) = **6** is printed. Depth n → **O(n)** memory; recursion that goes too deep throws `StackOverflowError`.

Step 10 / 10

```
1 int fact(int n) {
2   if (n <= 1)
3     return 1;
4   return n * fact(n - 1);
5 }
6 // in main:
▶ System.out.println(fact(3));
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

```
6
```

Reversing a String, Step by Step

`str = "ABCD"`

main calls `reverse("ABCD")`.

Step 1 / 13

```
1  String reverse(String str) {
2      Deque<Character> stack = new ArrayDeque<>
   ();
3      char[] chars = str.toCharArray();
4      for (char c : chars)
5          stack.push(c);
6      for (int i = 0; i < str.length(); i++)
7          chars[i] = stack.pop();
8      return new String(chars);
9  }
10 // in main:
▶  System.out.println(reverse("ABCD"));
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

Reversing a String, Step by Step

`str = "ABCD"`



Create an empty stack of characters.

Step 2 / 13

```
1  String reverse(String str) {
▶      Deque<Character> stack = new ArrayDeque<>
   ();
3      char[] chars = str.toCharArray();
4      for (char c : chars)
5          stack.push(c);
6      for (int i = 0; i < str.length(); i++)
7          chars[i] = stack.pop();
8      return new String(chars);
9  }
10 // in main:
11 System.out.println(reverse("ABCD"));
```

VARIABLES

`str = "ABCD"` `stack = []`

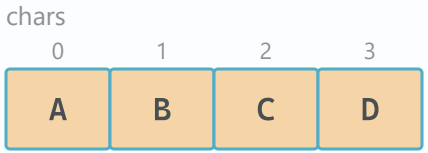
CALL STACK

`reverse("ABCD")`

`main()`

OUTPUT

Reversing a String, Step by Step





Copy the string into a char array (we will overwrite it).

Step 3 / 13

```
1  String reverse(String str) {
2      Deque<Character> stack = new ArrayDeque<>
   ();
▶      char[] chars = str.toCharArray();
4      for (char c : chars)
5          stack.push(c);
6      for (int i = 0; i < str.length(); i++)
7          chars[i] = stack.pop();
8      return new String(chars);
9  }
10 // in main:
11 System.out.println(reverse("ABCD"));
```

VARIABLES

`str = "ABCD"` `chars = [A, B, C, D]` `stack = []`

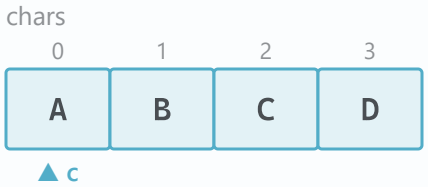
CALL STACK

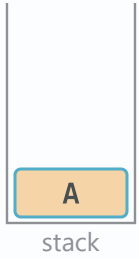
`reverse("ABCD")`

`main()`

OUTPUT

Reversing a String, Step by Step





First loop: c = 'A' → **push('A')**.

Step 4 / 13

```
1  String reverse(String str) {
2      Deque<Character> stack = new ArrayDeque<>
   ();
3      char[] chars = str.toCharArray();
4      for (char c : chars)
▶          stack.push(c);
6      for (int i = 0; i < str.length(); i++)
7          chars[i] = stack.pop();
8      return new String(chars);
9  }
10 // in main:
11 System.out.println(reverse("ABCD"));
```

VARIABLES

`str = "ABCD"` `chars = [A, B, C, D]` `c = A` `stack = [A]`

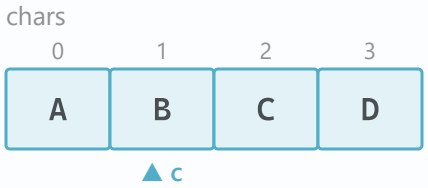
CALL STACK

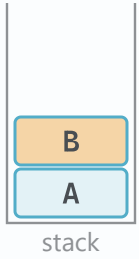
`reverse("ABCD")`

`main()`

OUTPUT

Reversing a String, Step by Step





First loop: c = 'B' → **push('B')**.

Step 5 / 13

```
1  String reverse(String str) {
2      Deque<Character> stack = new ArrayDeque<>
   ();
3      char[] chars = str.toCharArray();
4      for (char c : chars)
▶          stack.push(c);
6      for (int i = 0; i < str.length(); i++)
7          chars[i] = stack.pop();
8      return new String(chars);
9  }
10 // in main:
11 System.out.println(reverse("ABCD"));
```

VARIABLES

`str = "ABCD"` `chars = [A, B, C, D]` `c = B` `stack = [A, B]`

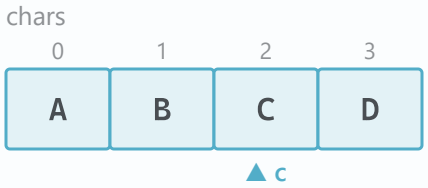
CALL STACK

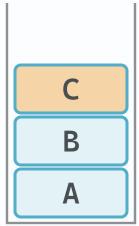
`reverse("ABCD")`

`main()`

OUTPUT

Reversing a String, Step by Step





First loop: c = 'C' → **push('C')**.

Step 6 / 13

```
1  String reverse(String str) {
2      Deque<Character> stack = new ArrayDeque<>
   ();
3      char[] chars = str.toCharArray();
4      for (char c : chars)
▶          stack.push(c);
6      for (int i = 0; i < str.length(); i++)
7          chars[i] = stack.pop();
8      return new String(chars);
9  }
10 // in main:
11 System.out.println(reverse("ABCD"));
```

VARIABLES

`str = "ABCD"` `chars = [A, B, C, D]`

`c = C` `stack = [A, B, C]`

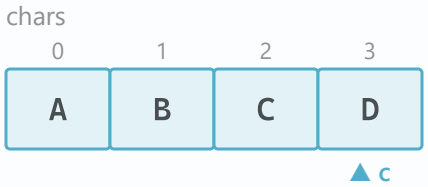
CALL STACK

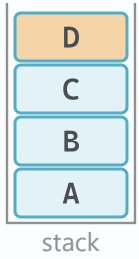
`reverse("ABCD")`

`main()`

OUTPUT

Reversing a String, Step by Step





First loop: c = 'D' → **push('D')**. Every character is on the stack, the last one on top.

Step 7 / 13

```
1  String reverse(String str) {
2      Deque<Character> stack = new ArrayDeque<>
   ();
3      char[] chars = str.toCharArray();
4      for (char c : chars)
▶          stack.push(c);
6      for (int i = 0; i < str.length(); i++)
7          chars[i] = stack.pop();
8      return new String(chars);
9  }
10 // in main:
11 System.out.println(reverse("ABCD"));
```

VARIABLES

`str = "ABCD"` `chars = [A, B, C, D]`

`c = D` `stack = [A, B, C, D]`

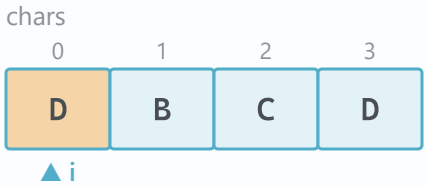
CALL STACK

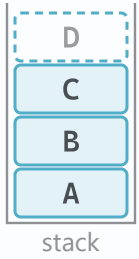
`reverse("ABCD")`

`main()`

OUTPUT

Reversing a String, Step by Step





Second loop, i = 0: **pop()** → 'D', chars[0] = 'D'. Items come out in **reverse** order.

Step 8 / 13

```
1  String reverse(String str) {
2      Deque<Character> stack = new ArrayDeque<>
   ();
3      char[] chars = str.toCharArray();
4      for (char c : chars)
5          stack.push(c);
6      for (int i = 0; i < str.length(); i++)
▶          chars[i] = stack.pop();
8      return new String(chars);
9  }
10 // in main:
11 System.out.println(reverse("ABCD"));
```

VARIABLES

`str = "ABCD"` `chars = [D, B, C, D]` `i = 0` `stack = [A, B, C]`

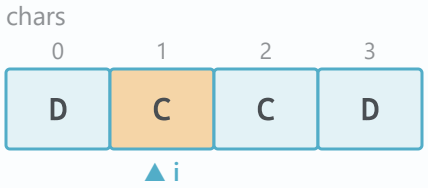
CALL STACK

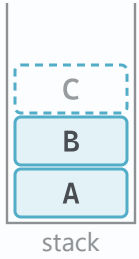
`reverse("ABCD")`

`main()`

OUTPUT

Reversing a String, Step by Step





Second loop, i = 1: **pop()** → 'C', chars[1] = 'C'. Items come out in **reverse** order.

Step 9 / 13

```
1  String reverse(String str) {
2      Deque<Character> stack = new ArrayDeque<>
   ();
3      char[] chars = str.toCharArray();
4      for (char c : chars)
5          stack.push(c);
6      for (int i = 0; i < str.length(); i++)
▶          chars[i] = stack.pop();
8      return new String(chars);
9  }
10 // in main:
11 System.out.println(reverse("ABCD"));
```

VARIABLES

`str = "ABCD"` `chars = [D, C, C, D]` `i = 1` `stack = [A, B]`

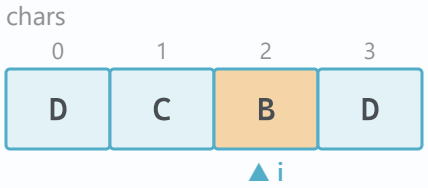
CALL STACK

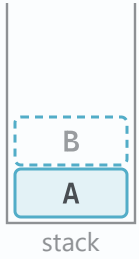
`reverse("ABCD")`

`main()`

OUTPUT

Reversing a String, Step by Step





Second loop, i = 2: **pop()** → 'B', chars[2] = 'B'. Items come out in **reverse** order.

Step 10 / 13

```
1  String reverse(String str) {
2      Deque<Character> stack = new ArrayDeque<>
   ();
3      char[] chars = str.toCharArray();
4      for (char c : chars)
5          stack.push(c);
6      for (int i = 0; i < str.length(); i++)
▶          chars[i] = stack.pop();
8      return new String(chars);
9  }
10 // in main:
11 System.out.println(reverse("ABCD"));
```

VARIABLES

`str = "ABCD"` `chars = [D, C, B, D]` `i = 2` `stack = [A]`

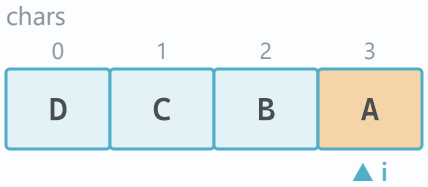
CALL STACK

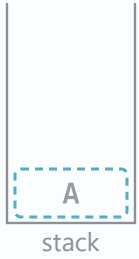
`reverse("ABCD")`

`main()`

OUTPUT

Reversing a String, Step by Step





Second loop, i = 3: **pop()** → 'A', chars[3] = 'A'. Items come out in **reverse** order.

Step 11 / 13

```
1  String reverse(String str) {
2      Deque<Character> stack = new ArrayDeque<>
   ();
3      char[] chars = str.toCharArray();
4      for (char c : chars)
5          stack.push(c);
6      for (int i = 0; i < str.length(); i++)
▶          chars[i] = stack.pop();
8      return new String(chars);
9  }
10 // in main:
11 System.out.println(reverse("ABCD"));
```

VARIABLES

`str = "ABCD"` `chars = [D, C, B, A]` `i = 3` `stack = []`

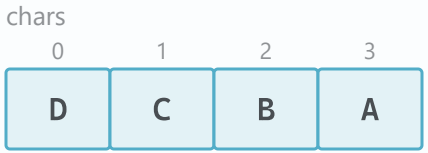
CALL STACK

`reverse("ABCD")`

`main()`

OUTPUT

Reversing a String, Step by Step





return "DCBA". Time and extra memory are **O(n)**.

Step 12 / 13

```
1  String reverse(String str) {
2      Deque<Character> stack = new ArrayDeque<>
   ();
3      char[] chars = str.toCharArray();
4      for (char c : chars)
5          stack.push(c);
6      for (int i = 0; i < str.length(); i++)
7          chars[i] = stack.pop();
▶      return new String(chars);
9  }
10 // in main:
11 System.out.println(reverse("ABCD"));
```

VARIABLES

`str = "ABCD"` `chars = [D, C, B, A]` `stack = []`

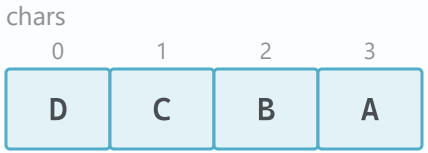
CALL STACK

`reverse("ABCD")`

`main()`

OUTPUT

Reversing a String, Step by Step





main prints the result: **DCBA**.

Step 13 / 13

```
1  String reverse(String str) {
2      Deque<Character> stack = new ArrayDeque<>
   ();
3      char[] chars = str.toCharArray();
4      for (char c : chars)
5          stack.push(c);
6      for (int i = 0; i < str.length(); i++)
7          chars[i] = stack.pop();
8      return new String(chars);
9  }
10 // in main:
▶  System.out.println(reverse("ABCD"));
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

DCBA

APPLICATIONS

Bracket Matching

Let the string S contain only the characters `( ) { } [ ]`. S is **balanced** if and only if

1. every opening bracket is closed by a bracket of the **same type**, and
2. brackets are closed in the **right order** (the last one opened is the first one closed → LIFO!).

Algorithm: keep the unmatched opening brackets on a stack and scan S from left to right:

- Opening bracket → **push** it.
- Closing bracket → if the stack is empty, **unbalanced**; otherwise **pop**, and if the popped bracket is a different type, **unbalanced**.
- At the end of S, an **empty stack means balanced**; anything left over is an unclosed bracket.

`{[()]}`	`([)]`	`(()`	`())`
balanced	wrong order	unclosed	extra closer

APPLICATIONS

Bracket Matching: Java

```java
boolean isBalanced(String s) {
  Deque<Character> stack = new ArrayDeque<>();
  for (char c : s.toCharArray()) {
    if (c == '(' || c == '[' || c == '{') {
      stack.push(c);
    } else {
      if (stack.isEmpty()) return false;
      char open = stack.pop();
      if (!matches(open, c)) return false;
    }
  }
  return stack.isEmpty();
}
```

Helper method

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

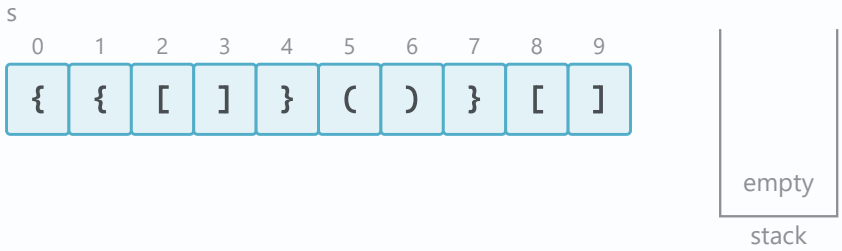
Empty-stack check

Without it, input such as `"())"` or `"]"` pops an empty stack → exception.

Complexity

O(n) time; in the worst case the stack needs **O(n)** memory.

Code Trace: `isBalanced("{{[]}()}[]")`



Create an empty stack to hold the unmatched opening brackets.

Step 1 / 17

```
1  boolean isBalanced(String s) {
▶    Deque<Character> stack = new ArrayDeque<>();
3    for (char c : s.toCharArray()) {
4      if (c == '(' || c == '[' || c == '{') {
5        stack.push(c);
6      } else {
7        if (stack.isEmpty()) return false;
8        char open = stack.pop();
9        if (!matches(open, c)) return false;
10     }
11   }
12   return stack.isEmpty();
13 }
```

VARIABLES

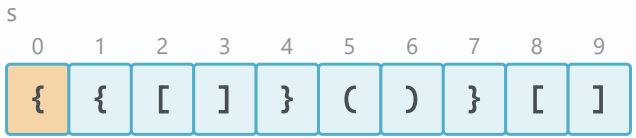
`s = "{{[]}()}[]"` `stack = []`

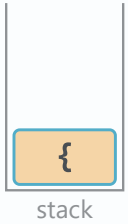
CALL STACK

`isBalanced("{{[]}()}[]")`

`main()`

Code Trace: `isBalanced("{{[]}()}[]")`





'{' is an opening bracket → **push**. It waits for its closer.

Step 2 / 17

```
1  boolean isBalanced(String s) {
2    Deque<Character> stack = new ArrayDeque<>();
3    for (char c : s.toCharArray()) {
▶      if (c == '(' || c == '[' || c == '{') {
▶        stack.push(c);
6      } else {
7        if (stack.isEmpty()) return false;
8        char open = stack.pop();
9        if (!matches(open, c)) return false;
10     }
11   }
12   return stack.isEmpty();
13 }
```

VARIABLES

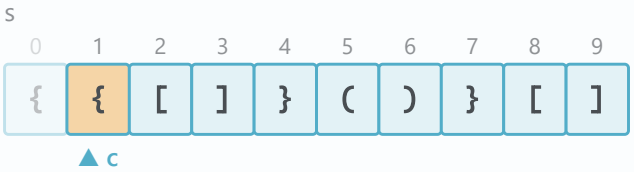
`s = "{{[]}()}[]"` `c = '{'` `stack = ['{']`

CALL STACK

`isBalanced("{{[]}()}[]")`

`main()`

Code Trace: `isBalanced("{{[]}()}[]")`



'{' is an opening bracket → **push**. It waits for its closer.

Step 3 / 17

```
1  boolean isBalanced(String s) {
2    Deque<Character> stack = new ArrayDeque<>();
3    for (char c : s.toCharArray()) {
▶      if (c == '(' || c == '[' || c == '{') {
▶        stack.push(c);
6      } else {
7        if (stack.isEmpty()) return false;
8        char open = stack.pop();
9        if (!matches(open, c)) return false;
10     }
11   }
12   return stack.isEmpty();
13 }
```

VARIABLES

`s = "{{[]}()}[]"` `c = '{'`

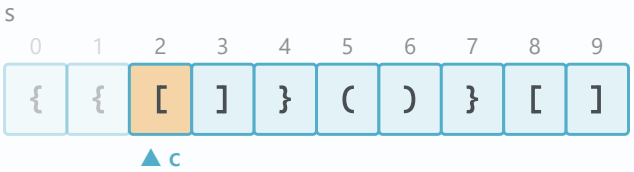
`stack = ['{', '{']`

CALL STACK

`isBalanced("{{[]}()}[]")`

`main()`

Code Trace: `isBalanced("{{[]}()}[]")`



'[' is an opening bracket → **push**. It waits for its closer.

Step 4 / 17

```
1  boolean isBalanced(String s) {
2    Deque<Character> stack = new ArrayDeque<>();
3    for (char c : s.toCharArray()) {
▶      if (c == '(' || c == '[' || c == '{') {
▶        stack.push(c);
6      } else {
7        if (stack.isEmpty()) return false;
8        char open = stack.pop();
9        if (!matches(open, c)) return false;
10     }
11   }
12   return stack.isEmpty();
13 }
```

VARIABLES

`s = "{{[]}()}[]"` `c = '['`

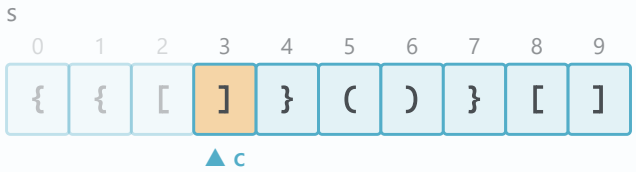
`stack = ['{', '{', '[']`

CALL STACK

`isBalanced("{{[]}()}[]")`

`main()`

Code Trace: `isBalanced("{{[]}()}[]")`



']' is a closing bracket; the stack is not empty → **pop** the most recent opener, '['.

Step 5 / 17

```
1  boolean isBalanced(String s) {
2    Deque<Character> stack = new ArrayDeque<>();
3    for (char c : s.toCharArray()) {
4      if (c == '(' || c == '[' || c == '{') {
5        stack.push(c);
6      } else {
▶        if (stack.isEmpty()) return false;
▶        char open = stack.pop();
9        if (!matches(open, c)) return false;
10     }
11   }
12   return stack.isEmpty();
13 }
```

VARIABLES

`s = "{{[]}()}[]"` `c = ']'`

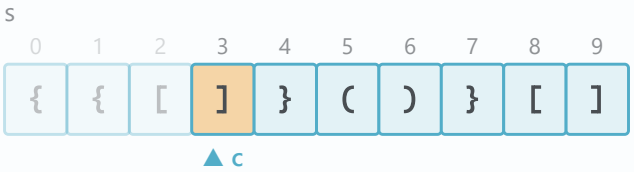
`open = '['` `stack = ['{', '{']`

CALL STACK

`isBalanced("{{[]}()}[]")`

`main()`

Code Trace: `isBalanced("{{[]}()}[]")`



matches('[', ']') → **true**: same type, right order. Continue.

Step 6 / 17

```
1  boolean isBalanced(String s) {
2    Deque<Character> stack = new ArrayDeque<>();
3    for (char c : s.toCharArray()) {
4      if (c == '(' || c == '[' || c == '{') {
5        stack.push(c);
6      } else {
7        if (stack.isEmpty()) return false;
8        char open = stack.pop();
▶        if (!matches(open, c)) return false;
10     }
11   }
12   return stack.isEmpty();
13 }
```

VARIABLES

`s = "{{[]}()}[]"` `c = ']'`

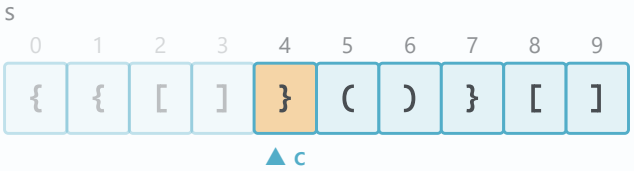
`open = '['` `stack = ['{', '{']`

CALL STACK

`isBalanced("{{[]}()}[]")`

`main()`

Code Trace: `isBalanced("{{[]}()}[]")`



'}' is a closing bracket; the stack is not empty → **pop** the most recent opener, '{'.

Step 7 / 17

```
1  boolean isBalanced(String s) {
2    Deque<Character> stack = new ArrayDeque<>();
3    for (char c : s.toCharArray()) {
4      if (c == '(' || c == '[' || c == '{') {
5        stack.push(c);
6      } else {
▶        if (stack.isEmpty()) return false;
▶        char open = stack.pop();
9        if (!matches(open, c)) return false;
10     }
11   }
12   return stack.isEmpty();
13 }
```

VARIABLES

`s = "{{[]}()}[]"` `c = '}'`

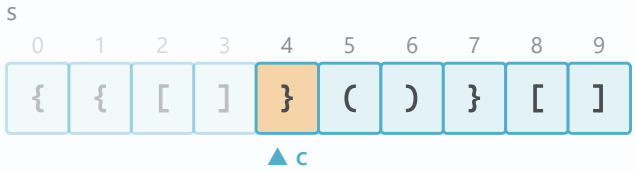
`open = '{'` `stack = ['{']`

CALL STACK

`isBalanced("{{[]}()}[]")`

`main()`

Code Trace: `isBalanced("{{[]}()}[]")`



matches('{', '}') → **true**: same type, right order. Continue.

Step 8 / 17

```
1  boolean isBalanced(String s) {
2    Deque<Character> stack = new ArrayDeque<>();
3    for (char c : s.toCharArray()) {
4      if (c == '(' || c == '[' || c == '{') {
5        stack.push(c);
6      } else {
7        if (stack.isEmpty()) return false;
8        char open = stack.pop();
▶        if (!matches(open, c)) return false;
10     }
11   }
12   return stack.isEmpty();
13 }
```

VARIABLES

`s = "{{[]}()}[]"` `c = '}'`

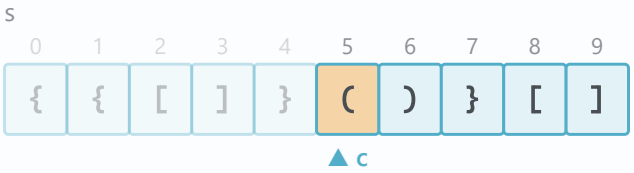
`open = '{'` `stack = ['{']`

CALL STACK

`isBalanced("{{[]}()}[]")`

`main()`

Code Trace: `isBalanced("{{[]}()}[]")`



'(' is an opening bracket → **push**. It waits for its closer.

Step 9 / 17

```
1  boolean isBalanced(String s) {
2    Deque<Character> stack = new ArrayDeque<>();
3    for (char c : s.toCharArray()) {
▶      if (c == '(' || c == '[' || c == '{') {
▶        stack.push(c);
6      } else {
7        if (stack.isEmpty()) return false;
8        char open = stack.pop();
9        if (!matches(open, c)) return false;
10     }
11   }
12   return stack.isEmpty();
13 }
```

VARIABLES

`s = "{{[]}()}[]"` `c = '('`

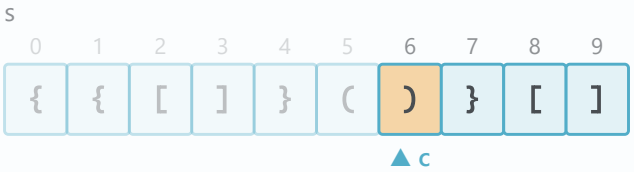
`stack = ['{', '(']`

CALL STACK

`isBalanced("{{[]}()}[]")`

`main()`

Code Trace: `isBalanced("{{[]}()}[]")`



')' is a closing bracket; the stack is not empty → **pop** the most recent opener, '('.

Step 10 / 17

```
1  boolean isBalanced(String s) {
2    Deque<Character> stack = new ArrayDeque<>();
3    for (char c : s.toCharArray()) {
4      if (c == '(' || c == '[' || c == '{') {
5        stack.push(c);
6      } else {
▶        if (stack.isEmpty()) return false;
▶        char open = stack.pop();
9        if (!matches(open, c)) return false;
10     }
11   }
12   return stack.isEmpty();
13 }
```

VARIABLES

`s = "{{[]}()}[]"` `c = ')'`

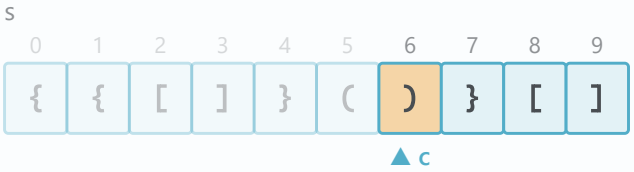
`open = '('` `stack = ['{']`

CALL STACK

`isBalanced("{{[]}()}[]")`

`main()`

Code Trace: `isBalanced("{{[]}()}[]")`



matches('(', ')') → **true**: same type, right order. Continue.

Step 11 / 17

```
1  boolean isBalanced(String s) {
2    Deque<Character> stack = new ArrayDeque<>();
3    for (char c : s.toCharArray()) {
4      if (c == '(' || c == '[' || c == '{') {
5        stack.push(c);
6      } else {
7        if (stack.isEmpty()) return false;
8        char open = stack.pop();
▶       if (!matches(open, c)) return false;
10     }
11   }
12   return stack.isEmpty();
13 }
```

VARIABLES

`s = "{{[]}()}[]"` `c = ')'`

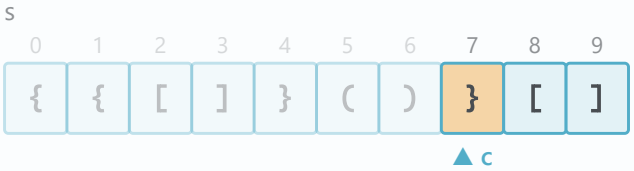
`open = '('` `stack = ['{']`

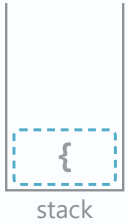
CALL STACK

`isBalanced("{{[]}()}[]")`

`main()`

Code Trace: `isBalanced("{{[]}()}[]")`





'}' is a closing bracket; the stack is not empty → **pop** the most recent opener, '{'.

Step 12 / 17

```
1  boolean isBalanced(String s) {
2    Deque<Character> stack = new ArrayDeque<>();
3    for (char c : s.toCharArray()) {
4      if (c == '(' || c == '[' || c == '{') {
5        stack.push(c);
6      } else {
▶        if (stack.isEmpty()) return false;
▶        char open = stack.pop();
9        if (!matches(open, c)) return false;
10     }
11   }
12   return stack.isEmpty();
13 }
```

VARIABLES

`s = "{{[]}()}[]"` `c = '}'`

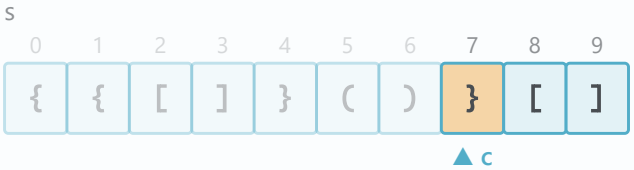
`open = '{'` `stack = []`

CALL STACK

`isBalanced("{{[]}()}[]")`

`main()`

Code Trace: `isBalanced("{{[]}()}[]")`



matches('{', '}') → **true**: same type, right order. Continue.

Step 13 / 17

```
1  boolean isBalanced(String s) {
2    Deque<Character> stack = new ArrayDeque<>();
3    for (char c : s.toCharArray()) {
4      if (c == '(' || c == '[' || c == '{') {
5        stack.push(c);
6      } else {
7        if (stack.isEmpty()) return false;
8        char open = stack.pop();
▶        if (!matches(open, c)) return false;
10     }
11   }
12   return stack.isEmpty();
13 }
```

VARIABLES

`s = "{{[]}()}[]"` `c = '}'`

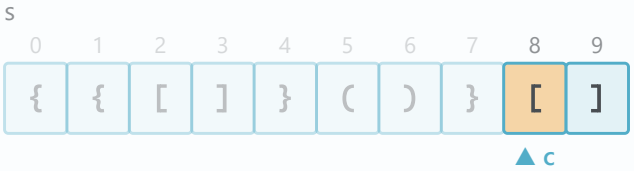
`open = '{'` `stack = []`

CALL STACK

`isBalanced("{{[]}()}[]")`

`main()`

Code Trace: `isBalanced("{{[]}()}[]")`



'[' is an opening bracket → **push**. It waits for its closer.

Step 14 / 17

```
1  boolean isBalanced(String s) {
2    Deque<Character> stack = new ArrayDeque<>();
3    for (char c : s.toCharArray()) {
▶      if (c == '(' || c == '[' || c == '{') {
▶        stack.push(c);
6      } else {
7        if (stack.isEmpty()) return false;
8        char open = stack.pop();
9        if (!matches(open, c)) return false;
10     }
11   }
12   return stack.isEmpty();
13 }
```

VARIABLES

`s = "{{[]}()}[]"` `c = '['` `stack = ['[']`

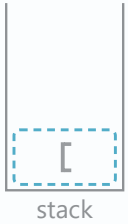
CALL STACK

`isBalanced("{{[]}()}[]")`

`main()`

Code Trace: `isBalanced("{{[]}()}[]")`





']' is a closing bracket; the stack is not empty → **pop** the most recent opener, '['.

Step 15 / 17

```
1  boolean isBalanced(String s) {
2    Deque<Character> stack = new ArrayDeque<>();
3    for (char c : s.toCharArray()) {
4      if (c == '(' || c == '[' || c == '{') {
5        stack.push(c);
6      } else {
▶        if (stack.isEmpty()) return false;
▶        char open = stack.pop();
9        if (!matches(open, c)) return false;
10     }
11   }
12   return stack.isEmpty();
13 }
```

VARIABLES

`s = "{{[]}()}[]"` `c = ']'`

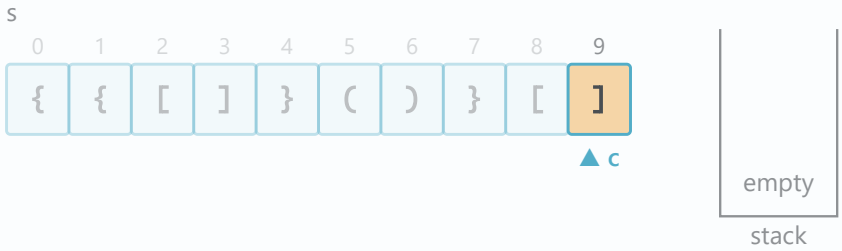
`open = '['` `stack = []`

CALL STACK

`isBalanced("{{[]}()}[]")`

`main()`

Code Trace: `isBalanced("{{[]}()}[]")`



matches('[', ']') → **true**: same type, right order. Continue.

Step 16 / 17

```
1  boolean isBalanced(String s) {
2    Deque<Character> stack = new ArrayDeque<>();
3    for (char c : s.toCharArray()) {
4      if (c == '(' || c == '[' || c == '{') {
5        stack.push(c);
6      } else {
7        if (stack.isEmpty()) return false;
8        char open = stack.pop();
▶       if (!matches(open, c)) return false;
10     }
11   }
12   return stack.isEmpty();
13 }
```

VARIABLES

`s = "{{[]}()}[]"` `c = ']'`

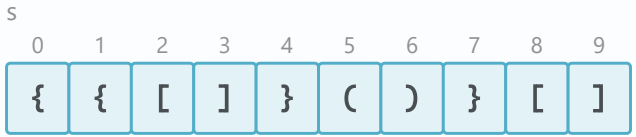
`open = '['` `stack = []`

CALL STACK

`isBalanced("{{[]}()}[]")`

`main()`

Code Trace: `isBalanced("{{[]}()}[]")`





The string is done and the stack is **empty** → **return true**: balanced. Each character was handled once → O(n).

Step 17 / 17

```
1  boolean isBalanced(String s) {
2    Deque<Character> stack = new ArrayDeque<>();
3    for (char c : s.toCharArray()) {
4      if (c == '(' || c == '[' || c == '{') {
5        stack.push(c);
6      } else {
7        if (stack.isEmpty()) return false;
8        char open = stack.pop();
9        if (!matches(open, c)) return false;
10     }
11   }
▶    return stack.isEmpty();
13 }
```

VARIABLES

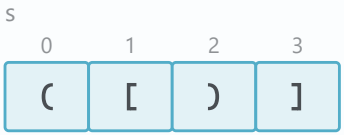
`s = "{{[]}()}[]"` `stack = []`

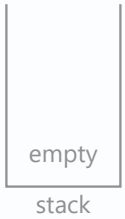
CALL STACK

`isBalanced("{{[]}()}[]")`

`main()`

Code Trace: `isBalanced("([)]")`





Create an empty stack to hold the unmatched opening brackets.

Step 1 / 5

```
1  boolean isBalanced(String s) {
▶    Deque<Character> stack = new ArrayDeque<>();
3    for (char c : s.toCharArray()) {
4      if (c == '(' || c == '[' || c == '{') {
5        stack.push(c);
6      } else {
7        if (stack.isEmpty()) return false;
8        char open = stack.pop();
9        if (!matches(open, c)) return false;
10     }
11   }
12   return stack.isEmpty();
13 }
```

VARIABLES

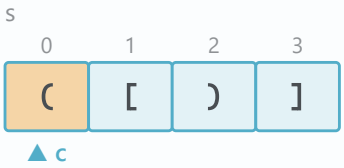
`s = "([)]"` `stack = []`

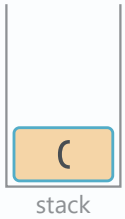
CALL STACK

`isBalanced("([)]")`

`main()`

Code Trace: `isBalanced("([)]")`





'(' is an opening bracket → **push**. It waits for its closer.

Step 2 / 5

```
1  boolean isBalanced(String s) {
2    Deque<Character> stack = new ArrayDeque<>();
3    for (char c : s.toCharArray()) {
▶      if (c == '(' || c == '[' || c == '{') {
▶        stack.push(c);
6      } else {
7        if (stack.isEmpty()) return false;
8        char open = stack.pop();
9        if (!matches(open, c)) return false;
10     }
11   }
12   return stack.isEmpty();
13 }
```

VARIABLES

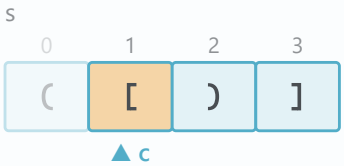
`s = "([)]"` `c = '('` `stack = ['(']`

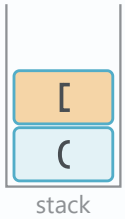
CALL STACK

`isBalanced("([)]")`

`main()`

Code Trace: `isBalanced("([)]")`





'[' is an opening bracket → **push**. It waits for its closer.

Step 3 / 5

```
1  boolean isBalanced(String s) {
2    Deque<Character> stack = new ArrayDeque<>();
3    for (char c : s.toCharArray()) {
▶      if (c == '(' || c == '[' || c == '{') {
▶        stack.push(c);
6      } else {
7        if (stack.isEmpty()) return false;
8        char open = stack.pop();
9        if (!matches(open, c)) return false;
10     }
11   }
12   return stack.isEmpty();
13 }
```

VARIABLES

`s = "([)]"` `c = '['`

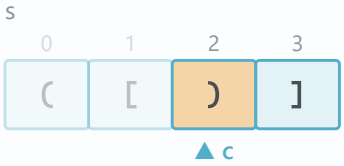
`stack = ['(', '[']`

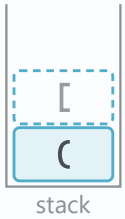
CALL STACK

`isBalanced("([)]")`

`main()`

Code Trace: `isBalanced("([)]")`





')' is a closing bracket; the stack is not empty → **pop** the most recent opener, '['.

Step 4 / 5

```
1  boolean isBalanced(String s) {
2    Deque<Character> stack = new ArrayDeque<>();
3    for (char c : s.toCharArray()) {
4      if (c == '(' || c == '[' || c == '{') {
5        stack.push(c);
6      } else {
▶        if (stack.isEmpty()) return false;
▶        char open = stack.pop();
9        if (!matches(open, c)) return false;
10     }
11   }
12   return stack.isEmpty();
13 }
```

VARIABLES

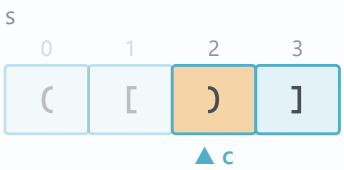
`s = "([)]"` `c = ')'` `open = '['` `stack = ['(']`

CALL STACK

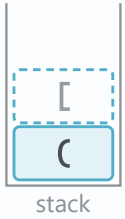
`isBalanced("([)]")`

`main()`

Code Trace: `isBalanced("([)]")`



no match!



matches('[', ')') → **false**: ')' arrives, but the most recent opener is '['. **return false**. Likewise, `(()` fails because the stack is not empty at the end, and `())` because a closer meets an empty stack.

Step 5 / 5

```
1  boolean isBalanced(String s) {
2    Deque<Character> stack = new ArrayDeque<>();
3    for (char c : s.toCharArray()) {
4      if (c == '(' || c == '[' || c == '{') {
5        stack.push(c);
6      } else {
7        if (stack.isEmpty()) return false;
8        char open = stack.pop();
▶        if (!matches(open, c)) return false;
10     }
11   }
12   return stack.isEmpty();
13 }
```

VARIABLES

`s = "([)]"` `c = ')'` `open = '['` `stack = ['(']`

CALL STACK

`isBalanced("([)]")`

`main()`

APPLICATIONS

Next Greater Element

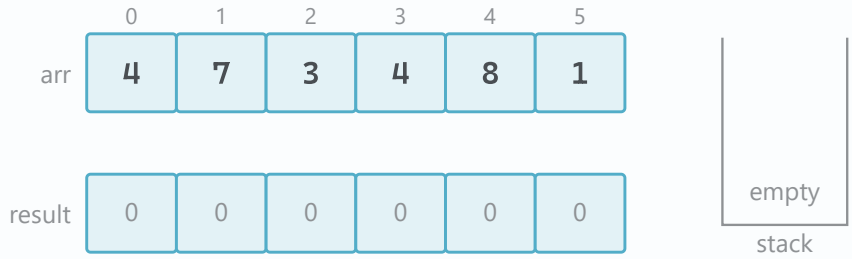
For each item in an array, find the **first larger item to its right**, or −1 if there is none.

arr	4	7	3	4	8	1
result	7	8	4	8	−1	−1

- Brute force: scan to the right of every item → $O(n^2)$.
- With a stack: scan the array **right to left**, keeping only "candidate" larger values → **O(n)**.
- From bottom to top, the stack stays in **decreasing** order (a monotonic stack).

```
int[] nextGreater(int[] arr) {
    int[] result = new int[arr.length];
    Deque<Integer> stack = new ArrayDeque<>();
    for (int i = arr.length-1; i >= 0; i--) {
        while (!stack.isEmpty() &&
               stack.peek() <= arr[i])
            stack.pop();
        result[i] = stack.isEmpty()
                  ? -1 : stack.peek();
        stack.push(arr[i]);
    }
    return result;
}
```

Code Trace: `nextGreater({4, 7, 3, 4, 8, 1})`



Create result (all 0s) and an empty stack. The array is scanned **right to left**.

Step 1 / 17

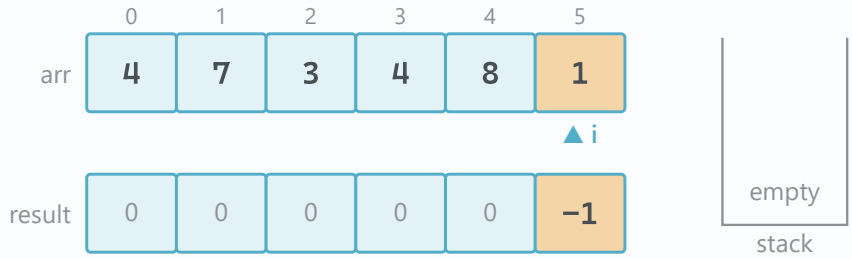
```
1  int[] nextGreater(int[] arr) {
▶      int[] result = new int[arr.length];
▶      Deque<Integer> stack = new ArrayDeque<>();
4      for (int i = arr.length-1; i >= 0; i--) {
5          while (!stack.isEmpty() &&
6                 stack.peek() <= arr[i])
7              stack.pop();
8          result[i] = stack.isEmpty()
9                   ? -1 : stack.peek();
10         stack.push(arr[i]);
11     }
12     return result;
13 }
```

VARIABLES

`result = [0, 0, 0, 0, 0, 0]`

`stack = []`

Code Trace: `nextGreater({4, 7, 3, 4, 8, 1})`



i = 5 (arr[i] = 1): the stack is empty → **−1**.

Step 2 / 17

```
1  int[] nextGreater(int[] arr) {
2      int[] result = new int[arr.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      for (int i = arr.length-1; i >= 0; i--) {
5          while (!stack.isEmpty() &&
6                 stack.peek() <= arr[i])
7              stack.pop();
▶          result[i] = stack.isEmpty()
▶                  ? -1 : stack.peek();
10         stack.push(arr[i]);
11     }
12     return result;
13 }
```

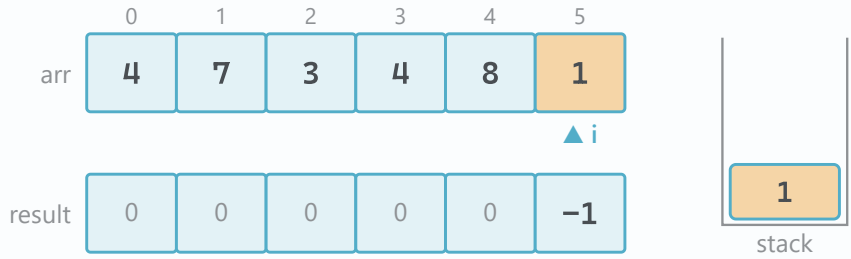
VARIABLES

`i = 5` `arr[i] = 1`

`result = [0, 0, 0, 0, 0, -1]`

`stack = []`

Code Trace: `nextGreater({4, 7, 3, 4, 8, 1})`



push(1): a new candidate for the items to the left. The stack stays decreasing from bottom to top.

Step 3 / 17

```
1  int[] nextGreater(int[] arr) {
2      int[] result = new int[arr.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      for (int i = arr.length-1; i >= 0; i--) {
5          while (!stack.isEmpty() &&
6                 stack.peek() <= arr[i])
7              stack.pop();
8          result[i] = stack.isEmpty()
9                   ? -1 : stack.peek();
▶          stack.push(arr[i]);
11     }
12     return result;
13 }
```

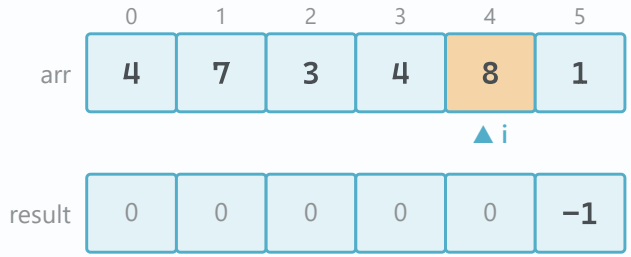
VARIABLES

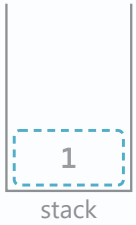
`i = 5` `arr[i] = 1`

`result = [0, 0, 0, 0, 0, -1]`

`stack = [1]`

Code Trace: `nextGreater({4, 7, 3, 4, 8, 1})`





i = 4: the top, 1, is ≤ 8 → **pop**. 1 can never be the answer for anything left of 8 (the larger 8 is in the way).

Step 4 / 17

```
1  int[] nextGreater(int[] arr) {
2      int[] result = new int[arr.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      for (int i = arr.length-1; i >= 0; i--) {
5          while (!stack.isEmpty() &&
6                 stack.peek() <= arr[i])
▶              stack.pop();
8          result[i] = stack.isEmpty()
9                   ? -1 : stack.peek();
10         stack.push(arr[i]);
11     }
12     return result;
13 }
```

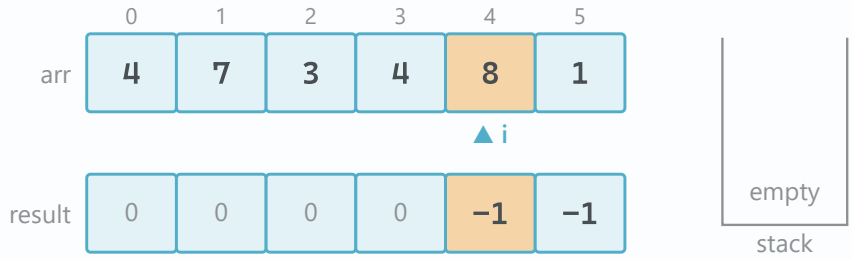
VARIABLES

`i = 4` `arr[i] = 8`

`result = [0, 0, 0, 0, 0, -1]`

`stack = []`

Code Trace: `nextGreater({4, 7, 3, 4, 8, 1})`



i = 4 (arr[i] = 8): the stack is empty → **−1**.

Step 5 / 17

```
1  int[] nextGreater(int[] arr) {
2      int[] result = new int[arr.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      for (int i = arr.length-1; i >= 0; i--) {
5          while (!stack.isEmpty() &&
6                 stack.peek() <= arr[i])
7              stack.pop();
▶          result[i] = stack.isEmpty()
▶                  ? -1 : stack.peek();
10         stack.push(arr[i]);
11     }
12     return result;
13 }
```

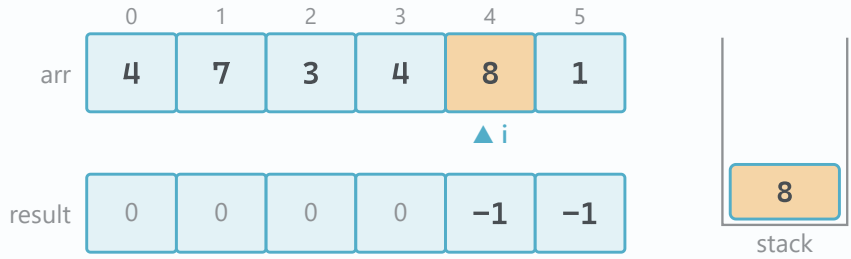
VARIABLES

`i = 4` `arr[i] = 8`

`result = [0, 0, 0, 0, −1, −1]`

`stack = []`

Code Trace: `nextGreater({4, 7, 3, 4, 8, 1})`



push(8): a new candidate for the items to the left. The stack stays decreasing from bottom to top.

Step 6 / 17

```
1  int[] nextGreater(int[] arr) {
2      int[] result = new int[arr.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      for (int i = arr.length-1; i >= 0; i--) {
5          while (!stack.isEmpty() &&
6                 stack.peek() <= arr[i])
7              stack.pop();
8          result[i] = stack.isEmpty()
9                   ? -1 : stack.peek();
▶          stack.push(arr[i]);
11     }
12     return result;
13 }
```

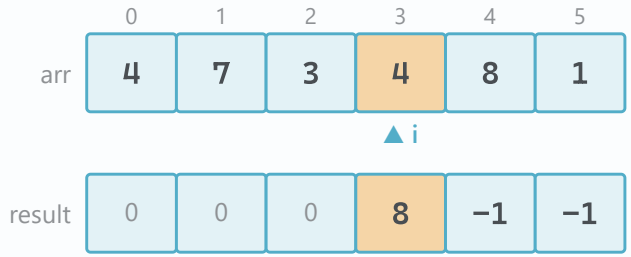
VARIABLES

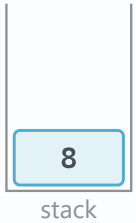
`i = 4` `arr[i] = 8`

`result = [0, 0, 0, 0, −1, −1]`

`stack = [8]`

Code Trace: `nextGreater({4, 7, 3, 4, 8, 1})`





i = 3 (arr[i] = 4): the top, 8, is > 4 → **result[3] = 8**.

Step 7 / 17

```
1  int[] nextGreater(int[] arr) {
2      int[] result = new int[arr.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      for (int i = arr.length-1; i >= 0; i--) {
5          while (!stack.isEmpty() &&
6                 stack.peek() <= arr[i])
7              stack.pop();
▶          result[i] = stack.isEmpty()
▶                  ? -1 : stack.peek();
10         stack.push(arr[i]);
11     }
12     return result;
13 }
```

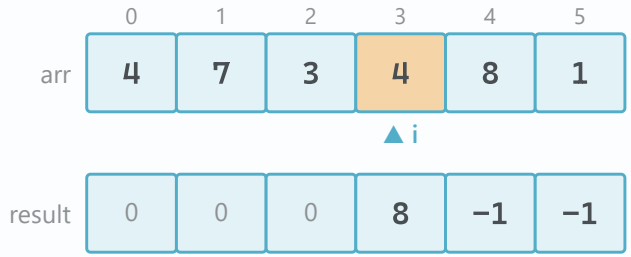
VARIABLES

`i = 3` `arr[i] = 4`

`result = [0, 0, 0, 8, -1, -1]`

`stack = [8]`

Code Trace: `nextGreater({4, 7, 3, 4, 8, 1})`





push(4): a new candidate for the items to the left. The stack stays decreasing from bottom to top.

Step 8 / 17

```
1  int[] nextGreater(int[] arr) {
2      int[] result = new int[arr.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      for (int i = arr.length-1; i >= 0; i--) {
5          while (!stack.isEmpty() &&
6                 stack.peek() <= arr[i])
7              stack.pop();
8          result[i] = stack.isEmpty()
9                   ? -1 : stack.peek();
▶          stack.push(arr[i]);
11     }
12     return result;
13 }
```

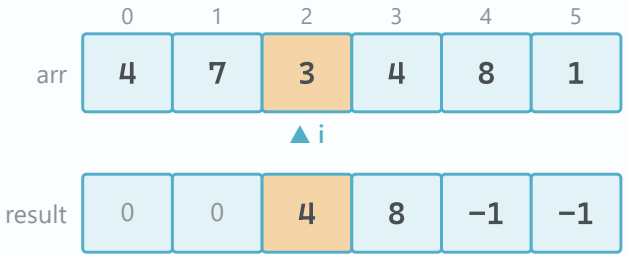
VARIABLES

`i = 3` `arr[i] = 4`

`result = [0, 0, 0, 8, -1, -1]`

`stack = [8, 4]`

Code Trace: `nextGreater({4, 7, 3, 4, 8, 1})`





i = 2 (arr[i] = 3): the top, 4, is > 3 → **result[2] = 4**.

Step 9 / 17

```
1  int[] nextGreater(int[] arr) {
2      int[] result = new int[arr.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      for (int i = arr.length-1; i >= 0; i--) {
5          while (!stack.isEmpty() &&
6                 stack.peek() <= arr[i])
7              stack.pop();
▶          result[i] = stack.isEmpty()
▶                  ? -1 : stack.peek();
10         stack.push(arr[i]);
11     }
12     return result;
13 }
```

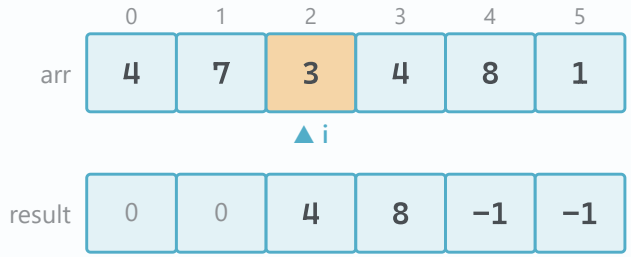
VARIABLES

`i = 2` `arr[i] = 3`

`result = [0, 0, 4, 8, −1, −1]`

`stack = [8, 4]`

Code Trace: `nextGreater({4, 7, 3, 4, 8, 1})`





push(3): a new candidate for the items to the left. The stack stays decreasing from bottom to top.

Step 10 / 17

```
1  int[] nextGreater(int[] arr) {
2      int[] result = new int[arr.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      for (int i = arr.length-1; i >= 0; i--) {
5          while (!stack.isEmpty() &&
6                 stack.peek() <= arr[i])
7              stack.pop();
8          result[i] = stack.isEmpty()
9                   ? -1 : stack.peek();
▶          stack.push(arr[i]);
11     }
12     return result;
13 }
```

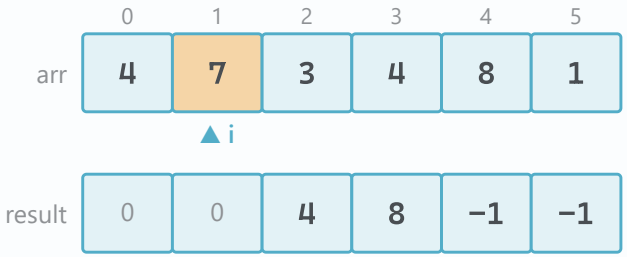
VARIABLES

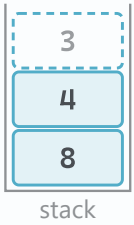
`i = 2` `arr[i] = 3`

`result = [0, 0, 4, 8, −1, −1]`

`stack = [8, 4, 3]`

Code Trace: `nextGreater({4, 7, 3, 4, 8, 1})`





i = 1: the top, 3, is ≤ 7 → **pop**. 3 can never be the answer for anything left of 7 (the larger 7 is in the way).

Step 11 / 17

```
1  int[] nextGreater(int[] arr) {
2      int[] result = new int[arr.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      for (int i = arr.length-1; i >= 0; i--) {
5          while (!stack.isEmpty() &&
6                 stack.peek() <= arr[i])
▶             stack.pop();
8          result[i] = stack.isEmpty()
9                   ? -1 : stack.peek();
10         stack.push(arr[i]);
11     }
12     return result;
13 }
```

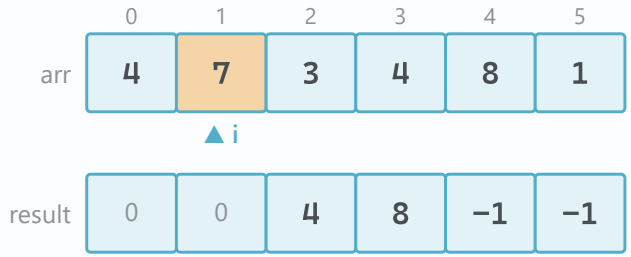
VARIABLES

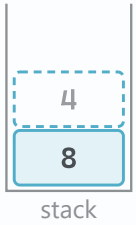
`i = 1` `arr[i] = 7`

`result = [0, 0, 4, 8, −1, −1]`

`stack = [8, 4]`

Code Trace: `nextGreater({4, 7, 3, 4, 8, 1})`





i = 1: the top, 4, is ≤ 7 → **pop**. 4 can never be the answer for anything left of 7 (the larger 7 is in the way).

Step 12 / 17

```
1  int[] nextGreater(int[] arr) {
2      int[] result = new int[arr.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      for (int i = arr.length-1; i >= 0; i--) {
5          while (!stack.isEmpty() &&
6                 stack.peek() <= arr[i])
▶             stack.pop();
8          result[i] = stack.isEmpty()
9                   ? -1 : stack.peek();
10         stack.push(arr[i]);
11     }
12     return result;
13 }
```

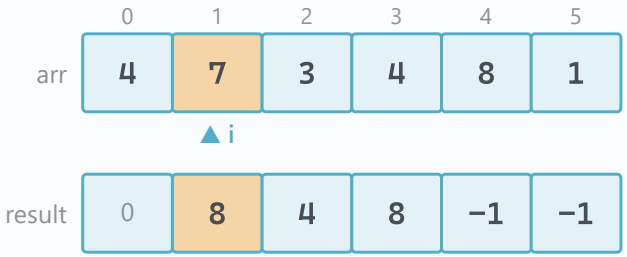
VARIABLES

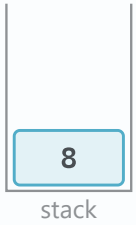
`i = 1` `arr[i] = 7`

`result = [0, 0, 4, 8, −1, −1]`

`stack = [8]`

Code Trace: `nextGreater({4, 7, 3, 4, 8, 1})`





i = 1 (arr[i] = 7): the top, 8, is > 7 → **result[1] = 8**.

Step 13 / 17

```
1  int[] nextGreater(int[] arr) {
2      int[] result = new int[arr.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      for (int i = arr.length-1; i >= 0; i--) {
5          while (!stack.isEmpty() &&
6                 stack.peek() <= arr[i])
7              stack.pop();
▶          result[i] = stack.isEmpty()
▶                  ? -1 : stack.peek();
10         stack.push(arr[i]);
11     }
12     return result;
13 }
```

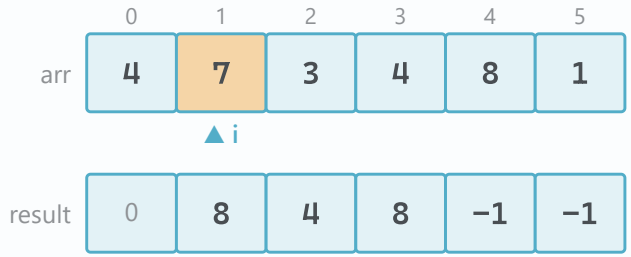
VARIABLES

`i = 1` `arr[i] = 7`

`result = [0, 8, 4, 8, −1, −1]`

`stack = [8]`

Code Trace: `nextGreater({4, 7, 3, 4, 8, 1})`





push(7): a new candidate for the items to the left. The stack stays decreasing from bottom to top.

Step 14 / 17

```
1  int[] nextGreater(int[] arr) {
2      int[] result = new int[arr.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      for (int i = arr.length-1; i >= 0; i--) {
5          while (!stack.isEmpty() &&
6                 stack.peek() <= arr[i])
7              stack.pop();
8          result[i] = stack.isEmpty()
9                   ? -1 : stack.peek();
▶          stack.push(arr[i]);
11     }
12     return result;
13 }
```

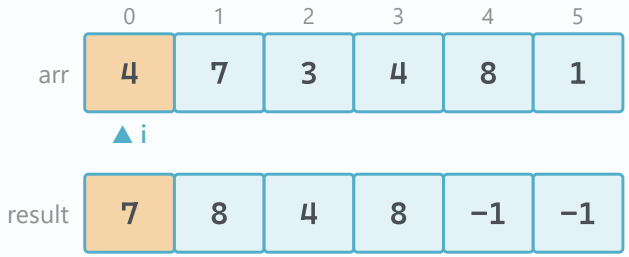
VARIABLES

`i = 1` `arr[i] = 7`

`result = [0, 8, 4, 8, −1, −1]`

`stack = [8, 7]`

Code Trace: `nextGreater({4, 7, 3, 4, 8, 1})`





i = 0 (arr[i] = 4): the top, 7, is > 4 → **result[0] = 7**.

Step 15 / 17

```
1  int[] nextGreater(int[] arr) {
2      int[] result = new int[arr.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      for (int i = arr.length-1; i >= 0; i--) {
5          while (!stack.isEmpty() &&
6                 stack.peek() <= arr[i])
7              stack.pop();
▶          result[i] = stack.isEmpty()
▶                  ? -1 : stack.peek();
10         stack.push(arr[i]);
11     }
12     return result;
13 }
```

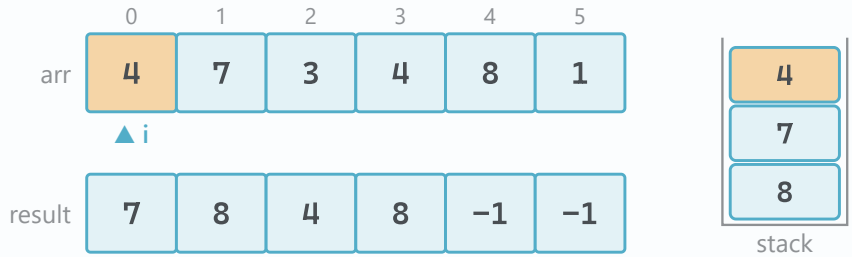
VARIABLES

`i = 0` `arr[i] = 4`

`result = [7, 8, 4, 8, −1, −1]`

`stack = [8, 7]`

Code Trace: `nextGreater({4, 7, 3, 4, 8, 1})`



push(4): a new candidate for the items to the left. The stack stays decreasing from bottom to top.

Step 16 / 17

```
1  int[] nextGreater(int[] arr) {
2      int[] result = new int[arr.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      for (int i = arr.length-1; i >= 0; i--) {
5          while (!stack.isEmpty() &&
6                 stack.peek() <= arr[i])
7              stack.pop();
8          result[i] = stack.isEmpty()
9                   ? -1 : stack.peek();
▶          stack.push(arr[i]);
11     }
12     return result;
13 }
```

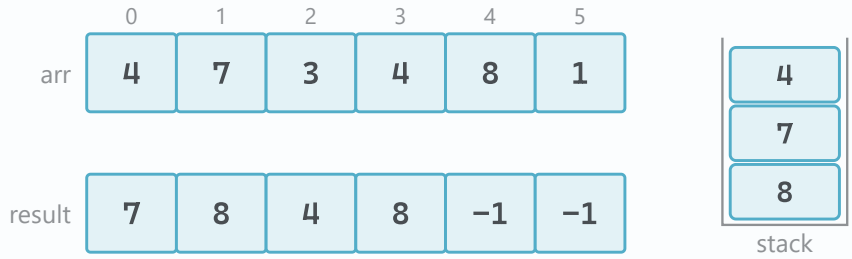
VARIABLES

`i = 0` `arr[i] = 4`

`result = [7, 8, 4, 8, −1, −1]`

`stack = [8, 7, 4]`

Code Trace: `nextGreater({4, 7, 3, 4, 8, 1})`



return [7, 8, 4, 8, -1, -1]. Each item is pushed at most once and popped at most once → **O(n)**.

Step 17 / 17

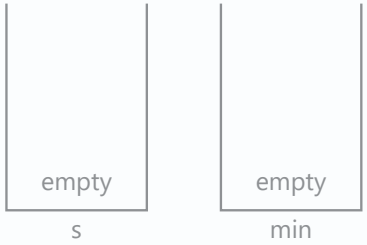
```
1  int[] nextGreater(int[] arr) {
2      int[] result = new int[arr.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      for (int i = arr.length-1; i >= 0; i--) {
5          while (!stack.isEmpty() &&
6                 stack.peek() <= arr[i])
7              stack.pop();
8          result[i] = stack.isEmpty()
9                   ? -1 : stack.peek();
10         stack.push(arr[i]);
11     }
▶      return result;
13 }
```

VARIABLES

`result = [7, 8, 4, 8, −1, −1]`

`stack = [8, 7, 4]`

Code Trace: A Stack with O(1) `getMin()`



`push(5)` `pop()`
`push(3)` `getMin()`
`push(7)` `pop()`
`push(3)` `pop()`
`getMin()` `getMin()`

top of min = current minimum

Two stacks: **s** holds every item, and **min** always has the current minimum on top. Goal: O(1) `getMin()`.

Step 1 / 15

```
▶ Deque<Integer> s = new ArrayDeque<>();
▶ Deque<Integer> min = new ArrayDeque<>();
3  void push(int x) {
4    s.push(x);
5    if (min.isEmpty() || x <= min.peek())
6      min.push(x);
7  }
8  int pop() {
9    int x = s.pop();
10   if (x == min.peek()) min.pop();
11   return x;
12 }
13 int getMin() { return min.peek(); }
```

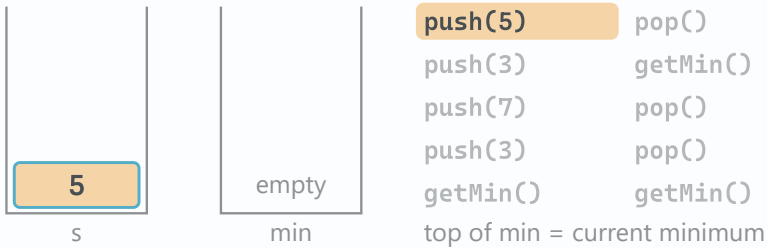
VARIABLES

`s = []` `min = []`

CALL STACK

`main()`

Code Trace: A Stack with O(1) `getMin()`



push(5): `s.push(5)`.

Step 2 / 15

```
1  Deque<Integer> s = new ArrayDeque<>();
2  Deque<Integer> min = new ArrayDeque<>();
3  void push(int x) {
▶    s.push(x);
5    if (min.isEmpty() || x <= min.peek())
6      min.push(x);
7  }
8  int pop() {
9    int x = s.pop();
10   if (x == min.peek()) min.pop();
11   return x;
12 }
13 int getMin() { return min.peek(); }
```

VARIABLES

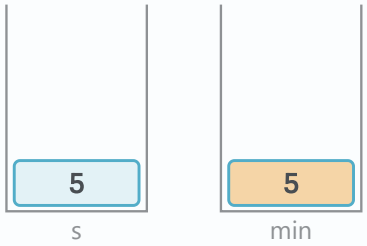
`s = [5]` `min = []` `x = 5`

CALL STACK

`push(5)`

`main()`

Code Trace: A Stack with O(1) `getMin()`



`push(5)` `pop()`
`push(3)` `getMin()`
`push(7)` `pop()`
`push(3)` `pop()`
`getMin()` `getMin()`

top of min = current minimum

min is empty → `min.push(5)`: a new (or equal) minimum.

Step 3 / 15

```
1  Deque<Integer> s = new ArrayDeque<>();
2  Deque<Integer> min = new ArrayDeque<>();
3  void push(int x) {
4    s.push(x);
▶    if (min.isEmpty() || x <= min.peek())
▶      min.push(x);
7  }
8  int pop() {
9    int x = s.pop();
10   if (x == min.peek()) min.pop();
11   return x;
12 }
13 int getMin() { return min.peek(); }
```

VARIABLES

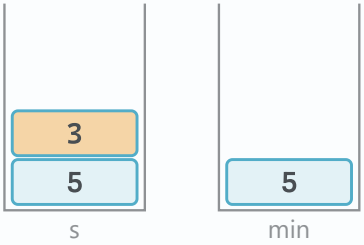
`s = [5]` `min = [5]` `x = 5`

CALL STACK

`push(5)`

`main()`

Code Trace: A Stack with O(1) `getMin()`



`push(5)` `push(3)` `push(7)` `push(3)` `getMin()`

`pop()` `getMin()` `pop()` `pop()` `getMin()`

top of min = current minimum

push(3): `s.push(3)` .

Step 4 / 15

```
1  Deque<Integer> s = new ArrayDeque<>();
2  Deque<Integer> min = new ArrayDeque<>();
3  void push(int x) {
▶    s.push(x);
5    if (min.isEmpty() || x <= min.peek())
6      min.push(x);
7  }
8  int pop() {
9    int x = s.pop();
10   if (x == min.peek()) min.pop();
11   return x;
12 }
13 int getMin() { return min.peek(); }
```

VARIABLES

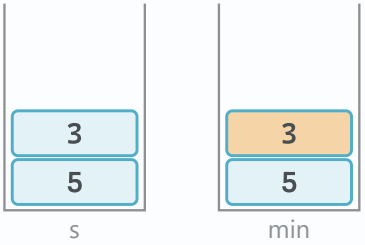
`s = [5, 3]` `min = [5]` `x = 3`

CALL STACK

`push(3)`

`main()`

Code Trace: A Stack with O(1) `getMin()`



`push(5)` `push(3)` `push(7)` `push(3)` `getMin()`

`pop()` `getMin()` `pop()` `pop()` `getMin()`

top of min = current minimum

3 <= 5 → true → `min.push(3)` : a new (or equal) minimum.

Step 5 / 15

```
1  Deque<Integer> s = new ArrayDeque<>();
2  Deque<Integer> min = new ArrayDeque<>();
3  void push(int x) {
4    s.push(x);
▶    if (min.isEmpty() || x <= min.peek())
▶      min.push(x);
7  }
8  int pop() {
9    int x = s.pop();
10   if (x == min.peek()) min.pop();
11   return x;
12 }
13 int getMin() { return min.peek(); }
```

VARIABLES

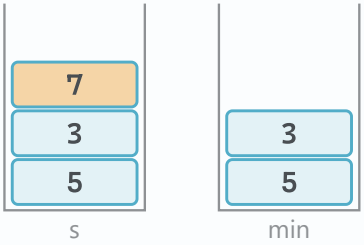
`s = [5, 3]` `min = [5, 3]` `x = 3`

CALL STACK

`push(3)`

`main()`

Code Trace: A Stack with O(1) `getMin()`



`push(5)` `push(3)` `push(7)` `push(3)` `getMin()`

`pop()` `getMin()` `pop()` `pop()` `getMin()`

top of min = current minimum

push(7): `s.push(7)`.

Step 6 / 15

```
1  Deque<Integer> s = new ArrayDeque<>();
2  Deque<Integer> min = new ArrayDeque<>();
3  void push(int x) {
▶    s.push(x);
5    if (min.isEmpty() || x <= min.peek())
6      min.push(x);
7  }
8  int pop() {
9    int x = s.pop();
10   if (x == min.peek()) min.pop();
11   return x;
12 }
13 int getMin() { return min.peek(); }
```

VARIABLES

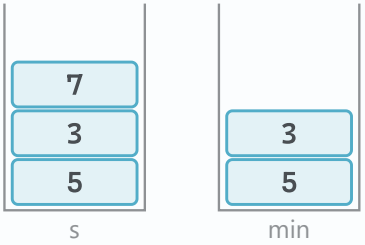
`s = [5, 3, 7]` `min = [5, 3]` `x = 7`

CALL STACK

`push(7)`

`main()`

Code Trace: A Stack with O(1) `getMin()`



`push(5)` `push(3)` `push(7)` `push(3)` `getMin()`

`pop()` `getMin()` `pop()` `pop()` `getMin()`

top of min = current minimum

7 <= 3 → false: 7 is not a new minimum, so min **does not change**.

Step 7 / 15

```
1  Deque<Integer> s = new ArrayDeque<>();
2  Deque<Integer> min = new ArrayDeque<>();
3  void push(int x) {
4    s.push(x);
▶    if (min.isEmpty() || x <= min.peek())
6      min.push(x);
7  }
8  int pop() {
9    int x = s.pop();
10   if (x == min.peek()) min.pop();
11   return x;
12 }
13 int getMin() { return min.peek(); }
```

VARIABLES

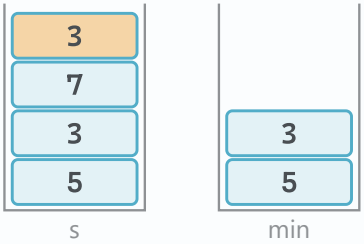
`s = [5, 3, 7]` `min = [5, 3]` `x = 7`

CALL STACK

`push(7)`

`main()`

Code Trace: A Stack with O(1) `getMin()`



push(5)	pop()
push(3)	getMin()
push(7)	pop()
push(3)	pop()
getMin()	getMin()

top of min = current minimum

push(3): `s.push(3)` .

Step 8 / 15

```
1  Deque<Integer> s = new ArrayDeque<>();
2  Deque<Integer> min = new ArrayDeque<>();
3  void push(int x) {
▶    s.push(x);
5    if (min.isEmpty() || x <= min.peek())
6      min.push(x);
7  }
8  int pop() {
9    int x = s.pop();
10   if (x == min.peek()) min.pop();
11   return x;
12 }
13 int getMin() { return min.peek(); }
```

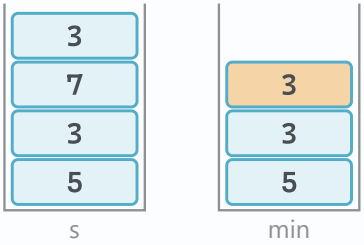
VARIABLES

`s = [5, 3, 7, 3]` `min = [5, 3]` `x = 3`

CALL STACK

- `push(3)`
- `main()`

Code Trace: A Stack with O(1) `getMin()`



push(5)	pop()
push(3)	getMin()
push(7)	pop()
push(3)	pop()
getMin()	getMin()

top of min = current minimum

3 <= 3 → true → `min.push(3)` : a new (or equal) minimum.

Step 9 / 15

```
1  Deque<Integer> s = new ArrayDeque<>();
2  Deque<Integer> min = new ArrayDeque<>();
3  void push(int x) {
4    s.push(x);
▶    if (min.isEmpty() || x <= min.peek())
▶      min.push(x);
7  }
8  int pop() {
9    int x = s.pop();
10   if (x == min.peek()) min.pop();
11   return x;
12 }
13 int getMin() { return min.peek(); }
```

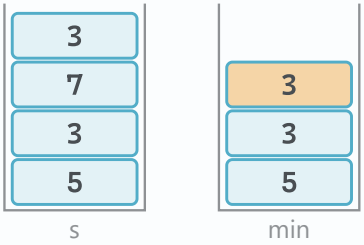
VARIABLES

`s = [5, 3, 7, 3]` `min = [5, 3, 3]` `x = 3`

CALL STACK

- **push(3)**
- main()

Code Trace: A Stack with O(1) `getMin()`



push(5)	pop()
push(3)	getMin()
push(7)	pop()
push(3)	pop()
getMin() → 3	getMin()

top of min = current minimum

getMin(): `min.peek()` = **3**.
No searching → O(1).

Step 10 / 15

```
1  Deque<Integer> s = new ArrayDeque<>();
2  Deque<Integer> min = new ArrayDeque<>();
3  void push(int x) {
4    s.push(x);
5    if (min.isEmpty() || x <= min.peek())
6      min.push(x);
7  }
8  int pop() {
9    int x = s.pop();
10   if (x == min.peek()) min.pop();
11   return x;
12 }
▶  int getMin() { return min.peek(); }
```

VARIABLES

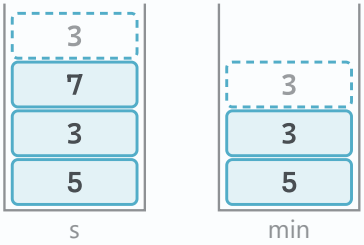
`s = [5, 3, 7, 3]` `min = [5, 3, 3]`

CALL STACK

`getMin()`

`main()`

Code Trace: A Stack with O(1) `getMin()`



push(5)	pop() → 3
push(3)	getMin()
push(7)	pop()
push(3)	pop()
getMin() → 3	getMin()

top of min = current minimum

pop(): **3** comes off s. 3 == min.peek()? **yes** → pop min too; the new minimum is 3.

Step 11 / 15

```
1  Deque<Integer> s = new ArrayDeque<>();
2  Deque<Integer> min = new ArrayDeque<>();
3  void push(int x) {
4    s.push(x);
5    if (min.isEmpty() || x <= min.peek())
6      min.push(x);
7  }
8  int pop() {
▶    int x = s.pop();
▶    if (x == min.peek()) min.pop();
11   return x;
12 }
13 int getMin() { return min.peek(); }
```

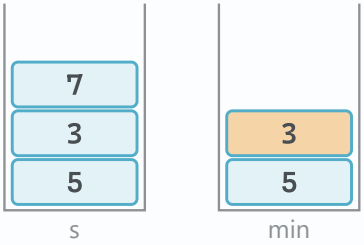
VARIABLES

`s = [5, 3, 7]` `min = [5, 3]` `x = 3`

CALL STACK

- `pop()`
- `main()`

Code Trace: A Stack with O(1) `getMin()`




```
push(5)          pop()      → 3
push(3)          getMin()   → 3
push(7)          pop()
push(3)          pop()
getMin()  → 3    getMin()
```

top of min = current minimum

getMin(): `min.peek()` = **3**.
No searching → O(1).

Step 12 / 15

```
1  Deque<Integer> s = new ArrayDeque<>();
2  Deque<Integer> min = new ArrayDeque<>();
3  void push(int x) {
4    s.push(x);
5    if (min.isEmpty() || x <= min.peek())
6      min.push(x);
7  }
8  int pop() {
9    int x = s.pop();
10   if (x == min.peek()) min.pop();
11   return x;
12 }
▶  int getMin() { return min.peek(); }
```

VARIABLES

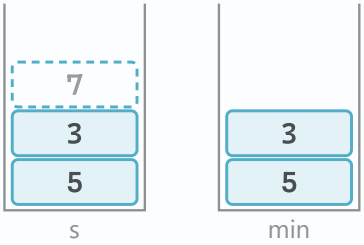
`s = [5, 3, 7]` `min = [5, 3]`

CALL STACK

`getMin()`

`main()`

Code Trace: A Stack with O(1) `getMin()`




```
push(5)          pop()      → 3
push(3)          getMin()   → 3
push(7)          pop()      → 7
push(3)          pop()
getMin()  → 3    getMin()
```

top of min = current minimum

pop(): **7** comes off s. 7 == min.peek()? no → min does not change.


```
1  Deque<Integer> s = new ArrayDeque<>();
2  Deque<Integer> min = new ArrayDeque<>();
3  void push(int x) {
4    s.push(x);
5    if (min.isEmpty() || x <= min.peek())
6      min.push(x);
7  }
8  int pop() {
▶    int x = s.pop();
▶    if (x == min.peek()) min.pop();
11   return x;
12 }
13 int getMin() { return min.peek(); }
```

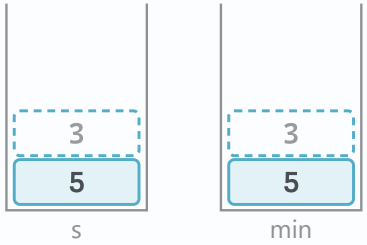
VARIABLES

`s = [5, 3]` `min = [5, 3]` `x = 7`

CALL STACK

- `pop()`
- `main()`

Code Trace: A Stack with O(1) `getMin()`




```
push(5)          pop()      → 3
push(3)          getMin()   → 3
push(7)          pop()      → 7
push(3)          pop()      → 3
getMin()  → 3    getMin()
```

top of min = current minimum

pop(): **3** comes off s. 3 == min.peek()? **yes** → pop min too; the new minimum is 5.

Step 14 / 15

```
1  Deque<Integer> s = new ArrayDeque<>();
2  Deque<Integer> min = new ArrayDeque<>();
3  void push(int x) {
4    s.push(x);
5    if (min.isEmpty() || x <= min.peek())
6      min.push(x);
7  }
8  int pop() {
▶    int x = s.pop();
▶    if (x == min.peek()) min.pop();
11   return x;
12 }
13 int getMin() { return min.peek(); }
```

VARIABLES

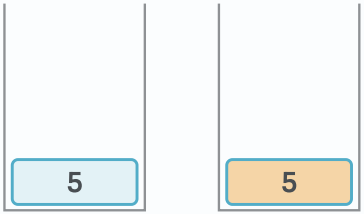
`s = [5]` `min = [5]` `x = 3`

CALL STACK

`pop()`

`main()`

Code Trace: A Stack with O(1) `getMin()`




```
push(5)          pop()     → 3
push(3)          getMin()  → 3
push(7)          pop()     → 7
push(3)          pop()     → 3
getMin()  → 3    getMin()  → 5
```

top of min = current minimum

getMin(): `min.peek()` = **5**. No searching → O(1). Thanks to `<=`, the minimum stayed 3 after the second 3 was popped.

Step 15 / 15

```
1  Deque<Integer> s = new ArrayDeque<>();
2  Deque<Integer> min = new ArrayDeque<>();
3  void push(int x) {
4    s.push(x);
5    if (min.isEmpty() || x <= min.peek())
6      min.push(x);
7  }
8  int pop() {
9    int x = s.pop();
10   if (x == min.peek()) min.pop();
11   return x;
12 }
▶  int getMin() { return min.peek(); }
```

VARIABLES

`s = [5]` `min = [5]`

CALL STACK

`getMin()`

`main()`

Towers of Hanoi: A Non-Recursive Solution with a Stack

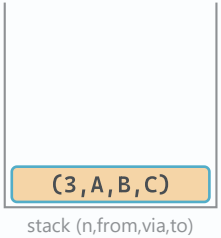
Rules: each move takes the single top disk; a larger disk never goes on a smaller one.

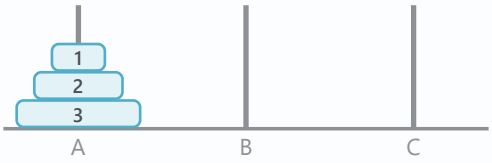
```
// Task: move n disks from 'from' to 'to' using 'via'
record Task(int n, char from, char via, char to) {}

void hanoi(int n) {
  Deque<Task> stack = new ArrayDeque<>();
  stack.push(new Task(n, 'A', 'B', 'C'));
  while (!stack.isEmpty()) {
    Task t = stack.pop();
    if (t.n() == 1)
      System.out.println(t.from() + " → " + t.to());
    else {                          // reverse order
      stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
      stack.push(new Task(1, t.from(), t.via(), t.to()));
      stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
    }
  }
}
```

An **explicit task stack** takes the place of the call stack in the recursive solution. When n > 1, three subtasks are pushed in **reverse order**: **1.** n−1 disks from source to auxiliary, **2.** 1 disk from source to destination, **3.** n−1 disks from auxiliary to destination, so that task 1 ends up on top.

Code Trace: `hanoi(3)` (with a stack)





Step 1 / 11

The initial task is **(3, A, B, C)**: “move 3 disks from A to C, using B”.

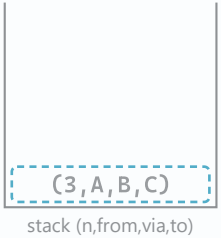
```
1  void hanoi(int n) {
2    Deque<Task> stack = new ArrayDeque<>();
▶    stack.push(new Task(n, 'A', 'B', 'C'));
4    while (!stack.isEmpty()) {
5      Task t = stack.pop();
6      if (t.n() == 1)
7        System.out.println(t.from() + " → " + t.to());
8      else {                          // reverse order
9        stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
10       stack.push(new Task(1, t.from(), t.via(), t.to()));
11       stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
12     }
13   }
14 }
```

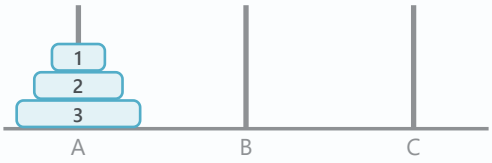
VARIABLES

`stack.size() = 1`

OUTPUT

Code Trace: `hanoi(3)` (with a stack)





pop → t = (3,A,B,C): this is the task to do now.

Step 2 / 11

```
1  void hanoi(int n) {
2    Deque<Task> stack = new ArrayDeque<>();
3    stack.push(new Task(n, 'A', 'B', 'C'));
4    while (!stack.isEmpty()) {
▶      Task t = stack.pop();
6      if (t.n() == 1)
7        System.out.println(t.from() + " → " + t.to());
8      else {                          // reverse order
9        stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
10       stack.push(new Task(1, t.from(), t.via(), t.to()));
11       stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
12     }
13   }
14 }
```

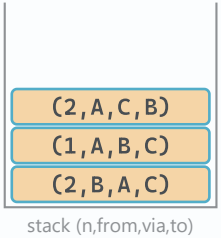
VARIABLES

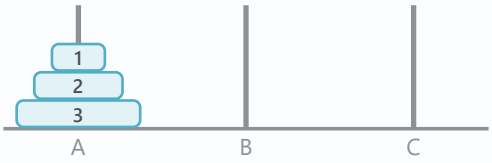
`t = (3,A,B,C)`

`stack.size() = 0`

OUTPUT

Code Trace: `hanoi(3)` (with a stack)





n = 3 > 1 → push three subtasks in **reverse order**; (2, A, C, B), which runs first, ends up on top (LIFO).

Step 3 / 11

```
1  void hanoi(int n) {
2    Deque<Task> stack = new ArrayDeque<>();
3    stack.push(new Task(n, 'A', 'B', 'C'));
4    while (!stack.isEmpty()) {
5      Task t = stack.pop();
6      if (t.n() == 1)
7        System.out.println(t.from() + " → " + t.to());
8      else {                              // reverse order
▶        stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
▶        stack.push(new Task(1, t.from(), t.via(), t.to()));
▶        stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
12     }
13   }
14 }
```

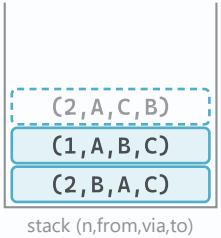
VARIABLES

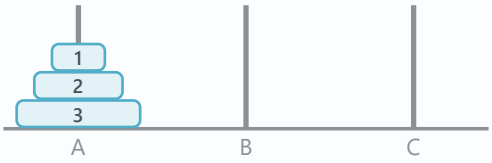
`t = (3,A,B,C)`

`stack.size() = 3`

OUTPUT

Code Trace: `hanoi(3)` (with a stack)





pop → t = (2,A,C,B): this is the task to do now.

Step 4 / 11

```
1  void hanoi(int n) {
2    Deque<Task> stack = new ArrayDeque<>();
3    stack.push(new Task(n, 'A', 'B', 'C'));
4    while (!stack.isEmpty()) {
▶      Task t = stack.pop();
6      if (t.n() == 1)
7        System.out.println(t.from() + " → " + t.to());
8      else {                          // reverse order
9        stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
10       stack.push(new Task(1, t.from(), t.via(), t.to()));
11       stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
12     }
13   }
14 }
```

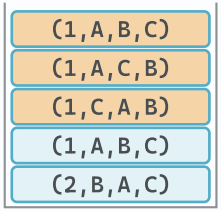
VARIABLES

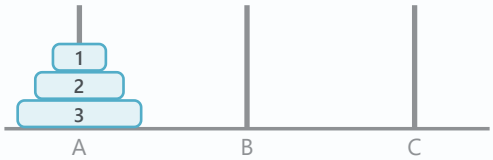
`t = (2,A,C,B)`

`stack.size() = 2`

OUTPUT

Code Trace: `hanoi(3)` (with a stack)





Step 5 / 11

n = 2 > 1 → push three subtasks in **reverse order**; (1, A, B, C), which runs first, ends up on top (LIFO).

```
1  void hanoi(int n) {
2    Deque<Task> stack = new ArrayDeque<>();
3    stack.push(new Task(n, 'A', 'B', 'C'));
4    while (!stack.isEmpty()) {
5      Task t = stack.pop();
6      if (t.n() == 1)
7        System.out.println(t.from() + " → " + t.to());
8      else {                              // reverse order
▶        stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
▶        stack.push(new Task(1, t.from(), t.via(), t.to()));
▶        stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
12     }
13   }
14 }
```

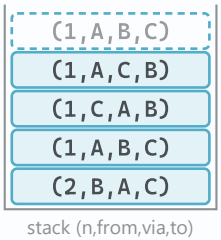
VARIABLES

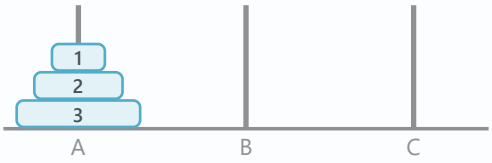
`t = (2,A,C,B)`

`stack.size() = 5`

OUTPUT

Code Trace: `hanoi(3)` (with a stack)





pop → t = (1,A,B,C): this is the task to do now.

Step 6 / 11

```
1  void hanoi(int n) {
2    Deque<Task> stack = new ArrayDeque<>();
3    stack.push(new Task(n, 'A', 'B', 'C'));
4    while (!stack.isEmpty()) {
▶      Task t = stack.pop();
6      if (t.n() == 1)
7        System.out.println(t.from() + " → " + t.to());
8      else {                              // reverse order
9        stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
10       stack.push(new Task(1, t.from(), t.via(), t.to()));
11       stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
12     }
13   }
14 }
```

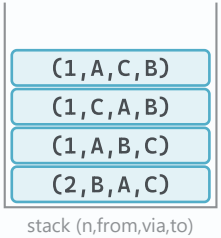
VARIABLES

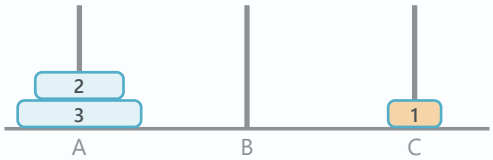
`t = (1,A,B,C)`

`stack.size() = 4`

OUTPUT

Code Trace: `hanoi(3)` (with a stack)





n = 1 → move directly: disk 1, **A** → **C** (move 1).

Step 7 / 11

```
1  void hanoi(int n) {
2    Deque<Task> stack = new ArrayDeque<>();
3    stack.push(new Task(n, 'A', 'B', 'C'));
4    while (!stack.isEmpty()) {
5      Task t = stack.pop();
6      if (t.n() == 1)
▶        System.out.println(t.from() + " → " + t.to());
8      else {                          // reverse order
9        stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
10       stack.push(new Task(1, t.from(), t.via(), t.to()));
11       stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
12     }
13   }
14 }
```

VARIABLES

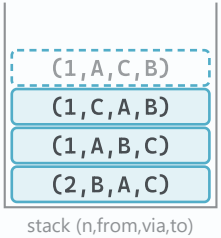
`t = (1,A,B,C)`

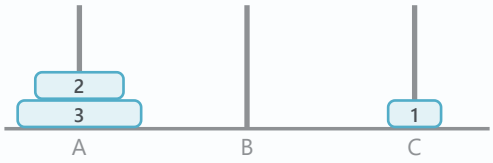
`stack.size() = 4`

OUTPUT

```
A → C
```

Code Trace: `hanoi(3)` (with a stack)





pop → t = (1,A,C,B): this is the task to do now.

Step 8 / 11

```
1  void hanoi(int n) {
2    Deque<Task> stack = new ArrayDeque<>();
3    stack.push(new Task(n, 'A', 'B', 'C'));
4    while (!stack.isEmpty()) {
▶      Task t = stack.pop();
6      if (t.n() == 1)
7        System.out.println(t.from() + " → " + t.to());
8      else {                          // reverse order
9        stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
10       stack.push(new Task(1, t.from(), t.via(), t.to()));
11       stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
12     }
13   }
14 }
```

VARIABLES

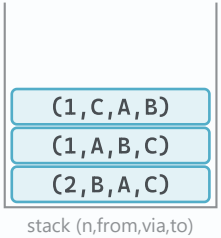
`t = (1,A,C,B)`

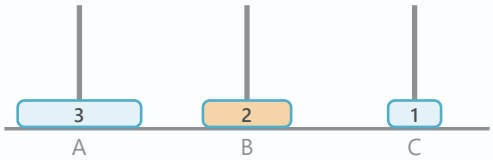
`stack.size() = 3`

OUTPUT

```
A → C
```

Code Trace: `hanoi(3)` (with a stack)





n = 1 → move directly: disk 2, **A** → **B** (move 2).

Step 9 / 11

```
1  void hanoi(int n) {
2    Deque<Task> stack = new ArrayDeque<>();
3    stack.push(new Task(n, 'A', 'B', 'C'));
4    while (!stack.isEmpty()) {
5      Task t = stack.pop();
6      if (t.n() == 1)
▶        System.out.println(t.from() + " → " + t.to());
8      else {                          // reverse order
9        stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
10       stack.push(new Task(1, t.from(), t.via(), t.to()));
11       stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
12     }
13   }
14 }
```

VARIABLES

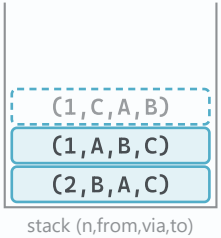
`t = (1,A,C,B)`

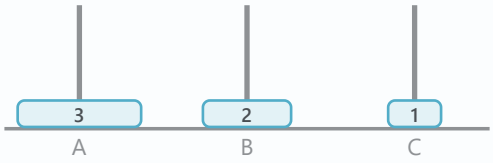
`stack.size() = 3`

OUTPUT

```
A → C
A → B
```

Code Trace: `hanoi(3)` (with a stack)





pop → t = (1,C,A,B): this is the task to do now.

Step 10 / 11

```
1  void hanoi(int n) {
2    Deque<Task> stack = new ArrayDeque<>();
3    stack.push(new Task(n, 'A', 'B', 'C'));
4    while (!stack.isEmpty()) {
▶      Task t = stack.pop();
6      if (t.n() == 1)
7        System.out.println(t.from() + " → " + t.to());
8      else {                          // reverse order
9        stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
10       stack.push(new Task(1, t.from(), t.via(), t.to()));
11       stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
12     }
13   }
14 }
```

VARIABLES

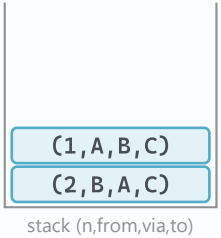
`t = (1,C,A,B)`

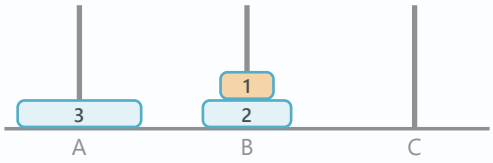
`stack.size() = 2`

OUTPUT

```
A → C
A → B
```

Code Trace: `hanoi(3)` (with a stack)





n = 1 → move directly: disk 1, **C** → **B** (move 3).

Step 11 / 11

```
1  void hanoi(int n) {
2    Deque<Task> stack = new ArrayDeque<>();
3    stack.push(new Task(n, 'A', 'B', 'C'));
4    while (!stack.isEmpty()) {
5      Task t = stack.pop();
6      if (t.n() == 1)
▶        System.out.println(t.from() + " → " + t.to());
8      else {                          // reverse order
9        stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
10       stack.push(new Task(1, t.from(), t.via(), t.to()));
11       stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
12     }
13   }
14 }
```

VARIABLES

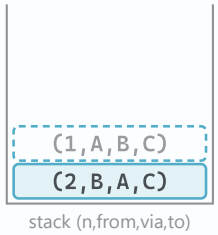
`t = (1,C,A,B)`

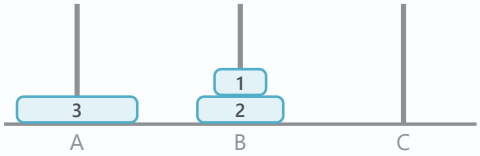
`stack.size() = 2`

OUTPUT

```
A → C
A → B
C → B
```

Code Trace: `hanoi(3)` (with a stack, continued)





pop → t = (1,A,B,C): this is the task to do now.

Step 1 / 11

```
1  void hanoi(int n) {
2    Deque<Task> stack = new ArrayDeque<>();
3    stack.push(new Task(n, 'A', 'B', 'C'));
4    while (!stack.isEmpty()) {
▶      Task t = stack.pop();
6      if (t.n() == 1)
7        System.out.println(t.from() + " → " + t.to());
8      else {                          // reverse order
9        stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
10       stack.push(new Task(1, t.from(), t.via(), t.to()));
11       stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
12     }
13   }
14 }
```

VARIABLES

`t = (1,A,B,C)`

`stack.size() = 1`

OUTPUT

```
A → C
A → B
C → B
```

Code Trace: `hanoi(3)` (with a stack, continued)



Step 2 / 11

n = 1 → move directly: disk 3, **A** → **C** (move 4).

```
1  void hanoi(int n) {
2    Deque<Task> stack = new ArrayDeque<>();
3    stack.push(new Task(n, 'A', 'B', 'C'));
4    while (!stack.isEmpty()) {
5      Task t = stack.pop();
6      if (t.n() == 1)
▶        System.out.println(t.from() + " → " + t.to());
8      else {                          // reverse order
9        stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
10       stack.push(new Task(1, t.from(), t.via(), t.to()));
11       stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
12     }
13   }
14 }
```

VARIABLES

`t = (1,A,B,C)`

`stack.size() = 1`

OUTPUT

```
A → C
A → B
C → B
A → C
```

Code Trace: `hanoi(3)` (with a stack, continued)



Step 3 / 11

pop → t = (2,B,A,C): this is the task to do now.

```
1  void hanoi(int n) {
2    Deque<Task> stack = new ArrayDeque<>();
3    stack.push(new Task(n, 'A', 'B', 'C'));
4    while (!stack.isEmpty()) {
▶      Task t = stack.pop();
6      if (t.n() == 1)
7        System.out.println(t.from() + " → " + t.to());
8      else {                          // reverse order
9        stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
10       stack.push(new Task(1, t.from(), t.via(), t.to()));
11       stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
12     }
13   }
14 }
```

VARIABLES

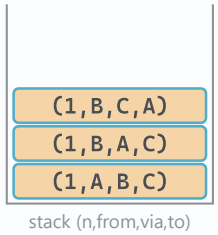
`t = (2,B,A,C)`

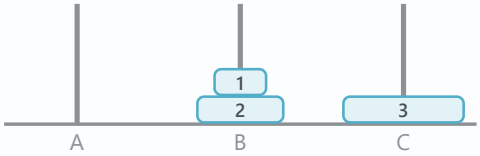
`stack.size() = 0`

OUTPUT

```
A → C
A → B
C → B
A → C
```

Code Trace: `hanoi(3)` (with a stack, continued)





n = 2 > 1 → push three subtasks in **reverse order**; (1, B, C, A), which runs first, ends up on top (LIFO).

Step 4 / 11

```
1  void hanoi(int n) {
2    Deque<Task> stack = new ArrayDeque<>();
3    stack.push(new Task(n, 'A', 'B', 'C'));
4    while (!stack.isEmpty()) {
5      Task t = stack.pop();
6      if (t.n() == 1)
7        System.out.println(t.from() + " → " + t.to());
8      else {                          // reverse order
▶        stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
▶        stack.push(new Task(1, t.from(), t.via(), t.to()));
▶        stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
12     }
13   }
14 }
```

VARIABLES

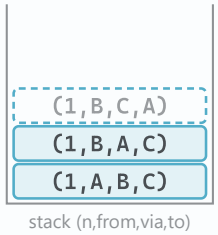
`t = (2,B,A,C)`

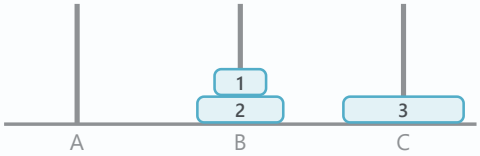
`stack.size() = 3`

OUTPUT

```
A → C
A → B
C → B
A → C
```

Code Trace: `hanoi(3)` (with a stack, continued)





pop → t = (1,B,C,A): this is the task to do now.

Step 5 / 11

```
1  void hanoi(int n) {
2    Deque<Task> stack = new ArrayDeque<>();
3    stack.push(new Task(n, 'A', 'B', 'C'));
4    while (!stack.isEmpty()) {
▶      Task t = stack.pop();
6      if (t.n() == 1)
7        System.out.println(t.from() + " → " + t.to());
8      else {                          // reverse order
9        stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
10       stack.push(new Task(1, t.from(), t.via(), t.to()));
11       stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
12     }
13   }
14 }
```

VARIABLES

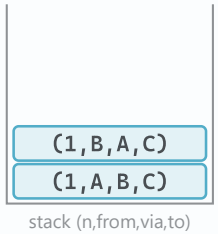
`t = (1,B,C,A)`

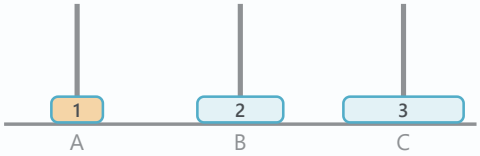
`stack.size() = 2`

OUTPUT

```
A → C
A → B
C → B
A → C
```

Code Trace: `hanoi(3)` (with a stack, continued)





n = 1 → move directly: disk 1, **B** → **A** (move 5).

Step 6 / 11

```
1  void hanoi(int n) {
2    Deque<Task> stack = new ArrayDeque<>();
3    stack.push(new Task(n, 'A', 'B', 'C'));
4    while (!stack.isEmpty()) {
5      Task t = stack.pop();
6      if (t.n() == 1)
▶        System.out.println(t.from() + " → " + t.to());
8      else {                          // reverse order
9        stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
10       stack.push(new Task(1, t.from(), t.via(), t.to()));
11       stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
12     }
13   }
14 }
```

VARIABLES

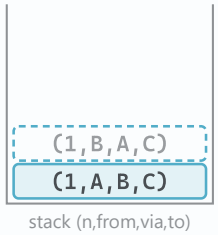
`t = (1,B,C,A)`

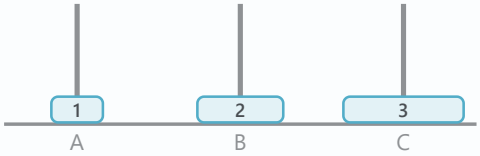
`stack.size() = 2`

OUTPUT

```
A → C
A → B
C → B
A → C
B → A
```

Code Trace: `hanoi(3)` (with a stack, continued)





pop → t = (1,B,A,C): this is the task to do now.

Step 7 / 11

```
1  void hanoi(int n) {
2    Deque<Task> stack = new ArrayDeque<>();
3    stack.push(new Task(n, 'A', 'B', 'C'));
4    while (!stack.isEmpty()) {
▶      Task t = stack.pop();
6      if (t.n() == 1)
7        System.out.println(t.from() + " → " + t.to());
8      else {                          // reverse order
9        stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
10       stack.push(new Task(1, t.from(), t.via(), t.to()));
11       stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
12     }
13   }
14 }
```

VARIABLES

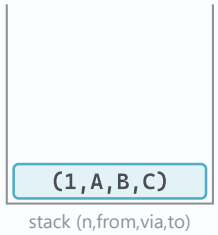
`t = (1,B,A,C)`

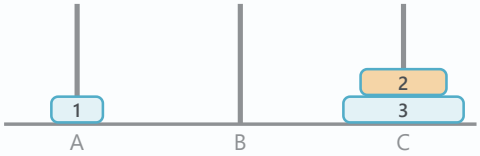
`stack.size() = 1`

OUTPUT

```
A → C
A → B
C → B
A → C
B → A
```

Code Trace: `hanoi(3)` (with a stack, continued)





n = 1 → move directly: disk 2, **B** → **C** (move 6).

Step 8 / 11

```
1  void hanoi(int n) {
2    Deque<Task> stack = new ArrayDeque<>();
3    stack.push(new Task(n, 'A', 'B', 'C'));
4    while (!stack.isEmpty()) {
5      Task t = stack.pop();
6      if (t.n() == 1)
▶        System.out.println(t.from() + " → " + t.to());
8      else {                          // reverse order
9        stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
10       stack.push(new Task(1, t.from(), t.via(), t.to()));
11       stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
12     }
13   }
14 }
```

VARIABLES

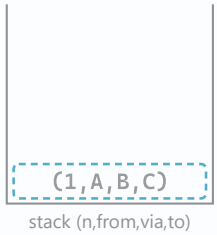
`t = (1,B,A,C)`

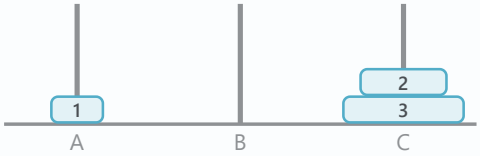
`stack.size() = 1`

OUTPUT

```
A → C
A → B
C → B
A → C
B → A
B → C
```

Code Trace: `hanoi(3)` (with a stack, continued)





pop → t = (1,A,B,C): this is the task to do now.

Step 9 / 11

```
1  void hanoi(int n) {
2    Deque<Task> stack = new ArrayDeque<>();
3    stack.push(new Task(n, 'A', 'B', 'C'));
4    while (!stack.isEmpty()) {
▶      Task t = stack.pop();
6      if (t.n() == 1)
7        System.out.println(t.from() + " → " + t.to());
8      else {                          // reverse order
9        stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
10       stack.push(new Task(1, t.from(), t.via(), t.to()));
11       stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
12     }
13   }
14 }
```

VARIABLES

`t = (1,A,B,C)`

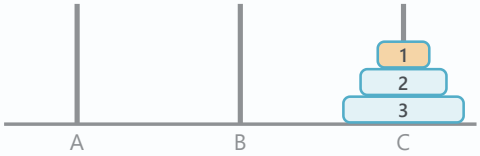
`stack.size() = 0`

OUTPUT

```
A → C
A → B
C → B
A → C
B → A
B → C
```

Code Trace: `hanoi(3)` (with a stack, continued)





n = 1 → move directly: disk 1, **A** → **C** (move 7).

Step 10 / 11

```
1  void hanoi(int n) {
2    Deque<Task> stack = new ArrayDeque<>();
3    stack.push(new Task(n, 'A', 'B', 'C'));
4    while (!stack.isEmpty()) {
5      Task t = stack.pop();
6      if (t.n() == 1)
▶        System.out.println(t.from() + " → " + t.to());
8      else {                          // reverse order
9        stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
10       stack.push(new Task(1, t.from(), t.via(), t.to()));
11       stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
12     }
13   }
14 }
```

VARIABLES

`t = (1,A,B,C)`

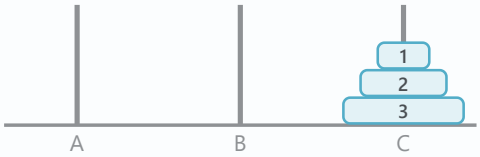
`stack.size() = 0`

OUTPUT

```
A → C
A → B
C → B
A → C
B → A
B → C
A → C
```

Code Trace: `hanoi(3)` (with a stack, continued)





The stack is empty → the loop ends. **7 = 2^3 − 1** moves; every disk is on C.

Step 11 / 11

```
1  void hanoi(int n) {
2    Deque<Task> stack = new ArrayDeque<>();
3    stack.push(new Task(n, 'A', 'B', 'C'));
▶    while (!stack.isEmpty()) {
5      Task t = stack.pop();
6      if (t.n() == 1)
7        System.out.println(t.from() + " → " + t.to());
8      else {                          // reverse order
9        stack.push(new Task(t.n()-1, t.via(), t.from(), t.to()));
10       stack.push(new Task(1, t.from(), t.via(), t.to()));
11       stack.push(new Task(t.n()-1, t.from(), t.to(), t.via()));
12     }
13   }
14 }
```

VARIABLES

`stack.size() = 0`

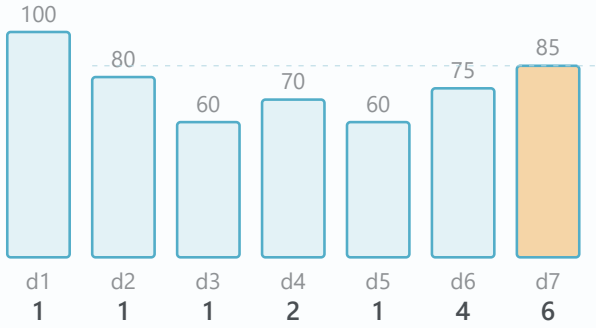
OUTPUT

```
A → C
A → B
C → B
A → C
B → A
B → C
A → C
```

STOCK SPAN

The Stock Span Problem

- An array holds daily closing prices.
- The **span** of a day is the number of consecutive days, counting back from and including that day, whose price is **less than or equal to** that day's price.
- In financial analysis it shows “for how many days” a price has not been beaten.



Day	Price	Span	Why?
1	100	1	first day
4	70	2	60 ≤ 70, 80 > 70
6	75	4	60, 70, 60 ≤ 75; 80 > 75
7	85	6	75, 60, 70, 60, 80 ≤ 85

Prices [100, 80, 60, 70, 60, 75, 85] → spans [1, 1, 1, 2, 1, 4, 6]. The bold numbers under the bars are the spans.

STOCK SPAN

Stock Span: An O(n) Solution with a Stack

```
int[] span(int[] prices) {
    int[] spans = new int[prices.length];
    Deque<Integer> stack = new ArrayDeque<>();
    stack.push(0); spans[0] = 1;
    for (int i = 1; i < prices.length; i++) {
        while (!stack.isEmpty() &&
            prices[i] >= prices[stack.peek()])
            stack.pop();
        spans[i] = stack.isEmpty() ? i + 1
                                   : i - stack.peek();
        stack.push(i);
    }
    return spans;
}
```

What is on the stack?

The **indices** of days not yet beaten (prices decrease from bottom to top).

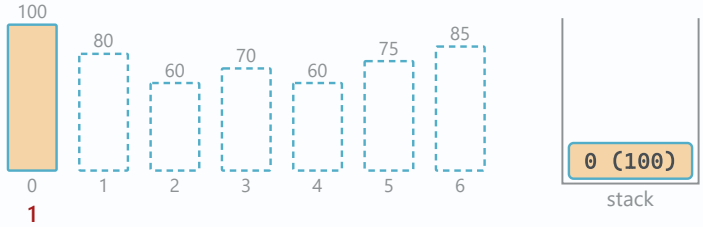
Span

The top is the nearest earlier day with a **higher** price: span = i − top, or i + 1 if the stack is empty.

Complexity

Each index is pushed once and popped at most once → **O(n)**.

Code Trace: `span({100, 80, 60, 70, 60, 75, 85})`



First day: push index 0; **spans[0] = 1** (no earlier days).

Step 1 / 19

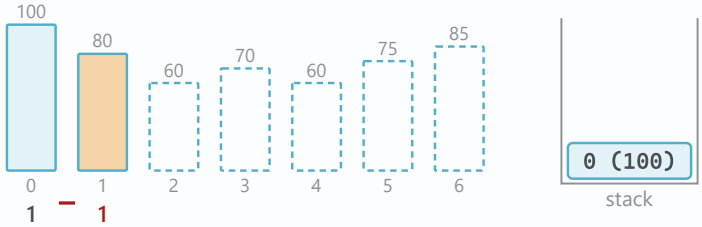
```
1  int[] span(int[] prices) {
2      int[] spans = new int[prices.length];
3      Deque<Integer> stack = new ArrayDeque<>();
▶      stack.push(0); spans[0] = 1;
5      for (int i = 1; i < prices.length; i++) {
6          while (!stack.isEmpty() &&
7              prices[i] >= prices[stack.peek()])
8              stack.pop();
9          spans[i] = stack.isEmpty() ? i + 1
10                                    : i - stack.peek();
11         stack.push(i);
12     }
13     return spans;
14 }
```

VARIABLES

`spans = [1, 0, 0, 0, 0, 0, 0]`

`stack = [0]`

Code Trace: `span({100, 80, 60, 70, 60, 75, 85})`



i = 1 (price 80): the top, day 0 (100), is higher → span = 1 − 0 = **1**.

Step 2 / 19

```
1  int[] span(int[] prices) {
2      int[] spans = new int[prices.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      stack.push(0); spans[0] = 1;
5      for (int i = 1; i < prices.length; i++) {
6          while (!stack.isEmpty() &&
7              prices[i] >= prices[stack.peek()])
8              stack.pop();
▶          spans[i] = stack.isEmpty() ? i + 1
▶                                     : i - stack.peek();
11         stack.push(i);
12     }
13     return spans;
14 }
```

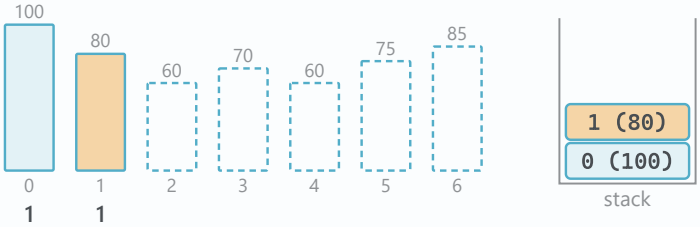
VARIABLES

`i = 1` `prices[i] = 80`

`spans = [1, 1, 0, 0, 0, 0, 0]`

`stack = [0]`

Code Trace: `span({100, 80, 60, 70, 60, 75, 85})`



push(1): a possible boundary for later days.

Step 3 / 19

```
1  int[] span(int[] prices) {
2      int[] spans = new int[prices.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      stack.push(0); spans[0] = 1;
5      for (int i = 1; i < prices.length; i++) {
6          while (!stack.isEmpty() &&
7              prices[i] >= prices[stack.peek()])
8              stack.pop();
9          spans[i] = stack.isEmpty() ? i + 1
10                                    : i - stack.peek();
▶          stack.push(i);
12     }
13     return spans;
14 }
```

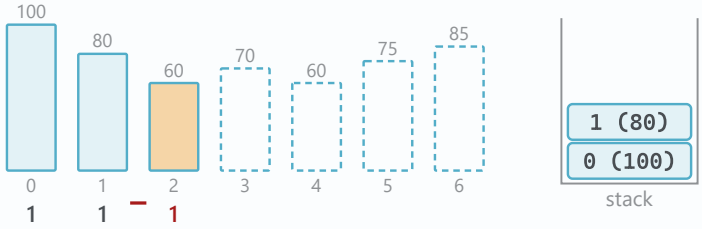
VARIABLES

`i = 1` `prices[i] = 80`

`spans = [1, 1, 0, 0, 0, 0, 0]`

`stack = [0, 1]`

Code Trace: `span({100, 80, 60, 70, 60, 75, 85})`



i = 2 (price 60): the top, day 1 (80), is higher → span = 2 − 1 = **1**.

Step 4 / 19

```
1  int[] span(int[] prices) {
2      int[] spans = new int[prices.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      stack.push(0); spans[0] = 1;
5      for (int i = 1; i < prices.length; i++) {
6          while (!stack.isEmpty() &&
7              prices[i] >= prices[stack.peek()])
8              stack.pop();
▶          spans[i] = stack.isEmpty() ? i + 1
▶                                     : i - stack.peek();
11         stack.push(i);
12     }
13     return spans;
14 }
```

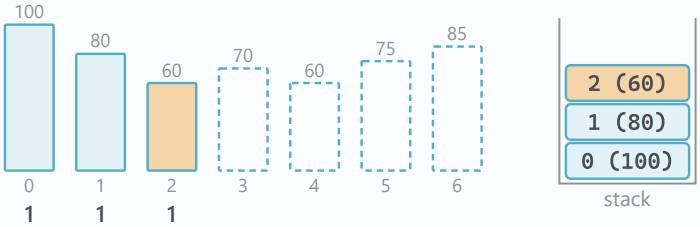
VARIABLES

`i = 2` `prices[i] = 60`

`spans = [1, 1, 1, 0, 0, 0, 0]`

`stack = [0, 1]`

Code Trace: `span({100, 80, 60, 70, 60, 75, 85})`



push(2): a possible boundary for later days.

Step 5 / 19

```
1  int[] span(int[] prices) {
2      int[] spans = new int[prices.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      stack.push(0); spans[0] = 1;
5      for (int i = 1; i < prices.length; i++) {
6          while (!stack.isEmpty() &&
7              prices[i] >= prices[stack.peek()])
8              stack.pop();
9          spans[i] = stack.isEmpty() ? i + 1
10                                    : i - stack.peek();
▶          stack.push(i);
12     }
13     return spans;
14 }
```

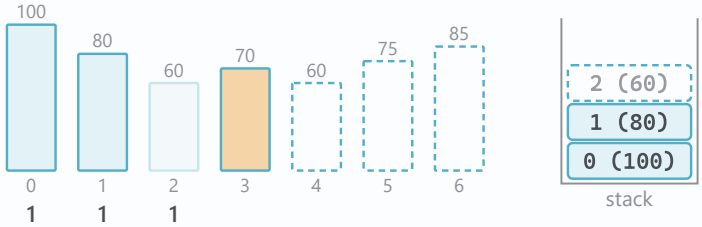
VARIABLES

`i = 2` `prices[i] = 60`

`spans = [1, 1, 1, 0, 0, 0, 0]`

`stack = [0, 1, 2]`

Code Trace: `span({100, 80, 60, 70, 60, 75, 85})`



i = 3: 70 ≥ prices[2] = 60 → **pop**. Day 2 can no longer be anyone's boundary.

Step 6 / 19

```
1  int[] span(int[] prices) {
2      int[] spans = new int[prices.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      stack.push(0); spans[0] = 1;
5      for (int i = 1; i < prices.length; i++) {
6          while (!stack.isEmpty() &&
7              prices[i] >= prices[stack.peek()])
▶             stack.pop();
9          spans[i] = stack.isEmpty() ? i + 1
10                                    : i - stack.peek();
11         stack.push(i);
12     }
13     return spans;
14 }
```

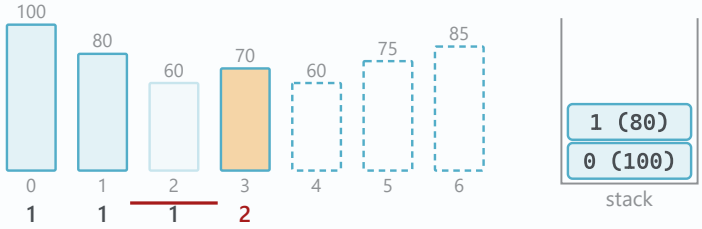
VARIABLES

`i = 3` `prices[i] = 70`

`spans = [1, 1, 1, 0, 0, 0, 0]`

`stack = [0, 1]`

Code Trace: `span({100, 80, 60, 70, 60, 75, 85})`



i = 3 (price 70): the top, day 1 (80), is higher → span = 3 − 1 = **2**.

Step 7 / 19

```
1  int[] span(int[] prices) {
2      int[] spans = new int[prices.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      stack.push(0); spans[0] = 1;
5      for (int i = 1; i < prices.length; i++) {
6          while (!stack.isEmpty() &&
7              prices[i] >= prices[stack.peek()])
8              stack.pop();
▶          spans[i] = stack.isEmpty() ? i + 1
▶                                     : i - stack.peek();
11         stack.push(i);
12     }
13     return spans;
14 }
```

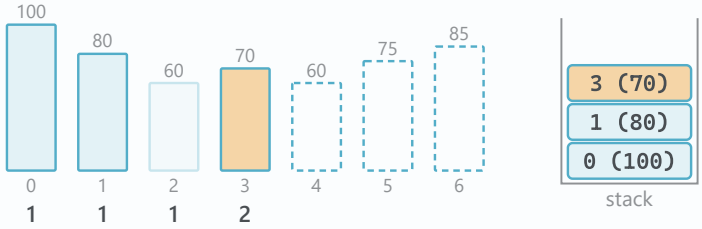
VARIABLES

`i = 3` `prices[i] = 70`

`spans = [1, 1, 1, 2, 0, 0, 0]`

`stack = [0, 1]`

Code Trace: `span({100, 80, 60, 70, 60, 75, 85})`



push(3): a possible boundary for later days.

Step 8 / 19

```
1  int[] span(int[] prices) {
2      int[] spans = new int[prices.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      stack.push(0); spans[0] = 1;
5      for (int i = 1; i < prices.length; i++) {
6          while (!stack.isEmpty() &&
7              prices[i] >= prices[stack.peek()])
8              stack.pop();
9          spans[i] = stack.isEmpty() ? i + 1
10                                    : i - stack.peek();
▶          stack.push(i);
12     }
13     return spans;
14 }
```

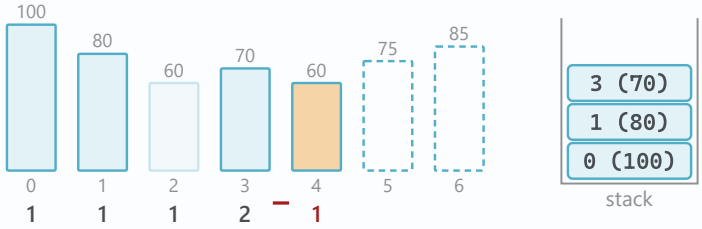
VARIABLES

`i = 3` `prices[i] = 70`

`spans = [1, 1, 1, 2, 0, 0, 0]`

`stack = [0, 1, 3]`

Code Trace: `span({100, 80, 60, 70, 60, 75, 85})`



i = 4 (price 60): the top, day 3 (70), is higher → span = 4 − 3 = **1**.

Step 9 / 19

```
1  int[] span(int[] prices) {
2      int[] spans = new int[prices.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      stack.push(0); spans[0] = 1;
5      for (int i = 1; i < prices.length; i++) {
6          while (!stack.isEmpty() &&
7              prices[i] >= prices[stack.peek()])
8              stack.pop();
▶          spans[i] = stack.isEmpty() ? i + 1
▶                                     : i - stack.peek();
11         stack.push(i);
12     }
13     return spans;
14 }
```

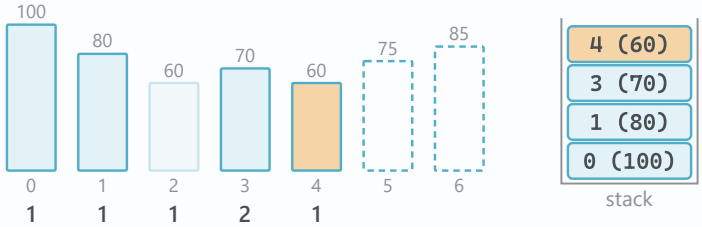
VARIABLES

`i = 4` `prices[i] = 60`

`spans = [1, 1, 1, 2, 1, 0, 0]`

`stack = [0, 1, 3]`

Code Trace: `span({100, 80, 60, 70, 60, 75, 85})`



push(4): a possible boundary for later days.

Step 10 / 19

```
1  int[] span(int[] prices) {
2      int[] spans = new int[prices.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      stack.push(0); spans[0] = 1;
5      for (int i = 1; i < prices.length; i++) {
6          while (!stack.isEmpty() &&
7              prices[i] >= prices[stack.peek()])
8              stack.pop();
9          spans[i] = stack.isEmpty() ? i + 1
10                                    : i - stack.peek();
▶          stack.push(i);
12     }
13     return spans;
14 }
```

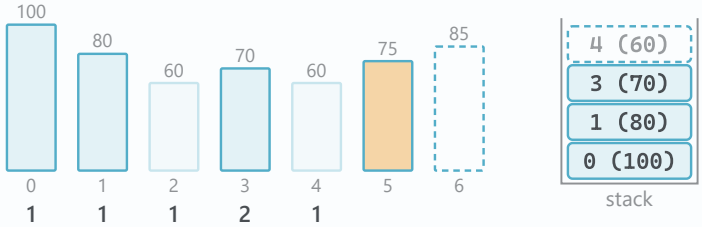
VARIABLES

`i = 4` `prices[i] = 60`

`spans = [1, 1, 1, 2, 1, 0, 0]`

`stack = [0, 1, 3, 4]`

Code Trace: `span({100, 80, 60, 70, 60, 75, 85})`



i = 5: 75 ≥ prices[4] = 60 → **pop**. Day 4 can no longer be anyone's boundary.

Step 11 / 19

```
1  int[] span(int[] prices) {
2      int[] spans = new int[prices.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      stack.push(0); spans[0] = 1;
5      for (int i = 1; i < prices.length; i++) {
6          while (!stack.isEmpty() &&
7              prices[i] >= prices[stack.peek()])
▶             stack.pop();
9          spans[i] = stack.isEmpty() ? i + 1
10                                    : i - stack.peek();
11         stack.push(i);
12     }
13     return spans;
14 }
```

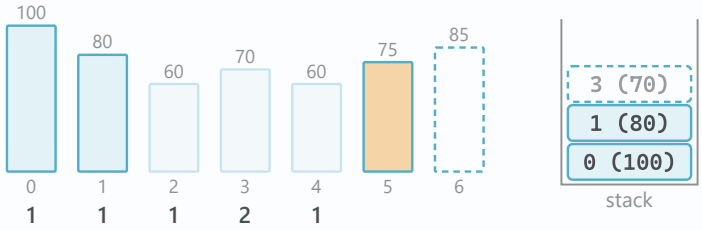
VARIABLES

`i = 5` `prices[i] = 75`

`spans = [1, 1, 1, 2, 1, 0, 0]`

`stack = [0, 1, 3]`

Code Trace: `span({100, 80, 60, 70, 60, 75, 85})`



i = 5: 75 ≥ prices[3] = 70 → **pop**. Day 3 can no longer be anyone's boundary.

Step 12 / 19

```
1  int[] span(int[] prices) {
2      int[] spans = new int[prices.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      stack.push(0); spans[0] = 1;
5      for (int i = 1; i < prices.length; i++) {
6          while (!stack.isEmpty() &&
7                 prices[i] >= prices[stack.peek()])
▶                stack.pop();
9          spans[i] = stack.isEmpty() ? i + 1
10                                    : i - stack.peek();
11         stack.push(i);
12     }
13     return spans;
14 }
```

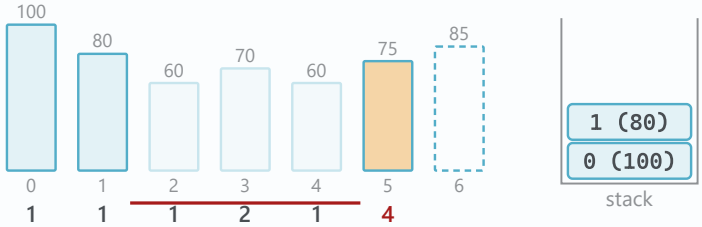
VARIABLES

`i = 5` `prices[i] = 75`

`spans = [1, 1, 1, 2, 1, 0, 0]`

`stack = [0, 1]`

Code Trace: `span({100, 80, 60, 70, 60, 75, 85})`



i = 5 (price 75): the top, day 1 (80), is higher → span = 5 − 1 = **4**.

Step 13 / 19

```
1  int[] span(int[] prices) {
2      int[] spans = new int[prices.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      stack.push(0); spans[0] = 1;
5      for (int i = 1; i < prices.length; i++) {
6          while (!stack.isEmpty() &&
7              prices[i] >= prices[stack.peek()])
8              stack.pop();
▶          spans[i] = stack.isEmpty() ? i + 1
▶                                     : i - stack.peek();
11         stack.push(i);
12     }
13     return spans;
14 }
```

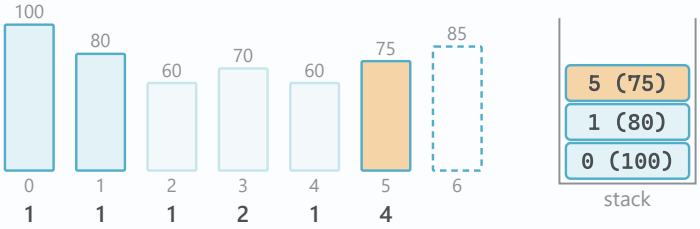
VARIABLES

`i = 5` `prices[i] = 75`

`spans = [1, 1, 1, 2, 1, 4, 0]`

`stack = [0, 1]`

Code Trace: `span({100, 80, 60, 70, 60, 75, 85})`



push(5): a possible boundary for later days.

Step 14 / 19

```
1  int[] span(int[] prices) {
2      int[] spans = new int[prices.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      stack.push(0); spans[0] = 1;
5      for (int i = 1; i < prices.length; i++) {
6          while (!stack.isEmpty() &&
7              prices[i] >= prices[stack.peek()])
8              stack.pop();
9          spans[i] = stack.isEmpty() ? i + 1
10                                    : i - stack.peek();
▶          stack.push(i);
12     }
13     return spans;
14 }
```

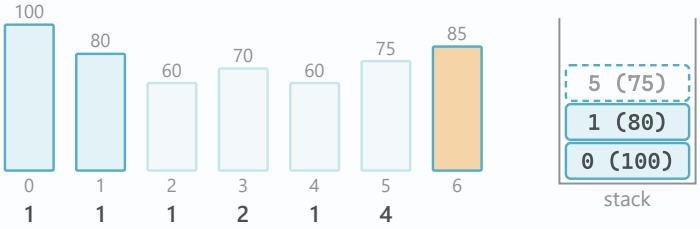
VARIABLES

`i = 5` `prices[i] = 75`

`spans = [1, 1, 1, 2, 1, 4, 0]`

`stack = [0, 1, 5]`

Code Trace: `span({100, 80, 60, 70, 60, 75, 85})`



i = 6: 85 ≥ prices[5] = 75 → **pop**. Day 5 can no longer be anyone's boundary.

Step 15 / 19

```
1  int[] span(int[] prices) {
2      int[] spans = new int[prices.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      stack.push(0); spans[0] = 1;
5      for (int i = 1; i < prices.length; i++) {
6          while (!stack.isEmpty() &&
7                 prices[i] >= prices[stack.peek()])
▶                stack.pop();
9          spans[i] = stack.isEmpty() ? i + 1
10                                    : i - stack.peek();
11         stack.push(i);
12     }
13     return spans;
14 }
```

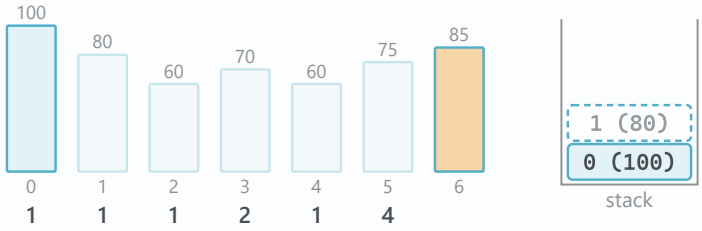
VARIABLES

`i = 6` `prices[i] = 85`

`spans = [1, 1, 1, 2, 1, 4, 0]`

`stack = [0, 1]`

Code Trace: `span({100, 80, 60, 70, 60, 75, 85})`



i = 6: 85 ≥ prices[1] = 80 → **pop**. Day 1 can no longer be anyone's boundary.

Step 16 / 19

```
1  int[] span(int[] prices) {
2      int[] spans = new int[prices.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      stack.push(0); spans[0] = 1;
5      for (int i = 1; i < prices.length; i++) {
6          while (!stack.isEmpty() &&
7                 prices[i] >= prices[stack.peek()])
▶                stack.pop();
9          spans[i] = stack.isEmpty() ? i + 1
10                                    : i - stack.peek();
11         stack.push(i);
12     }
13     return spans;
14 }
```

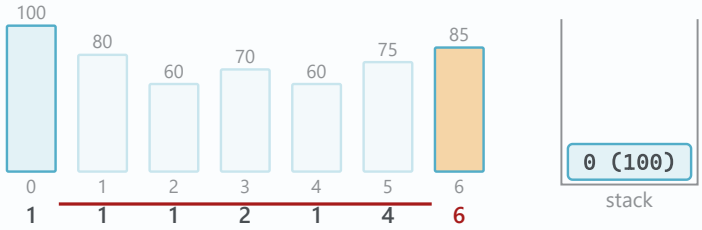
VARIABLES

`i = 6` `prices[i] = 85`

`spans = [1, 1, 1, 2, 1, 4, 0]`

`stack = [0]`

Code Trace: `span({100, 80, 60, 70, 60, 75, 85})`



i = 6 (price 85): the top, day 0 (100), is higher → span = 6 − 0 = **6**.

Step 17 / 19

```
1  int[] span(int[] prices) {
2      int[] spans = new int[prices.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      stack.push(0); spans[0] = 1;
5      for (int i = 1; i < prices.length; i++) {
6          while (!stack.isEmpty() &&
7              prices[i] >= prices[stack.peek()])
8              stack.pop();
▶          spans[i] = stack.isEmpty() ? i + 1
▶                                     : i - stack.peek();
11         stack.push(i);
12     }
13     return spans;
14 }
```

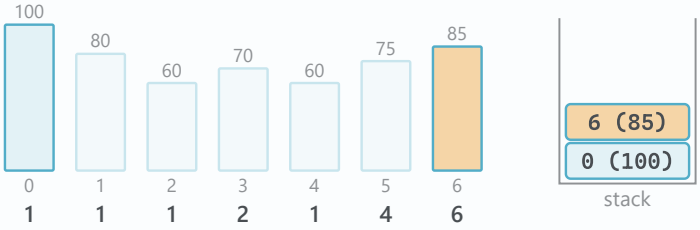
VARIABLES

`i = 6` `prices[i] = 85`

`spans = [1, 1, 1, 2, 1, 4, 6]`

`stack = [0]`

Code Trace: `span({100, 80, 60, 70, 60, 75, 85})`



push(6): a possible boundary for later days.

Step 18 / 19

```
1  int[] span(int[] prices) {
2      int[] spans = new int[prices.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      stack.push(0); spans[0] = 1;
5      for (int i = 1; i < prices.length; i++) {
6          while (!stack.isEmpty() &&
7              prices[i] >= prices[stack.peek()])
8              stack.pop();
9          spans[i] = stack.isEmpty() ? i + 1
10                                    : i - stack.peek();
▶          stack.push(i);
12     }
13     return spans;
14 }
```

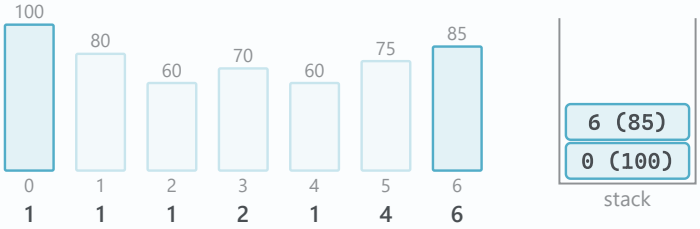
VARIABLES

`i = 6` `prices[i] = 85`

`spans = [1, 1, 1, 2, 1, 4, 6]`

`stack = [0, 6]`

Code Trace: `span({100, 80, 60, 70, 60, 75, 85})`



return [1, 1, 1, 2, 1, 4, 6]. Each index is pushed once and popped at most once → **O(n)**.

Step 19 / 19

```
1  int[] span(int[] prices) {
2      int[] spans = new int[prices.length];
3      Deque<Integer> stack = new ArrayDeque<>();
4      stack.push(0); spans[0] = 1;
5      for (int i = 1; i < prices.length; i++) {
6          while (!stack.isEmpty() &&
7              prices[i] >= prices[stack.peek()])
8              stack.pop();
9          spans[i] = stack.isEmpty() ? i + 1
10                                    : i - stack.peek();
11         stack.push(i);
12     }
▶      return spans;
14 }
```

VARIABLES

`spans = [1, 1, 1, 2, 1, 4, 6]`

`stack = [0, 6]`

EXPRESSION NOTATION

Infix, Prefix, and Postfix Notation

Notation	Where the operator goes	A * (B + C) / D
Infix	between the operands	`A * ( B + C ) / D`
Prefix · Polish notation	before the operands	`/ * A + B C D`
Postfix · reverse Polish (RPN)	after the operands	`A B C + * D /`

- Prefix and postfix need **no parentheses and no precedence rules**: a stack evaluates the expression in a single left-to-right (or right-to-left) pass.
- Compilers and interpreters usually translate expressions into postfix form (e.g., stack-based bytecode), and some calculators use RPN.
- They also appear in symbolic algebra and in work with expression trees.

Careful: the prefix form `* A / + B C D` is wrong; it means `A * ((B + C) / D)`. The correct prefix, which keeps the left-to-right order of operations, is `/ * A + B C D`.

EXPRESSION NOTATION

Precedence, Associativity, and Converting by Hand

Operator	Precedence	Associativity
`^` (power)	3	right to left
`* /`	2	left to right
`+ -`	1	left to right

Equal precedence groups left to right: `A - B + C` = `(A - B) + C` → postfix `A B - C +`.

By hand: full parenthesization

1. Parenthesize every operation: `((A * (B + C)) / D)`
2. **Postfix**: move each operator to the **right** of its parentheses → `A B C + * D /`
3. **Prefix**: move each operator to the **left** of its parentheses → `/ * A + B C D`

The operands (A, B, C, D) appear in the **same order** in all three notations; only the operators move. That makes a quick check on any conversion.

EXPRESSION NOTATION

Infix → Postfix: The Algorithm

Read the input from left to right, holding operators and opening parentheses on a stack.

Token read	Action
Operand (A, B, 5 …)	Write it straight to the output.
`(`	Push it.
`)`	Pop to the output until you reach `(`; discard the `(`.
Operator op	While the top is not `(` and has precedence ≥ op, pop it to the output; then push op.
End of input	Pop every remaining operator to the output.

The "≥" rule gives left-to-right associativity. For right-associative `^`, pop only operators with precedence >. Time O(n).

Infix → Postfix: Java

```java
String infixToPostfix(String expr) {
  StringBuilder out = new StringBuilder();
  Deque<Character> stack = new ArrayDeque<>();
  for (char c : expr.toCharArray()) {
    if (Character.isLetterOrDigit(c)) out.append(c);
    else if (c == '(') stack.push(c);
    else if (c == ')') {
      while (stack.peek() != '(')
        out.append(stack.pop());
      stack.pop();                    // discard '('
    } else {                          // operator
      while (!stack.isEmpty() &&
          prec(stack.peek()) >= prec(c))
        out.append(stack.pop());
      stack.push(c);
    }
  }
  while (!stack.isEmpty()) out.append(stack.pop());
  return out.toString();
}
```

```java
int prec(char op) {
  switch (op) {
    case '+': case '-': return 1;
    case '*': case '/': return 2;
    default: return 0;   // '('
  }
}
```

Assumptions

Single-character operands, only + − * / (no ^), no spaces, and a valid expression.

Why (→ 0?

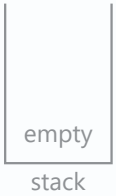
An incoming operator never pops `(`; only `)` discards it.

Code Trace: `infixToPostfix("A*(B+C)/D")`

expr

A	*	(	B	+	C	)	/	D

out:



The input is read left to right. Operands go to the output; operators and '(' go on the stack.

Step 1 / 13

```
▶ for (char c : expr.toCharArray()) {
2   if (Character.isLetterOrDigit(c)) out.append(c);
3   else if (c == '(') stack.push(c);
4   else if (c == ')') {
5     while (stack.peek() != '(')
6       out.append(stack.pop());
7     stack.pop();                  // discard '('
8   } else {                        // operator
9     while (!stack.isEmpty() &&
10        prec(stack.peek()) >= prec(c))
11      out.append(stack.pop());
12    stack.push(c);
13  }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`out = ""` `stack = []`

Code Trace: `infixToPostfix("A*(B+C)/D")`

expr

A	*	(	B	+	C	)	/	D

▲ c

out: **A**



A is an operand → straight to the output.

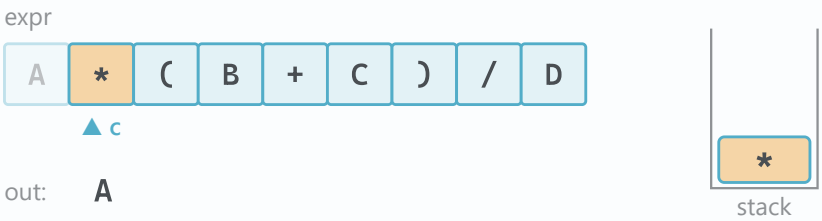
Step 2 / 13

```
1  for (char c : expr.toCharArray()) {
▶    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                  // discard '('
8    } else {                        // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) >= prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = 'A'` `out = "A"` `stack = []`

Code Trace: `infixToPostfix("A*(B+C)/D")`



*: the stack is empty → **push** *.

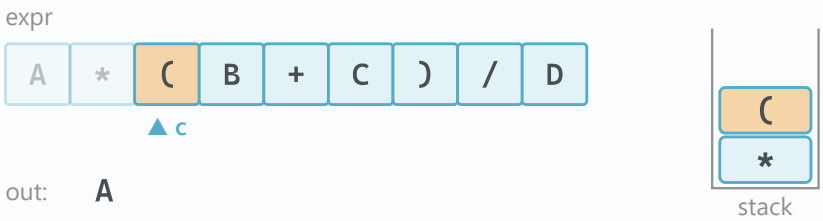
Step 3 / 13

```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                    // discard '('
8    } else {                          // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) >= prec(c))
11       out.append(stack.pop());
▶      stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = '*'` `out = "A"` `stack = [*]`

Code Trace: `infixToPostfix("A*(B+C)/D")`



(→ always pushed. It shields the operators inside until it is closed.

Step 4 / 13

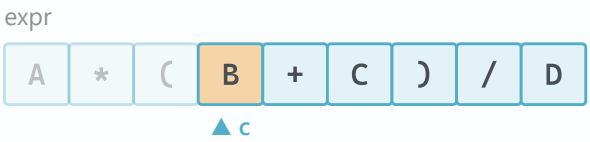
```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
▶    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                  // discard '('
8    } else {                        // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) >= prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

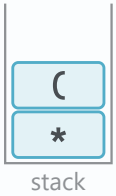
VARIABLES

`c = '('` `out = "A"`

`stack = [*, (]`

Code Trace: `infixToPostfix("A*(B+C)/D")`





B is an operand → straight to the output.

Step 5 / 13

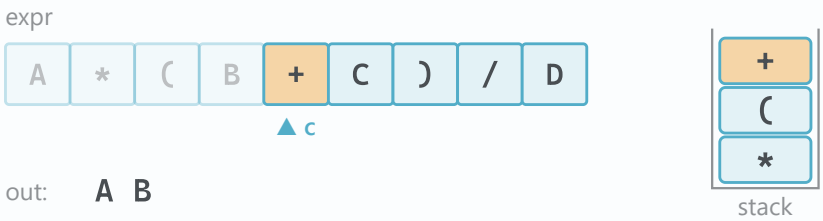
```
1  for (char c : expr.toCharArray()) {
▶    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                  // discard '('
8    } else {                        // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) >= prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = 'B'` `out = "AB"`

`stack = [*, (]`

Code Trace: `infixToPostfix("A*(B+C)/D")`



+: '(' is on top → **push** +.

Step 6 / 13

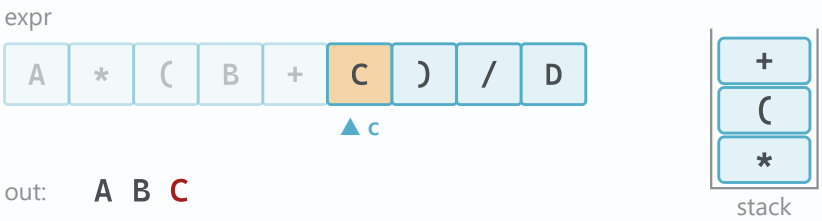
```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                    // discard '('
8    } else {                          // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) >= prec(c))
11       out.append(stack.pop());
▶      stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = '+'` `out = "AB"`

`stack = [*, (, +]`

Code Trace: `infixToPostfix("A*(B+C)/D")`



C is an operand → straight to the output.

Step 7 / 13

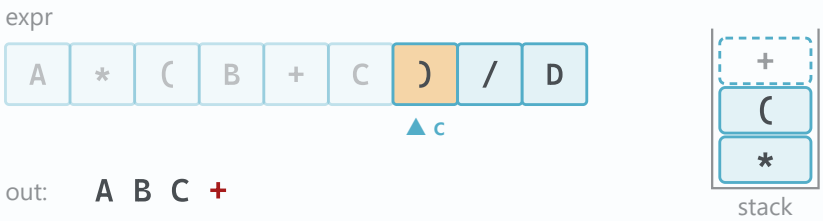
```
1  for (char c : expr.toCharArray()) {
▶    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                    // discard '('
8    } else {                          // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) >= prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = 'C'` `out = "ABC"`

`stack = [*, (, +]`

Code Trace: `infixToPostfix("A*(B+C)/D")`



): pop until '(' → **+** goes to the output.

Step 8 / 13

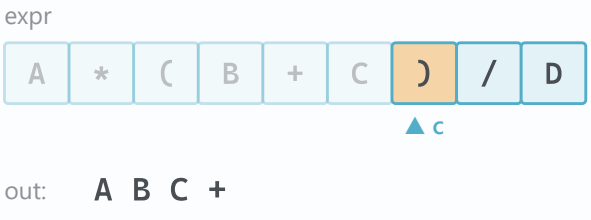
```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
▶      while (stack.peek() != '(')
▶        out.append(stack.pop());
7      stack.pop();                  // discard '('
8    } else {                        // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) >= prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

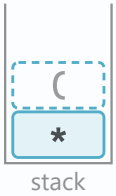
VARIABLES

`c = ')'` `out = "ABC+"`

`stack = [*, (]`

Code Trace: `infixToPostfix("A*(B+C)/D")`





'(' is popped and discarded; parentheses **never** reach the output.

Step 9 / 13

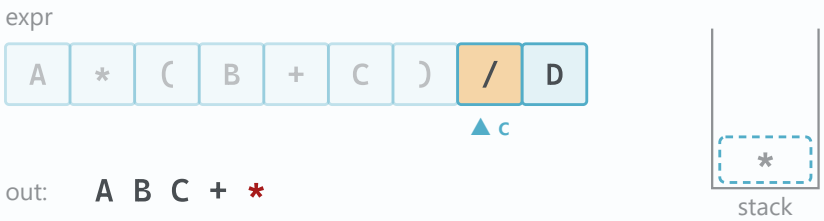
```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
▶      stack.pop();                    // discard '('
8    } else {                          // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) >= prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = ')'` `out = "ABC+"`

`stack = [*]`

Code Trace: `infixToPostfix("A*(B+C)/D")`



/: precedence of * on top (2) = precedence of / (2) → * goes to the output.

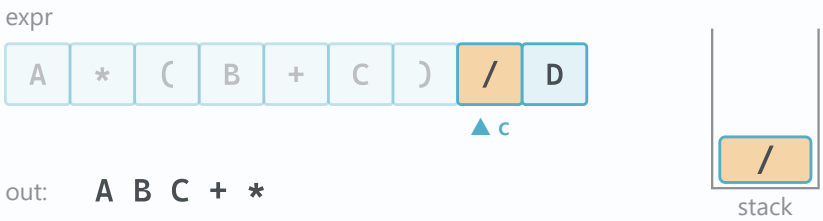
Step 10 / 13

```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                    // discard '('
8    } else {                          // operator
▶      while (!stack.isEmpty() &&
▶          prec(stack.peek()) >= prec(c))
▶        out.append(stack.pop());
12     stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = '/'` `out = "ABC+*"` `stack = []`

Code Trace: `infixToPostfix("A*(B+C)/D")`



/: the stack is empty → **push /**.

Step 11 / 13

```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                    // discard '('
8    } else {                          // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) >= prec(c))
11       out.append(stack.pop());
▶      stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = '/'` `out = "ABC+*"`

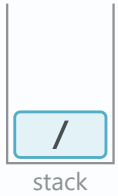
`stack = [/]`

Code Trace: `infixToPostfix("A*(B+C)/D")`

expr

A	*	(	B	+	C	)	/	D
								▲ c

out: **A B C + * D**



D is an operand → straight to the output.

Step 12 / 13

```
1  for (char c : expr.toCharArray()) {
▶    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                  // discard '('
8    } else {                        // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) >= prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = 'D'` `out = "ABC+*D"` `stack = [/]`

Code Trace: `infixToPostfix("A*(B+C)/D")`

expr

A	*	(	B	+	C	)	/	D

out: **A B C + * D /**



End of input: the remaining operator **/** goes to the output.
Result: **A B C + * D /**

Step 13 / 13

```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                  // discard '('
8    } else {                        // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) >= prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
▶ while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`out = "ABC+*D/"` `stack = []`

Code Trace: `infixToPostfix("A+B*C-D/E")`

expr

A	+	B	*	C	-	D	/	E

out:



stack

The input is read left to right. Operands go to the output; operators and '(' go on the stack.

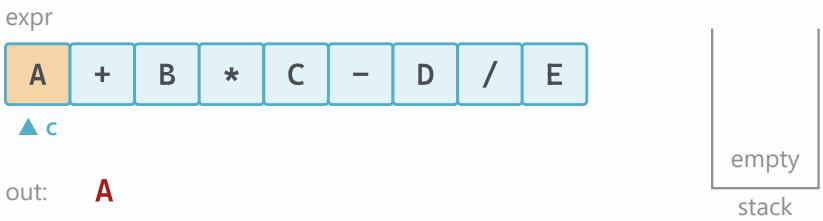
Step 1 / 14

```
▶ for (char c : expr.toCharArray()) {
2   if (Character.isLetterOrDigit(c)) out.append(c);
3   else if (c == '(') stack.push(c);
4   else if (c == ')') {
5     while (stack.peek() != '(')
6       out.append(stack.pop());
7     stack.pop();                  // discard '('
8   } else {                        // operator
9     while (!stack.isEmpty() &&
10        prec(stack.peek()) >= prec(c))
11      out.append(stack.pop());
12    stack.push(c);
13  }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`out = ""` `stack = []`

Code Trace: `infixToPostfix("A+B*C-D/E")`



A is an operand → straight to the output.

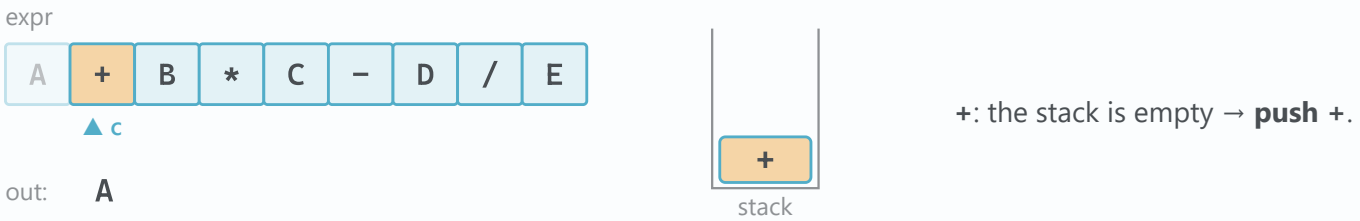
Step 2 / 14

```
1  for (char c : expr.toCharArray()) {
▶    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                    // discard '('
8    } else {                          // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) >= prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = 'A'` `out = "A"` `stack = []`

Code Trace: `infixToPostfix("A+B*C-D/E")`



+: the stack is empty → **push** +.

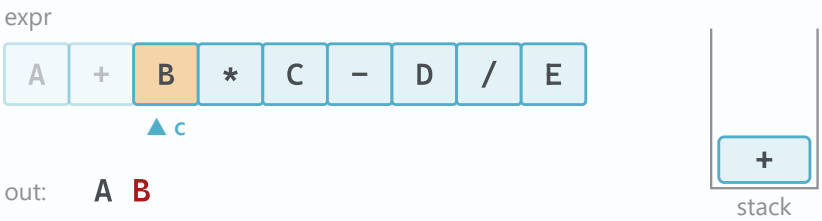
Step 3 / 14

```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                    // discard '('
8    } else {                          // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) >= prec(c))
11       out.append(stack.pop());
▶      stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = '+'` `out = "A"` `stack = [+]`

Code Trace: `infixToPostfix("A+B*C-D/E")`



B is an operand → straight to the output.

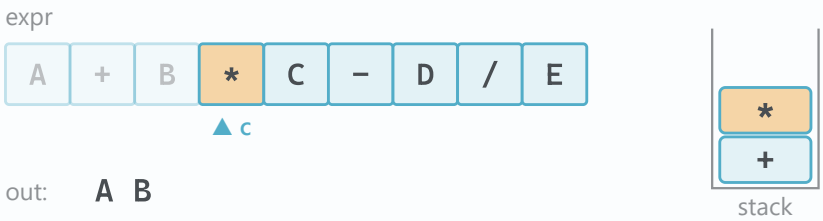
Step 4 / 14

```
1  for (char c : expr.toCharArray()) {
▶    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                    // discard '('
8    } else {                          // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) >= prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = 'B'` `out = "AB"` `stack = [+]`

Code Trace: `infixToPostfix("A+B*C-D/E")`



*: precedence of + on top (1) < 2 → **push** *.

Step 5 / 14

```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                    // discard '('
8    } else {                          // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) >= prec(c))
11       out.append(stack.pop());
▶      stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = '*'` `out = "AB"` `stack = [+, *]`

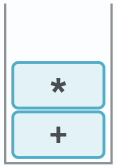
Code Trace: `infixToPostfix("A+B*C-D/E")`

expr

A	+	B	*	C	-	D	/	E

▲ c

out: **A B C**



stack

C is an operand → straight to the output.

Step 6 / 14

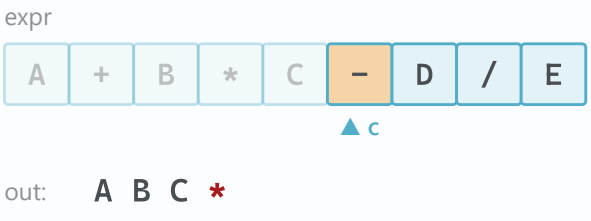
```
1  for (char c : expr.toCharArray()) {
▶    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                  // discard '('
8    } else {                        // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) >= prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = 'C'` `out = "ABC"`

`stack = [+, *]`

Code Trace: `infixToPostfix("A+B*C-D/E")`



-: precedence of * on top (2) > precedence of - (1) → * goes to the output.

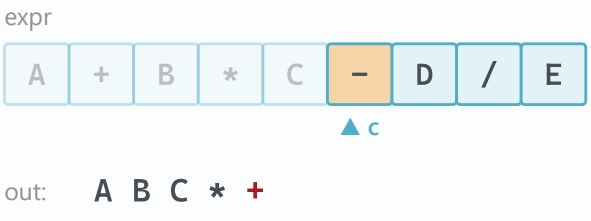
Step 7 / 14

```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                    // discard '('
8    } else {                          // operator
▶      while (!stack.isEmpty() &&
▶          prec(stack.peek()) >= prec(c))
▶        out.append(stack.pop());
12     stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = '-'` `out = "ABC*"` `stack = [+]`

Code Trace: `infixToPostfix("A+B*C-D/E")`



-: precedence of + on top (1) = precedence of - (1) → + goes to the output.

Step 8 / 14

```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                    // discard '('
8    } else {                          // operator
▶      while (!stack.isEmpty() &&
▶          prec(stack.peek()) >= prec(c))
▶        out.append(stack.pop());
12     stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = '-'` `out = "ABC*+"` `stack = []`

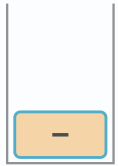
Code Trace: `infixToPostfix("A+B*C-D/E")`

expr

A	+	B	*	C	-	D	/	E

▲ c

out: **A B C * +**



stack

-: the stack is empty → **push -**.

Step 9 / 14

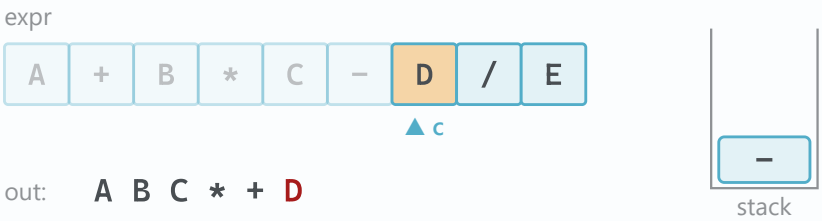
```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                    // discard '('
8    } else {                          // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) >= prec(c))
11       out.append(stack.pop());
▶      stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = '-'` `out = "ABC*+"`

`stack = [-]`

Code Trace: `infixToPostfix("A+B*C-D/E")`



D is an operand → straight to the output.

Step 10 / 14

```
1  for (char c : expr.toCharArray()) {
▶    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                    // discard '('
8    } else {                          // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) >= prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = 'D'` `out = "ABC*+D"` `stack = [-]`

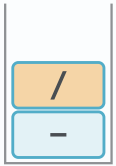
Code Trace: `infixToPostfix("A+B*C-D/E")`

expr

A	+	B	*	C	-	D	/	E

▲ c

out: **A B C * + D**



stack

/: precedence of - on top (1) < 2 → **push /**.

Step 11 / 14

```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                    // discard '('
8    } else {                          // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) >= prec(c))
11       out.append(stack.pop());
▶      stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

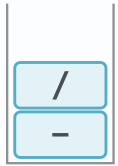
`c = '/'` `out = "ABC*+D"` `stack = [-, /]`

Code Trace: `infixToPostfix("A+B*C-D/E")`

expr

A	+	B	*	C	-	D	/	E
								▲ c

out: **A B C * + D E**



stack

E is an operand → straight to the output.

Step 12 / 14

```
1  for (char c : expr.toCharArray()) {
▶    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                  // discard '('
8    } else {                        // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) >= prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = 'E'` `out = "ABC*+DE"` `stack = [-, /]`

Code Trace: `infixToPostfix("A+B*C-D/E")`

expr

A	+	B	*	C	-	D	/	E

out: **A B C * + D E /**



stack

End of input: the remaining operator **/** goes to the output.

Step 13 / 14

```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                    // discard '('
8    } else {                          // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) >= prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
▶ while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

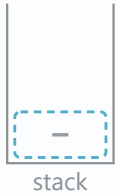
`out = "ABC*+DE/"` `stack = [-]`

Code Trace: `infixToPostfix("A+B*C-D/E")`

expr

A	+	B	*	C	-	D	/	E

out: **A B C * + D E / -**



End of input: the remaining operator **-** goes to the output. Result: **A B C * + D E / −** = (A + B·C) − D/E

Step 14 / 14

```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                    // discard '('
8    } else {                          // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) >= prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
▶ while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`out = "ABC*+DE/-"` `stack = []`

Evaluating a Postfix Expression

- Read left to right.
- **Number** → push it.
- **Operator** → pop twice, the **right** operand first and then the **left**; push the result.
- At the end, the one item left on the stack is the answer. No parentheses or precedence → O(n).

```
int apply(int left, char op, int right) {
  switch (op) {
    case '+': return left + right;
    case '-': return left - right;
    case '*': return left * right;
    default:  return left / right;
  }
}
```

```
int evaluatePostfix(String expr) {
  Deque<Integer> stack = new ArrayDeque<>();
  for (char c : expr.toCharArray()) {
    if (Character.isDigit(c)) {
      stack.push(c - '0');
    } else {
      int right = stack.pop();
      int left = stack.pop();
      stack.push(apply(left, c, right));
    }
  }
  return stack.pop();
}
```

Assumes single-digit numbers, no spaces, and a valid expression.

Code Trace: `evaluatePostfix("412+*3/")`

expr





With A=4, B=1, C=2, D=3, **A B C + * D /** → "412+*3/". Create an empty operand stack.

Step 1 / 12

```
1  int evaluatePostfix(String expr) {
▶    Deque<Integer> stack = new ArrayDeque<>
   ();
3    for (char c : expr.toCharArray()) {
4      if (Character.isDigit(c)) {
5        stack.push(c - '0');
6      } else {
7        int right = stack.pop();
8        int left = stack.pop();
9        stack.push(apply(left, c, right));
10     }
11   }
12   return stack.pop();
13 }
```

VARIABLES

`stack = []`

Code Trace: `evaluatePostfix("412+*3/")`

expr

4	1	2	+	*	3	/

▲ c



4 is a digit → push `c - '0'` = 4.

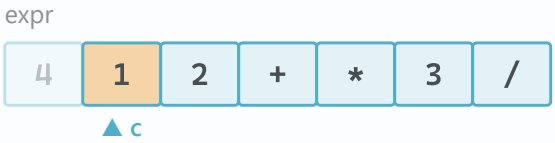
Step 2 / 12

```
1  int evaluatePostfix(String expr) {
2    Deque<Integer> stack = new ArrayDeque<>
   ();
3    for (char c : expr.toCharArray()) {
▶      if (Character.isDigit(c)) {
▶        stack.push(c - '0');
6      } else {
7        int right = stack.pop();
8        int left = stack.pop();
9        stack.push(apply(left, c, right));
10     }
11   }
12   return stack.pop();
13 }
```

VARIABLES

`c = '4'` `stack = [4]`

Code Trace: `evaluatePostfix("412+*3/")`





1 is a digit → push `c - '0'` = 1.

Step 3 / 12

```
1  int evaluatePostfix(String expr) {
2    Deque<Integer> stack = new ArrayDeque<>
   ();
3    for (char c : expr.toCharArray()) {
▶      if (Character.isDigit(c)) {
▶        stack.push(c - '0');
6      } else {
7        int right = stack.pop();
8        int left = stack.pop();
9        stack.push(apply(left, c, right));
10     }
11   }
12   return stack.pop();
13 }
```

VARIABLES

`c = '1'` `stack = [4, 1]`

Code Trace: `evaluatePostfix("412+*3/")`

expr





2 is a digit → push `c - '0'` = 2.

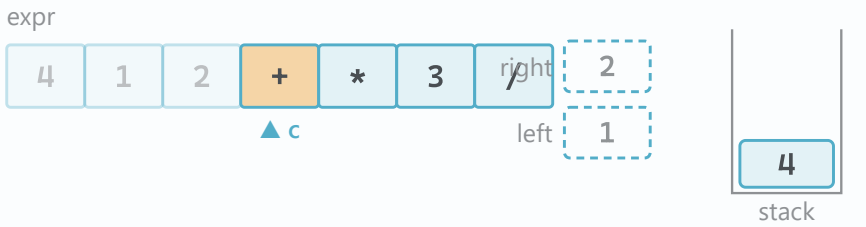
Step 4 / 12

```
1  int evaluatePostfix(String expr) {
2    Deque<Integer> stack = new ArrayDeque<>
   ();
3    for (char c : expr.toCharArray()) {
▶      if (Character.isDigit(c)) {
▶        stack.push(c - '0');
6      } else {
7        int right = stack.pop();
8        int left = stack.pop();
9        stack.push(apply(left, c, right));
10     }
11   }
12   return stack.pop();
13 }
```

VARIABLES

`c = '2'` `stack = [4, 1, 2]`

Code Trace: `evaluatePostfix("412+*3/")`



+ is an operator: pop **right = 2** first, then **left = 1** (order matters for − and /).

Step 5 / 12

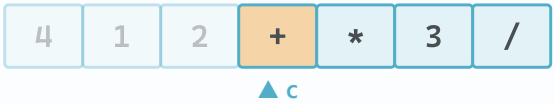
```
1  int evaluatePostfix(String expr) {
2    Deque<Integer> stack = new ArrayDeque<>
   ();
3    for (char c : expr.toCharArray()) {
4      if (Character.isDigit(c)) {
5        stack.push(c - '0');
6      } else {
▶        int right = stack.pop();
▶        int left = stack.pop();
9        stack.push(apply(left, c, right));
10     }
11   }
12   return stack.pop();
13 }
```

VARIABLES

`c = '+'` `right = 2` `left = 1` `stack = [4]`

Code Trace: `evaluatePostfix("412+*3/")`

expr



1 + 2 = 3



apply(1, '+', 2) = **3** → push the result back.

Step 6 / 12

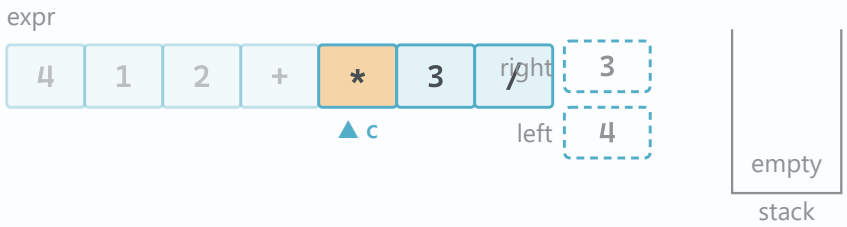
```
1  int evaluatePostfix(String expr) {
2    Deque<Integer> stack = new ArrayDeque<>
   ();
3    for (char c : expr.toCharArray()) {
4      if (Character.isDigit(c)) {
5        stack.push(c - '0');
6      } else {
7        int right = stack.pop();
8        int left = stack.pop();
▶        stack.push(apply(left, c, right));
10     }
11   }
12   return stack.pop();
13 }
```

VARIABLES

`c = '+'` `right = 2` `left = 1`

`stack = [4, 3]`

Code Trace: `evaluatePostfix("412+*3/")`



* is an operator: pop **right = 3** first, then **left = 4** (order matters for − and /).

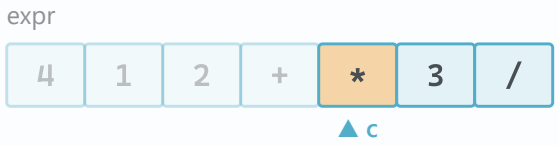
Step 7 / 12

```
1  int evaluatePostfix(String expr) {
2    Deque<Integer> stack = new ArrayDeque<>
   ();
3    for (char c : expr.toCharArray()) {
4      if (Character.isDigit(c)) {
5        stack.push(c - '0');
6      } else {
▶        int right = stack.pop();
▶        int left = stack.pop();
9        stack.push(apply(left, c, right));
10     }
11   }
12   return stack.pop();
13 }
```

VARIABLES

`c = '*'` `right = 3` `left = 4` `stack = []`

Code Trace: `evaluatePostfix("412+*3/")`



4 * 3 = 12



apply(4, '*', 3) = **12** → push the result back.

Step 8 / 12

```
1  int evaluatePostfix(String expr) {
2    Deque<Integer> stack = new ArrayDeque<>
   ();
3    for (char c : expr.toCharArray()) {
4      if (Character.isDigit(c)) {
5        stack.push(c - '0');
6      } else {
7        int right = stack.pop();
8        int left = stack.pop();
▶        stack.push(apply(left, c, right));
10     }
11   }
12   return stack.pop();
13 }
```

VARIABLES

`c = '*'` `right = 3` `left = 4`

`stack = [12]`

Code Trace: `evaluatePostfix("412+*3/")`





3 is a digit → push `c - '0'` = 3.

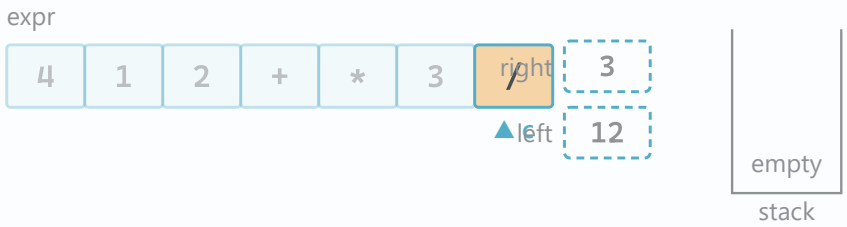
Step 9 / 12

```
1  int evaluatePostfix(String expr) {
2    Deque<Integer> stack = new ArrayDeque<>
   ();
3    for (char c : expr.toCharArray()) {
▶      if (Character.isDigit(c)) {
▶        stack.push(c - '0');
6      } else {
7        int right = stack.pop();
8        int left = stack.pop();
9        stack.push(apply(left, c, right));
10     }
11   }
12   return stack.pop();
13 }
```

VARIABLES

`c = '3'` `stack = [12, 3]`

Code Trace: `evaluatePostfix("412+*3/")`



/ is an operator: pop **right = 3** first, then **left = 12** (order matters for − and /).

Step 10 / 12

```
1  int evaluatePostfix(String expr) {
2    Deque<Integer> stack = new ArrayDeque<>
   ();
3    for (char c : expr.toCharArray()) {
4      if (Character.isDigit(c)) {
5        stack.push(c - '0');
6      } else {
▶        int right = stack.pop();
▶        int left = stack.pop();
9        stack.push(apply(left, c, right));
10     }
11   }
12   return stack.pop();
13 }
```

VARIABLES

`c = '/'` `right = 3` `left = 12` `stack = []`

Code Trace: `evaluatePostfix("412+*3/")`

expr

4	1	2	+	*	3	/

▲ c

12 / 3 = 4



apply(12, '/', 3) = **4** → push the result back.

Step 11 / 12

```
1  int evaluatePostfix(String expr) {
2    Deque<Integer> stack = new ArrayDeque<>
   ();
3    for (char c : expr.toCharArray()) {
4      if (Character.isDigit(c)) {
5        stack.push(c - '0');
6      } else {
7        int right = stack.pop();
8        int left = stack.pop();
▶        stack.push(apply(left, c, right));
10     }
11   }
12   return stack.pop();
13 }
```

VARIABLES

`c = '/'` `right = 3` `left = 12`

`stack = [4]`

Code Trace: `evaluatePostfix("412+*3/")`

expr

4	1	2	+	*	3	/

result = 4



The expression is done; the single value on the stack is the answer: **return 4**. No parentheses or precedence needed → O(n).

Step 12 / 12

```
1  int evaluatePostfix(String expr) {
2    Deque<Integer> stack = new ArrayDeque<>
   ();
3    for (char c : expr.toCharArray()) {
4      if (Character.isDigit(c)) {
5        stack.push(c - '0');
6      } else {
7        int right = stack.pop();
8        int left = stack.pop();
9        stack.push(apply(left, c, right));
10     }
11   }
▶    return stack.pop();
13 }
```

VARIABLES

`stack = []`

EXPRESSION NOTATION

Infix → Prefix Conversion

1. **Reverse** the infix expression.
2. **Swap** the parentheses: `(` ↔ `)`.
3. Run the postfix algorithm, but **don't pop** on equal precedence; pop only operators with precedence **>**.
4. **Reverse** the output.

Step	A * (B + C) / D
1–2. Reverse, swap parentheses	`D / ( C + B ) * A`
3. Postfix (no pop on equal)	`D C B + A * /`
4. Reverse → **prefix**	`/ * A + B C D`

In step 3, when `*` arrives, `/` of equal precedence is on top and is not popped. After the final reversal this preserves left associativity: first `A * (B + C)`, then `/ D`.

Example

`A + B * C - D / E` → `- + A * B C / D E`

Prefix evaluation

Like postfix, but read the expression **right to left**; the first operand popped is the **left** one.

Code Trace: `infixToPrefix("A*(B+C)/D")`

expr

A	*	(	B	+	C	)	/	D

main calls `infixToPrefix("A*(B+C)/D")`.

Step 1 / 6

```
1  String rev(String s) {
2    return new StringBuilder(s).reverse().toString();
3  }
4  String infixToPrefix(String expr) {
5    String t = rev(expr);              // 1. reverse
6    t = swapParens(t);                 // 2. ( ↔ )
7    String p = postfixForPrefix(t);    // 3. no pop on equal
8    return rev(p);                     // 4. reverse
9  }
10 // in main:
▶  System.out.println(infixToPrefix("A*(B+C)/D"));
```

VARIABLES

—

CALL STACK

`main()`

OUTPUT

Code Trace: `infixToPrefix("A*(B+C)/D")`

expr	A	*	(	B	+	C	)	/	D
t = rev(expr)	D	/	)	C	+	B	(	*	A

1. The string is **reversed**. The parentheses flip too: ')' is now on the left, '(' on the right.

Step 2 / 6

```
1  String rev(String s) {
2    return new StringBuilder(s).reverse().toString();
3  }
4  String infixToPrefix(String expr) {
▶    String t = rev(expr);              // 1. reverse
6    t = swapParens(t);                 // 2. ( ↔ )
7    String p = postfixForPrefix(t);    // 3. no pop on equal
8    return rev(p);                     // 4. reverse
9  }
10 // in main:
11 System.out.println(infixToPrefix("A*(B+C)/D"));
```

VARIABLES

`t = "D/)C+B(*A"`

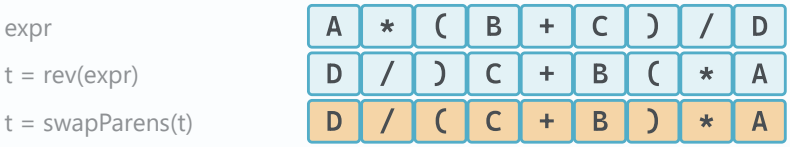
CALL STACK

`infixToPrefix("A*(B+C)/D")`

`main()`

OUTPUT

Code Trace: `infixToPrefix("A*(B+C)/D")`



2. Swap **(↔)**; now it is a valid infix expression: **D / (C + B) * A**.

Step 3 / 6

```
1  String rev(String s) {
2    return new StringBuilder(s).reverse().toString();
3  }
4  String infixToPrefix(String expr) {
5    String t = rev(expr);              // 1. reverse
▶    t = swapParens(t);                 // 2. ( ↔ )
7    String p = postfixForPrefix(t);    // 3. no pop on equal
8    return rev(p);                     // 4. reverse
9  }
10 // in main:
11 System.out.println(infixToPrefix("A*(B+C)/D"));
```

VARIABLES

`t = "D/(C+B)*A"`

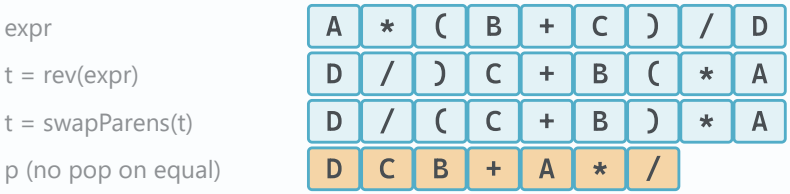
CALL STACK

`infixToPrefix("A*(B+C)/D")`

`main()`

OUTPUT

Code Trace: `infixToPrefix("A*(B+C)/D")`



3. The postfix algorithm, but **without popping** on equal precedence → p = **D C B + A * /**.

Step 4 / 6

```
1  String rev(String s) {
2    return new StringBuilder(s).reverse().toString();
3  }
4  String infixToPrefix(String expr) {
5    String t = rev(expr);              // 1. reverse
6    t = swapParens(t);                 // 2. ( ↔ )
▶    String p = postfixForPrefix(t);    // 3. no pop on equal
8    return rev(p);                     // 4. reverse
9  }
10 // in main:
11 System.out.println(infixToPrefix("A*(B+C)/D"));
```

VARIABLES

`t = "D/(C+B)*A"`

`p = "DCB+A*/"`

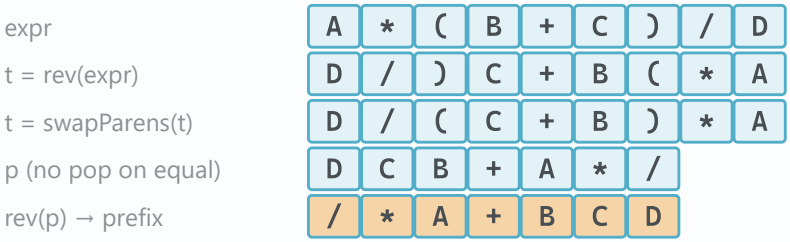
CALL STACK

`infixToPrefix("A*(B+C)/D")`

`main()`

OUTPUT

Code Trace: `infixToPrefix("A*(B+C)/D")`



4. Reverse p → returns **/ * A + B C D**.

Step 5 / 6

```
1  String rev(String s) {
2    return new StringBuilder(s).reverse().toString();
3  }
4  String infixToPrefix(String expr) {
5    String t = rev(expr);              // 1. reverse
6    t = swapParens(t);                 // 2. ( ↔ )
7    String p = postfixForPrefix(t);    // 3. no pop on equal
▶    return rev(p);                     // 4. reverse
9  }
10 // in main:
11 System.out.println(infixToPrefix("A*(B+C)/D"));
```

VARIABLES

`t = "D/(C+B)*A"`

`p = "DCB+A*/"`

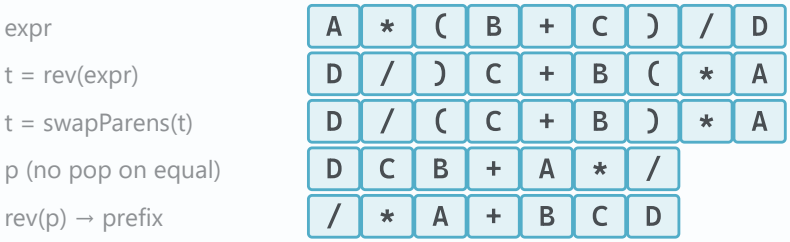
CALL STACK

`infixToPrefix("A*(B+C)/D")`

`main()`

OUTPUT

Code Trace: `infixToPrefix("A*(B+C)/D")`



Step 6 / 6

main prints the result. Check: the operands are still in the order A, B, C, D.

```
1  String rev(String s) {
2    return new StringBuilder(s).reverse().toString();
3  }
4  String infixToPrefix(String expr) {
5    String t = rev(expr);             // 1. reverse
6    t = swapParens(t);                // 2. ( ↔ )
7    String p = postfixForPrefix(t);   // 3. no pop on equal
8    return rev(p);                    // 4. reverse
9  }
10 // in main:
▶  System.out.println(infixToPrefix("A*(B+C)/D"));
```

VARIABLES

—

CALL STACK

`main()`

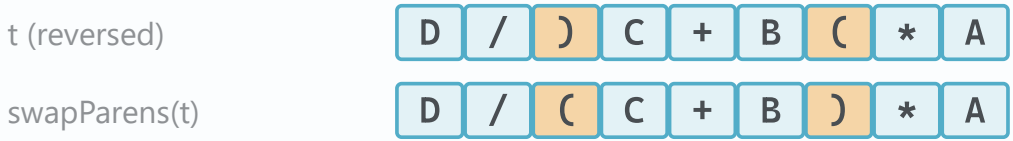
OUTPUT

```
/*A+BCD
```

Step 2: `swapParens`

Reversing also flips the direction of the parentheses. This method writes `)` for every `(` and `(` for every `)`, leaving all other characters unchanged.

```
String swapParens(String s) {
  StringBuilder sb = new StringBuilder();
  for (char c : s.toCharArray()) {
    if (c == '(') sb.append(')');
    else if (c == ')') sb.append('(');
    else sb.append(c);
  }
  return sb.toString();
}
```



One pass over the string: O(n) time, O(n) extra memory.

Code Trace: `postfixForPrefix("D/(C+B)*A")`

t (reversed, parentheses swapped)

D	/	(	C	+	B	)	*	A

out:



stack

The input is read left to right. Operands go to the output; operators and '(' go on the stack.

Step 1 / 13

```
▶ for (char c : expr.toCharArray()) {
2   if (Character.isLetterOrDigit(c)) out.append(c);
3   else if (c == '(') stack.push(c);
4   else if (c == ')') {
5     while (stack.peek() != '(')
6       out.append(stack.pop());
7     stack.pop();                  // discard '('
8   } else {                        // operator
9     while (!stack.isEmpty() &&
10        prec(stack.peek()) > prec(c))
11      out.append(stack.pop());
12    stack.push(c);
13  }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`out = ""` `stack = []`

Code Trace: `postfixForPrefix("D/(C+B)*A")`

t (reversed, parentheses swapped)

D	/	(	C	+	B	)	*	A

▲ c

out: D



D is an operand → straight to the output.

Step 2 / 13

```
1  for (char c : expr.toCharArray()) {
▶    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                  // discard '('
8    } else {                        // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) > prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

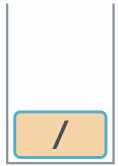
`c = 'D'` `out = "D"` `stack = []`

Code Trace: `postfixForPrefix("D/(C+B)*A")`

t (reversed, parentheses swapped)

D	/	(	C	+	B	)	*	A
	▲ c							

out: D



stack

/: the stack is empty → **push /**.

Step 3 / 13

```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                  // discard '('
8    } else {                        // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) > prec(c))
11       out.append(stack.pop());
▶      stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

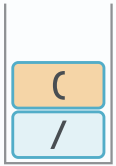
`c = '/'` `out = "D"` `stack = [/]`

Code Trace: `postfixForPrefix("D/(C+B)*A")`

t (reversed, parentheses swapped)

D	/	(	C	+	B	)	*	A
		▲ c						

out: D



stack

(→ always pushed. It shields the operators inside until it is closed.

Step 4 / 13

```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
▶    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                  // discard '('
8    } else {                        // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) > prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = '('` `out = "D"`

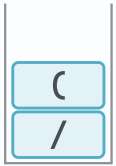
`stack = [/, (]`

Code Trace: `postfixForPrefix("D/(C+B)*A")`

t (reversed, parentheses swapped)

D	/	(	C	+	B	)	*	A
			▲ c					

out: D C



stack

C is an operand → straight to the output.

Step 5 / 13

```
1  for (char c : expr.toCharArray()) {
▶    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                  // discard '('
8    } else {                        // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) > prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = 'C'` `out = "DC"`

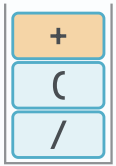
`stack = [/, (]`

Code Trace: `postfixForPrefix("D/(C+B)*A")`

t (reversed, parentheses swapped)

D	/	(	C	+	B	)	*	A
			▲ c					

out: D C



stack

+: '(' is on top → **push** +.

Step 6 / 13

```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                    // discard '('
8    } else {                          // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) > prec(c))
11       out.append(stack.pop());
▶      stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = '+'` `out = "DC"`

`stack = [/, (, +]`

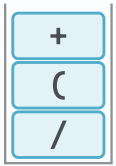
Code Trace: `postfixForPrefix("D/(C+B)*A")`

t (reversed, parentheses swapped)

D	/	(	C	+	B	)	*	A

▲ c

out: D C B



stack

B is an operand → straight to the output.

Step 7 / 13

```
1  for (char c : expr.toCharArray()) {
▶    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                    // discard '('
8    } else {                          // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) > prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = 'B'` `out = "DCB"`

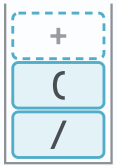
`stack = [/, (, +]`

Code Trace: `postfixForPrefix("D/(C+B)*A")`

t (reversed, parentheses swapped)

D	/	(	C	+	B	)	*	A
						▲ c		

out: D C B +



stack

): pop until '(' → **+** goes to the output.

Step 8 / 13

```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
▶      while (stack.peek() != '(')
▶        out.append(stack.pop());
7      stack.pop();                  // discard '('
8    } else {                        // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) > prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

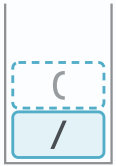
`c = ')'` `out = "DCB+"` `stack = [/, (]`

Code Trace: `postfixForPrefix("D/(C+B)*A")`

t (reversed, parentheses swapped)

D	/	(	C	+	B	)	*	A
						▲ c		

out: D C B +



stack

'(' is popped and discarded; parentheses **never** reach the output.

Step 9 / 13

```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
▶      stack.pop();                    // discard '('
8    } else {                          // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) > prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = ')'` `out = "DCB+"`

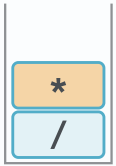
`stack = [/]`

Code Trace: `postfixForPrefix("D/(C+B)*A")`

t (reversed, parentheses swapped)

D	/	(	C	+	B	)	*	A
							▲ c	

out: D C B +



stack

*: precedence of / on top (2) = 2, and we don't pop on equal → **push** *.

Step 10 / 13

```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                    // discard '('
8    } else {                          // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) > prec(c))
11       out.append(stack.pop());
▶      stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

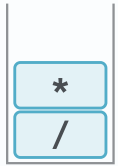
`c = '*'` `out = "DCB+"` `stack = [/, *]`

Code Trace: `postfixForPrefix("D/(C+B)*A")`

t (reversed, parentheses swapped)

D	/	(	C	+	B	)	*	A
								▲ c

out: **D C B + A**



stack

A is an operand → straight to the output.

Step 11 / 13

```
1  for (char c : expr.toCharArray()) {
▶    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                    // discard '('
8    } else {                          // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) > prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
15 while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`c = 'A'` `out = "DCB+A"`

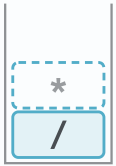
`stack = [/, *]`

Code Trace: `postfixForPrefix("D/(C+B)*A")`

t (reversed, parentheses swapped)

D	/	(	C	+	B	)	*	A

out: **D C B + A ***



stack

End of input: the remaining operator * goes to the output.

Step 12 / 13

```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                    // discard '('
8    } else {                          // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) > prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
▶ while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`out = "DCB+A*"` `stack = [/]`

Code Trace: `postfixForPrefix("D/(C+B)*A")`

t (reversed, parentheses swapped)

D	/	(	C	+	B	)	*	A

out: **D C B + A * /**



stack

End of input: the remaining operator **/** goes to the output. p = **D C B + A * /** → reversed: **/ * A + B C D**

Step 13 / 13

```
1  for (char c : expr.toCharArray()) {
2    if (Character.isLetterOrDigit(c)) out.append(c);
3    else if (c == '(') stack.push(c);
4    else if (c == ')') {
5      while (stack.peek() != '(')
6        out.append(stack.pop());
7      stack.pop();                    // discard '('
8    } else {                          // operator
9      while (!stack.isEmpty() &&
10         prec(stack.peek()) > prec(c))
11       out.append(stack.pop());
12     stack.push(c);
13   }
14 }
▶ while (!stack.isEmpty()) out.append(stack.pop());
```

VARIABLES

`out = "DCB+A*/"` `stack = []`

SUMMARY

Common Mistakes

pop / peek on an empty stack

Call `isEmpty()` first; otherwise you get an exception or `null`, depending on the library. In bracket matching, `"())"` triggers exactly this bug.

Mixing up top and size

In an array stack, `top` is the index of the last item, so `size = top + 1`. The stack is full when `top == capacity - 1`.

Not clearing a popped cell

In an array of objects, skipping `data[top--] = null` keeps the object away from the garbage collector (a memory leak called loitering).

Operand order in postfix

The first pop is the **right** operand and the second the **left**: `5 2 -` = 5 − 2 = 3, not 2 − 5.

Iteration order

A for-each over `ArrayDeque` goes **top to bottom**, but the legacy `Stack` goes **bottom to top**, so their output looks reversed.

Infinite recursion

Without a base case, frames pile up until a `StackOverflowError` occurs (the call stack overflows).

LAB · JAVA CODE, STEP BY STEP

Lab: Bracket Matching and Postfix

Balanced brackets: {{[]}()}[]

Input (only the characters ( ) [ ] { }) {{[]}()}[]

CODE · **step 1 / 38**

```
1 boolean isBalanced(String s) {
2   Deque<Character> stack = new ArrayDeq…
3   for (char c : s.toCharArray()) {
4     if (c == '(' || c == '[' || c == '{…
5       stack.push(c);
6     } else {
7       if (stack.isEmpty()) return false;
8       char open = stack.pop();
9       if (!matches(open, c)) return fal…
10    }
11  }
12  return stack.isEmpty();
13 }
```

s { { [ ] } ( ) } [ ]

stack empty

stack.size() = 0

RETURN VALUE —

isBalanced("{{[]}()}[]") is called: 10 characters will be scanned left to right.

SUMMARY

Summary: Problems Solved with a Stack

Problem	What the stack holds	Time
push / pop / top / isEmpty / size	—	O(1)
Reversing a string	characters	O(n)
Bracket matching	unmatched opening brackets	O(n)
Next greater element	candidate values to the right (decreasing)	O(n)
Stock span	indices of days not yet beaten	O(n)
Towers of Hanoi	subtasks (n, source, aux., destination)	$O(2^n)$
Infix → postfix / prefix	operators and `(`	O(n)
Postfix evaluation	intermediate results	O(n)
Min stack (`getMin`)	the current minimum	O(1)
Back / forward navigation	visited pages (two stacks)	O(1)

The common pattern: when we care about "the most recent item that is still unresolved," we reach for a stack (LIFO).

SUMMARY

Interactive Simulators

Stack Simulator

push, pop, and peek, shown on both an array and a linked list.

Stock Span Simulator

How the stack changes as the price array is processed.

Infix → Postfix Converter

The operator stack and the output, token by token.

Infix → Prefix Converter

The reversal method, step by step.

SUMMARY

Sample Problems

1. Starting from an empty stack, we run push(5), push(3), pop(), push(8), push(1), pop(), pop(), push(4). What do the pop() calls return, and what is left on the stack?
2. Which of `{[()()]}` and `{[(])}` is balanced? In the other one, at which character is the error detected?
3. Convert `(A + B) * (C - D) / E` to postfix and to prefix.
4. What is the value of the postfix expression `5 1 2 + 4 * + 3 -`?
5. Find the spans for the prices [10, 4, 5, 90, 120, 80].
6. Find the next greater element of each item in {2, 5, 3, 1, 4}.

▶ **Show answers**

SUMMARY

Exercises

1. We push 1, 2, 3, 4 in that order, with pops mixed in anywhere. Are the output orders `2 4 3 1` and `4 1 2 3` possible?
2. Two stacks share one array (one grows from the front, the other from the back). When do they overflow?
3. Sort a stack using a single auxiliary stack. What is the complexity?
4. Use a stack to check whether a word is a palindrome.
5. How would you check that HTML tags such as `<b><i>…</i></b>` are properly nested?
6. If a full array grew by **+10** instead of doubling, what would n pushes cost in total?

▶ **Hints / answers**

SUMMARY

Further Reading

- Goodrich, Tamassia, and Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., 2014.
- Lafore, *Data Structures and Algorithms in Java*, 2nd ed., 2002.
- Oracle, Java SE 17 API documentation for `Deque` and `ArrayDeque`.

The End

Chapter 7: Stacks

Next deck: Chapter 8: Queues and Deques · Study questions · Simulators: Stack, Stock Span, Infix → Prefix Converter, Infix → Postfix Converter · Glossary