Data Structures: Stacks

Data Structure Course
DGIST

October 2, 2025

Contents

Introduction to Stacks

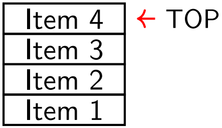
What is a Stack?

Definition

A **stack** is a linear data structure that follows the **Last In, First Out (LIFO)** principle.

Key characteristics:

- Elements are added and removed from the same end (top)
- Only the top element is accessible
- Perfect for managing nested operations and histories
- Natural structure for function calls and recursion



Stack Structure

Core Operations

Essential Stack Operations

Primary Operations

- **Push**: Add element to top
- **Pop**: Remove and return top element
- **Peek/Top**: Return top element without removing
- **Empty**: Check if stack is empty
- **Size**: Get number of elements

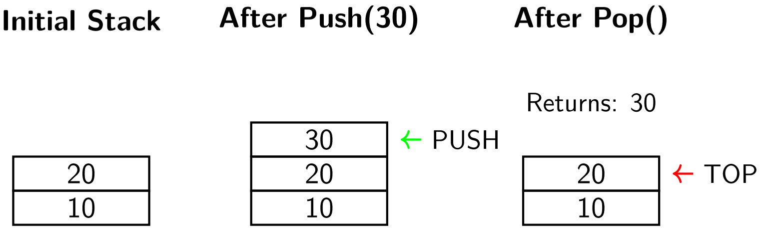
C Example

```c
int s[100];
int top = -1;          // empty

// push
s[++top] = 10;
s[++top] = 20;

// peek (top element)
int topVal = s[top];   // -> 20

// pop
int x = s[top--];      // -> 20

// empty?
int empty = (top == -1);
```

Time Complexity

All operations are **O(1)** - constant time (amortized for push in dynamic arrays)

Stack Operations Visualization



Implementation Approaches

Array-based vs Linked List Implementation

Array-based Stack

Pros:

- Contiguous memory
- Great cache locality
- Simple implementation
- O(1) amortized push

Cons:

- Occasional resize cost
- Capacity management

Linked List-based Stack

Pros:

- No resize cost
- Always O(1) operations
- Dynamic size

Cons:

- Extra pointer memory
- Poor cache locality
- More complex

Linked List Stack Implementation

```python
class Node:
    def __init__(self, val, next=None):
        self.val = val
        self.next = next

class StackLL:
    def __init__(self):
        self.head = None
        self.n = 0

    def push(self, x):
        self.head = Node(x, self.head)
        self.n += 1

    def pop(self):
        if not self.head:
            raise IndexError("pop from empty stack")
```

Applications

Expression Evaluation: Parentheses Matching

Problem

Check if parentheses, brackets, and braces are properly balanced.

Algorithm:

1. Push opening brackets onto stack
2. For closing brackets:
   - Check if stack is empty
   - Check if top matches type
   - Pop if match, return false if not
3. Stack should be empty at end

```python
def valid_brackets(s):
    pairs = {')':'(', ']':'[', '}':'{'}
    st = []
    for ch in s:
        if ch in '([{':
            st.append(ch)
        elif ch in ')]}':
            if not st or st[-1] != pairs[ch]:
                return False
            st.pop()
    return not st

# Examples:
# valid_brackets("([]){}") -> True
# valid_brackets("([)]") -> False
```

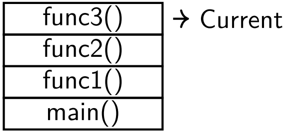
Function Call Stack and Recursion

Call Stack Mechanism

- Each function call creates a **stack frame**
- Frame contains: parameters, local variables, return address
- Recursion uses call stack implicitly
- Deep recursion can cause stack overflow

Converting Recursion to Iteration

Use explicit stack to avoid stack overflow for deep recursion



Call Stack

Undo/Redo Operations

Two-Stack Approach

- **Undo Stack**: stores performed actions
- **Redo Stack**: stores undone actions

Operations:

- **Action**: push to undo, clear redo
- **Undo**: pop from undo, apply inverse, push to redo
- **Redo**: pop from redo, apply, push to undo

Simple undo/redo pattern in pseudocode:

- `do(action)` - execute and save
- `undo()` - reverse last action
- `redo()` - replay undone action

Real-world Examples

Text editors, image editing software, IDEs, web browsers

Complexity Analysis

Time and Space Complexity

Implementation	Push	Pop/Peek	Space
Array-based	O(1) amortized	O(1)	O(n)
Linked List-based	O(1)	O(1)	O(n) + pointer overhead

Array-based Considerations

- Amortized O(1) push due to resize
- Worst-case single push: O(n)
- Better cache performance

Linked List Considerations

- Guaranteed O(1) for all operations
- Extra memory per node
- Dynamic allocation overhead

Summary

Key Takeaways

Stack Fundamentals

- LIFO data structure with top-only access
- Essential operations: push, pop, peek, empty, size
- All operations are O(1) time complexity

Implementation Choices

- Array-based: better cache, amortized O(1)
- Linked list-based: guaranteed O(1), more memory

Important Applications

- Expression evaluation and parentheses matching
- Function call management and recursion
- Undo/redo functionality
- Backtracking algorithms

Thank You!

Questions?