Implement STACK using one QUEUE Implements STACK using two QUEUES Implement QUEUE using one STACK Implement QUEUE using two STACK first give me theoretically detailed answer so that i can understand it properly and make presentation slide

Implementing Stacks and Queues Using Each Other

Table of Contents

1. Implementing a Stack Using One Queue

Theory:

A stack follows LIFO (Last-In-First-Out) while a queue follows FIFO (First-In-First-Out). To simulate stack behavior with a single queue, we need to reverse the order of elements during push operations.

Approach:

- **Push Operation**:
    1. Add the new element to the queue
    2. Rotate the queue by moving all existing elements (except the new one) to the back of the queue
    3. This makes the new element effectively at the front of the queue
- **Pop Operation**: Simply remove the front element from the queue
- **Top Operation**: Return the front element of the queue
- **Empty Check**: Check if the queue is empty

Complexity Analysis:

- Push: O(n) time (since we need to rotate n-1 elements for each push)

- Pop: O(1) time
- Top: O(1) time
- Empty: O(1) time

Visualization:

Initial queue: []
Push 1: [1]
Push 2:

- Add 2: [1, 2]
- Rotate: [2, 1]
  Push 3:
- Add 3: [2, 1, 3]
- Rotate: [3, 2, 1]

2. Implementing a Stack Using Two Queues

Theory:

We can use two queues (q1 and q2) where one queue acts as the main storage and the other helps in reordering elements during push operations.

Approach 1 (Costly Push):

- **Push Operation**:
    1. Add the new element to q2
    2. Move all elements from q1 to q2 (one by one)
    3. Swap the names of q1 and q2
- **Pop Operation**: Simply remove from q1
- **Top Operation**: Return front of q1
- **Empty Check**: Check if q1 is empty

Approach 2 (Costly Pop):

- **Push Operation**: Simply add to q1
- **Pop Operation**:
    1. Move all elements except one from q1 to q2

2. Remove and return the last element from q1

3. Swap the names of q1 and q2

- **Top Operation**: Similar to pop but without removing

- **Empty Check**: Check if q1 is empty

Complexity Analysis (Costly Push):

- Push: O(n) time
- Pop: O(1) time
- Top: O(1) time
- Empty: O(1) time

Visualization (Costly Push):

Initial: q1 = [], q2 = []
Push 1:

- q2 = [1]
- q1 empty, no transfer
  Push 2:
- q2 = [2]
- Move from q1: q2 = [2,1]
- Swap: q1 = [2,1], q2 = []
  Pop: returns 2 (q1 = [1])

3. Implementing a Queue Using One Stack

Theory:

This is more challenging as we need FIFO behavior with LIFO structure. We can use recursion (the call stack) as an implicit second stack.

Approach:

- **Enqueue Operation**: Simply push to the stack

- **Dequeue Operation**:

  1. If stack has one element, pop and return it

  2. Otherwise, pop an item, recursively dequeue, then push the item back

3. Return the result of the recursive call

Complexity Analysis:

- Enqueue: O(1) time
- Dequeue: O(n) time (due to recursive calls)
- Front: Similar to dequeue but without removal
- Empty: O(1) time

Visualization:

Stack: []
Enqueue 1: [1]
Enqueue 2: [1,2]
Enqueue 3: [1,2,3]
Dequeue:

- pop 3, recursive call
- pop 2, recursive call
- pop 1, return 1
- push 2 back
- push 3 back
  Result: returns 1, stack = [2,3]

4. Implementing a Queue Using Two Stacks

Theory:

We can use one stack (s1) for enqueue operations and another (s2) for dequeue operations.

Approach:

- **Enqueue Operation**: Push to s1
- **Dequeue Operation**:
  1. If s2 is empty, transfer all elements from s1 to s2
  2. Pop from s2
- **Front Operation**: Similar to dequeue but without removal
- **Empty Check**: Both stacks must be empty

Complexity Analysis (Amortized):

- Enqueue: O(1) time
- Dequeue: O(1) amortized time (O(n) when s2 is empty and needs refill)
- Front: O(1) amortized time
- Empty: O(1) time

Visualization:

s1 = [], s2 = []
Enqueue 1: s1 = [1]
Enqueue 2: s1 = [1,2]
Dequeue:

- s2 empty, transfer: s1 = [], s2 = [2,1]
- pop from s2: returns 1, s2 = [2]
  Enqueue 3: s1 = [3], s2 = [2]
  Dequeue: pop from s2: returns 2

Summary Table

Implementation	Push/Enqueue	Pop/Dequeue	Space Complexity
Stack with 1 Queue	O(n)	O(1)	O(n)
Stack with 2 Queues	O(n)	O(1)	O(n)
Queue with 1 Stack	O(1)	O(n)	O(n)
Queue with 2 Stacks	O(1)	O(1)*	O(n)

*Amortized complexity

These implementations demonstrate how fundamental data structures can simulate each other's behavior, though often with trade-offs in time complexity for certain operations.

you can give few more examples on visualization with more details

Detailed Visualizations for Stack/Queue Implementations

Let me provide more detailed, step-by-step visualizations for each implementation with multiple operations to clearly show how the data structures transform.

1. Stack Using One Queue

Operations Sequence: Push 1, Push 2, Push 3, Pop, Push 4, Pop, Pop

```
Initial:
Queue = []

Push 1:
1. Add to queue: [1]
2. No rotation needed (first element)
Queue = [1]

Push 2:
1. Add to queue: [1, 2]
2. Rotate by moving 1 to back:
   - Dequeue 1: Queue = [2]
   - Enqueue 1: Queue = [2, 1]

Push 3:
1. Add to queue: [2, 1, 3]
2. Rotate by moving 2,1 to back:
   - Dequeue 2: Queue = [1, 3]
   - Enqueue 2: Queue = [1, 3, 2]
   - Dequeue 1: Queue = [3, 2]
   - Enqueue 1: Queue = [3, 2, 1]

Pop:
Remove front: 3
Queue = [2, 1]

Push 4:
1. Add to queue: [2, 1, 4]
2. Rotate by moving 2,1 to back:
   - Dequeue 2: Queue = [1, 4]
   - Enqueue 2: Queue = [1, 4, 2]
   - Dequeue 1: Queue = [4, 2]
   - Enqueue 1: Queue = [4, 2, 1]
```

```
Pop:
Remove front: 4
Queue = [2, 1]

Pop:
Remove front: 2
Queue = [1]
```

2. Stack Using Two Queues (Costly Push Approach)

Operations Sequence: Push A, Push B, Push C, Pop, Push D, Pop, Pop

text

Copy

Download

```
Initial:
q1 = [], q2 = []

Push A:
1. Add to q2: q2 = [A]
2. q1 is empty, no transfer
3. Swap: q1 = [A], q2 = []

Push B:
1. Add to q2: q2 = [B]
2. Transfer from q1: move A to q2 → q2 = [B, A]
3. Swap: q1 = [B, A], q2 = []

Push C:
1. Add to q2: q2 = [C]
2. Transfer from q1:
   - Move B to q2 → q2 = [C, B]
   - Move A to q2 → q2 = [C, B, A]
3. Swap: q1 = [C, B, A], q2 = []

Pop:
Remove from q1: C
q1 = [B, A]

Push D:
1. Add to q2: q2 = [D]
2. Transfer from q1:
```

```
   - Move B to q2 → q2 = [D, B]
   - Move A to q2 → q2 = [D, B, A]
3. Swap: q1 = [D, B, A], q2 = []

Pop:
Remove from q1: D
q1 = [B, A]

Pop:
Remove from q1: B
q1 = [A]
```

3. Queue Using One Stack (Recursive Approach)

Operations Sequence: Enqueue 10, Enqueue 20, Dequeue, Enqueue 30, Dequeue, Dequeue

text

Copy

Download

```
Initial:
Stack = []

Enqueue 10:
Push 10: [10]

Enqueue 20:
Push 20: [10, 20]

Dequeue:
1. Stack has [10,20]
2. Pop 20, recursive call:
   - Stack now [10]
   - Pop 10, recursive call:
     - Stack empty, return 10
   - Push 10 back: [10]
   - Return 10
3. Push 20 back: [10,20]
4. Return 10 (dequeued)
Final stack: [20]

Enqueue 30:
```

```
Push 30: [20, 30]

Dequeue:
1. Stack has [20,30]
2. Pop 30, recursive call:
   - Stack now [20]
   - Pop 20, recursive call:
     - Stack empty, return 20
   - Push 20 back: [20]
   - Return 20
3. Push 30 back: [20,30]
4. Return 20 (dequeued)
Final stack: [30]

Dequeue:
1. Stack has [30]
2. Pop and return 30
Final stack: []
```

4. Queue Using Two Stacks

Operations Sequence: Enqueue X, Enqueue Y, Dequeue, Enqueue Z, Dequeue, Dequeue

text

Copy
Download

```
Initial:
s1 = [], s2 = []

Enqueue X:
Push to s1: s1 = [X], s2 = []

Enqueue Y:
Push to s1: s1 = [X, Y], s2 = []

Dequeue:
1. s2 is empty, transfer from s1:
   - Pop X from s1: s1 = [Y]
   - Push X to s2: s2 = [X]
   - Pop Y from s1: s1 = []
   - Push Y to s2: s2 = [Y, X]
```

```
2. Pop from s2: Y
Final state: s1 = [], s2 = [X]

Enqueue Z:
Push to s1: s1 = [Z], s2 = [X]

Dequeue:
1. s2 not empty (has X)
2. Pop from s2: X
Final state: s1 = [Z], s2 = []

Dequeue:
1. s2 is empty, transfer from s1:
   - Pop Z from s1: s1 = []
   - Push Z to s2: s2 = [Z]
2. Pop from s2: Z
Final state: s1 = [], s2 = []
```

Key Observations:

1. **Stack with One Queue**:
    - Each push requires O(n) rotations
    - The newest element always ends up at the front of the queue
    - Visualize the rotation as "pulling the rug" to bring new elements to the front
2. **Stack with Two Queues**:
    - The active queue always maintains elements in stack order
    - The secondary queue is only used temporarily during push operations
    - Each push operation effectively reverses the order twice (once when transferring, once when swapping)
3. **Queue with One Stack**:
    - The recursion stack serves as temporary storage
    - Each dequeue requires completely unwinding and rewinding the stack
    - The recursion depth equals the number of elements
4. **Queue with Two Stacks**:
    - s2 maintains elements in queue order (oldest at top)

- s1 is where new elements are added
- Transfer between stacks only happens when s2 is empty
- Amortized cost is better because not every dequeue requires transfer

These visualizations show how the elements move between data structures during each operation, helping understand the underlying transformations that enable emulating one ADT with another.

Non-Recursive Approaches for Implementing Queues Using Stacks

Here are alternative methods to implement a queue using stacks without recursion, along with detailed visualizations:

1. Queue Using Two Stacks (Non-Recursive Approach)

This is the standard approach I mentioned earlier, but let me explain it more thoroughly as an alternative to the recursive method.

Approach:

- **Stack1 (s1)**: Used for enqueue operations
- **Stack2 (s2)**: Used for dequeue operations

Operations:

- **Enqueue**:
    1. Simply push the element onto s1
- **Dequeue**:
    1. If s2 is empty, transfer all elements from s1 to s2 (this reverses the order)
    2. Pop the top element from s2
- **Front**:
    1. Same as dequeue but without removing the element
- **Empty**:
    1. Check if both stacks are empty

Complexity:

- Enqueue: O(1)

- Dequeue: O(1) amortized (O(n) when s2 is empty and needs refilling)
- Front: O(1) amortized
- Empty: O(1)

Detailed Visualization:

text

Copy

Download

```
Initial:
s1 = [], s2 = []

Enqueue A:
s1 = [A], s2 = []

Enqueue B:
s1 = [A, B], s2 = []

Enqueue C:
s1 = [A, B, C], s2 = []

Dequeue:
1. s2 is empty, so transfer from s1 to s2:
   - Pop from s1: C → s1 = [A, B], s2 = [C]
   - Pop from s1: B → s1 = [A], s2 = [B, C]
   - Pop from s1: A → s1 = [], s2 = [A, B, C]
2. Now pop from s2: A
Final state: s1 = [], s2 = [B, C]

Enqueue D:
s1 = [D], s2 = [B, C]

Enqueue E:
s1 = [D, E], s2 = [B, C]

Dequeue:
1. s2 is not empty ([B, C])
2. Pop from s2: B
Final state: s1 = [D, E], s2 = [C]

Dequeue:
```

```
1. s2 is not empty ([C])
2. Pop from s2: C
Final state: s1 = [D, E], s2 = []

Dequeue:
1. s2 is empty, transfer from s1:
   - Pop from s1: E → s1 = [D], s2 = [E]
   - Pop from s1: D → s1 = [], s2 = [D, E]
2. Pop from s2: D
Final state: s1 = [], s2 = [E]
```

2. Queue Using One Stack (Non-Recursive Approach)

This is a more challenging problem, but here's a method using one stack and temporary variables:

Approach:

- **Stack (s)**: Main storage
- **Front variable**: Stores the front element of the queue
- **Count variable**: Tracks number of elements

Operations:

- **Enqueue**:
    1. If stack is empty, set front = new element
    2. Push the element to stack
    3. Increment count
- **Dequeue**:
    1. If stack has only one element, pop and return it
    2. Otherwise:
       a. Pop all elements except the bottom one to a temporary stack
       b. Store and remove the bottom element
       c. Push all elements back from temporary stack
       d. Update front to the new bottom element
- **Front**:
    1. Return the front variable
- **Empty**:
    1. Check if count == 0

Complexity:

- Enqueue: O(1)
- Dequeue: O(n) (but no recursion)
- Front: O(1)
- Empty: O(1)

Detailed Visualization:

text

Copy

Download

```
Initial:
s = [], front = null, count = 0

Enqueue 10:
1. Stack empty, set front = 10
2. Push 10: s = [10]
3. count = 1

Enqueue 20:
1. Stack not empty
2. Push 20: s = [10, 20]
3. count = 2

Enqueue 30:
1. Stack not empty
2. Push 30: s = [10, 20, 30]
3. count = 3

Dequeue:
1. count = 3 > 1
2. Create temp stack
3. Pop until one element left:
   - Pop 30: s = [10,20], temp = [30]
   - Pop 20: s = [10], temp = [20,30]
4. Store bottom element: 10
5. Push elements back:
   - Pop 20 from temp: s = [20,10], temp = [30]
   - Pop 30 from temp: s = [30,20,10], temp = []
```

```
6. Update front = 20 (new bottom)
7. Return 10
Final state: s = [30,20], front = 20, count = 2

Dequeue:
1. count = 2 > 1
2. Create temp stack
3. Pop until one element left:
   - Pop 30: s = [20], temp = [30]
4. Store bottom element: 20
5. Push elements back:
   - Pop 30 from temp: s = [30,20], temp = []
6. Update front = 30
7. Return 20
Final state: s = [30], front = 30, count = 1

Dequeue:
1. count = 1
2. Pop and return 30
Final state: s = [], front = null, count = 0
```

Key Differences from Recursive Approach:

1. **Explicit Temporary Storage**:
    - Uses an explicit temporary stack instead of the call stack
    - More memory efficient as it doesn't use recursion stack frames
2. **Front Tracking**:
    - Maintains a front variable for O(1) front operations
    - Updates the front during dequeue operations
3. **Count Variable**:
    - Helps optimize special cases (like single element dequeues)

Comparison Table:

Method	Enqueue	Dequeue	Space	Notes
Recursive 1-Stack	O(1)	O(n)	O(n)	Uses call stack
Non-Recursive 2-Stack	O(1)	O(1)*	O(n)	Amortized complexity
Non-Recursive 1-Stack	O(1)	O(n)	O(n)	Uses temp stack

*Amortized time complexity

The two-stack approach is generally preferred for actual implementations as it provides better amortized performance for dequeue operations while still maintaining O(1) enqueue operations. The one-stack non-recursive method is mainly of theoretical interest to show it can be done without recursion.

Understanding Recursive Calls

A recursive call is when a function calls itself during its own execution. This is a fundamental concept in computer science used to solve problems by breaking them down into smaller, similar subproblems.

How Recursive Calls Work

1. **Base Case**: The simplest instance of the problem that can be solved directly
2. **Recursive Case**: The function calls itself with a modified version of the original problem

Simple Example: Factorial Calculation

python

Copy

Download

```
def factorial(n):
    if n == 0:  # Base case
        return 1
    else:       # Recursive case
        return n * factorial(n-1)  # Recursive call
```

When you call `factorial(3)`:

1. `factorial(3)` calls `factorial(2)`
2. `factorial(2)` calls `factorial(1)`
3. `factorial(1)` calls `factorial(0)`
4. `factorial(0)` returns 1 (base case)
5. The calls unwind with each returning its result

In the Context of Queue Implementation

When implementing a queue using one stack with recursion:

python

Copy

Download

```
def dequeue():
    if stack.size() == 1:
        return stack.pop()
    else:
        item = stack.pop()          # Remove top item
        result = dequeue()          # Recursive call
        stack.push(item)            # Put item back
        return result
```

What Happens During Execution

For a stack containing [1, 2, 3] (where 1 is at the bottom):

1. First call to `dequeue()`:
    - Pops 3
    - Makes recursive call
2. Second call to `dequeue()`:
    - Pops 2
    - Makes recursive call
3. Third call to `dequeue()`:
    - Stack has just 1 (base case)
    - Returns 1
4. Unwinding begins:
    - Second call pushes 2 back, returns 1
    - First call pushes 3 back, returns 1

Key Characteristics of Recursion

1. **Call Stack**: Each recursive call creates a new stack frame
2. **Memory Usage**: Deep recursion can lead to stack overflow
3. **Problem Solving**: Naturally suited for problems with recursive structure

Visualizing the Recursive Dequeue

text

Copy

Download

```
Initial stack: [1, 2, 3] (3 is top)

Call 1:
- Pops 3
- Stack: [1, 2]
- Calls dequeue()

Call 2:
- Pops 2
- Stack: [1]
- Calls dequeue()

Call 3 (base case):
- Pops 1
- Returns 1

Unwinding:
Call 2:
- Pushes 2 back
- Stack: [2]
- Returns 1

Call 1:
- Pushes 3 back
- Stack: [3, 2]
- Returns 1
```

Why Use Recursion Here?

In the queue implementation:

- The recursion helps us reach the bottom of the stack (oldest element)
- As we unwind, we rebuild the stack with remaining elements
- The call stack effectively serves as our second stack

Potential Issues

1. **Stack Overflow**: For very large queues, too many recursive calls

2. **Performance**: Each dequeue requires O(n) operations

3. **Memory**: Each recursive call consumes additional memory

The non-recursive two-stack approach is generally preferred in practice for these reasons.