Data Structure and Algorithm Analysis and Designs

Stack and Queue

ARUN REUNGSINKONKARN, PH.D.

Stack and Queue

Stack and Queue

A **stack** and a **queue** are both fundamental abstract data structures widely used in computer science to organize data in a way that supports efficient insertion, removal, and retrieval operations. While both are linear structures in which elements are arranged sequentially, they differ primarily in the order in which elements are accessed and removed. Understanding their structures, properties, operations, and common use cases is crucial for effective problem-solving and algorithm design.

Stack

Concept

A stack is a linear data structure that operates on the principle of Last-In, First-Out (LIFO). This means that the most recently inserted element is the first one to be removed. You can think of a stack like a stack of books on a table: you place books on top (push), and the next book you take is always the one on top (pop).

Key Operations

- **Push:** Add (insert) an element to the top of the stack.
- **Pop:** Remove (delete) the topmost element from the stack and return it.
- **Peek (Top):** Retrieve the topmost element without removing it.
- **IsEmpty:** Check whether the stack is empty.
- **IsFull:** (If using a fixed-size stack) Check whether the stack has reached its capacity.

Properties

- **Order of Elements**: The order is strictly enforced by LIFO behavior.
- **Access Complexity**:
  - Pushing and popping are O(1) operations when implemented properly.
  - Accessing arbitrary elements inside a stack is not direct and generally O(n), since you must remove elements until you reach the desired one.
- **Implementation**: Can be implemented using arrays (fixed-size) or linked lists (dynamic size).

Common Use Cases

- **Function Call Management**: Programming languages use a call stack to manage function calls, local variables, and return addresses.
- **Backtracking Algorithms**: Stacks are ideal for scenarios where you need to explore possible options, and revert (“backtrack”) once a dead end is reached, such as in maze solving or depth-first search (DFS).
- **Expression Evaluation and Parsing**: Stacks help evaluate postfix (Reverse Polish Notation) expressions and also assist in checking the correctness of parentheses or bracket sequences.
- **Undo/Redo Operations**: Applications like text editors maintain a history of changes using stacks to provide undo/redo capabilities.

Stack Coding in Python

ARUN REUNGSINKONKARN, PH.D.

Stack Implementation in Python

```python
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, item):
        self.stack.append(item)

    def pop(self):
        if not self.is_empty():
            return self.stack.pop()
        else:
            raise IndexError("Pop from an empty stack")

    def peek(self):
        if not self.is_empty():
            return self.stack[-1]
        else:
            raise IndexError("Peek from an empty stack")

    def is_empty(self):
        return len(self.stack) == 0

# Example usage:
s = Stack()
s.push(10)
s.push(20)
print(s.peek())  # Output: 20
s.pop()
print(s.is_empty())  # Output: False
```

Example 1: Undo/Redo in Text Editors

- **How it works**:
  - Actions (like typing, deleting) are stored in a stack.
  - Undo: The last action (top of the stack) is popped and reversed.
  - Redo: Pushed back into another stack when undone actions are reapplied.

```
undo_stack = []
redo_stack = []

# Simulating actions
undo_stack.append("Type A")
undo_stack.append("Type B")
undo_action = undo_stack.pop()  # Undo last action
redo_stack.append(undo_action)  # Redo can push this back
```

Example 2: Balanced Parentheses

Problem: Check if an expression containing parentheses is balanced.

Example:

- Input: "(a + b) * (c - d)" -> Output: True
- Input: "((a + b)" -> Output: False

```python
def is_balanced(expression):
    """
    Check if the parentheses in the expression are balanced.

    :param expression: A string containing the expression
    :return: True if balanced, False otherwise
    """
    # Stack to store opening brackets
    stack = []

    # Dictionary to match closing brackets to their corresponding opening brackets
    matching_parentheses = {')': '(', '}': '{', ']': '['}

    for char in expression:
        if char in matching_parentheses.values():  # Check for opening brackets
            stack.append(char)
        elif char in matching_parentheses.keys():  # Check for closing brackets
            if not stack or stack[-1] != matching_parentheses[char]:
                return False  # Unmatched closing bracket or stack is empty
            stack.pop()  # Remove matched opening bracket

    return len(stack) == 0  # If stack is empty, parentheses are balanced

# Example usage
expression1 = "(a + b) * (c - d)"
expression2 = "((a + b)"

print(is_balanced(expression1))  # Output: True
print(is_balanced(expression2))  # Output: False
```

Example 3: Reverse a String

Problem: Reverse a string using a stack.
Example:

- Input: "hello" -> Output: "olleh"

```python
def reverse_string(string):
    stack = []
    for char in string:
        stack.append(char)
    reversed_string = ""
    while stack:
        reversed_string += stack.pop()
    return reversed_string

# Test
print(reverse_string("hello"))  # Output: "olleh"
```

Challenge 1: Min Stack

Problem:
Create a stack that supports these actions quickly:

1. Add a new number to the stack.
2. Remove the number from the top of the stack.
3. Get the number on the top of the stack.
4. Get the smallest number in the stack at any time.

```
# Steps:
stack = MinStack()
stack.push(-2)        # Add -2
stack.push(0)         # Add 0
stack.push(-3)        # Add -3
print(stack.get_min())  # Smallest is -3
stack.pop()           # Remove -3
print(stack.top())      # Top is 0
print(stack.get_min())  # Smallest is -2
```

Queue

Concept

A **queue** is a linear data structure that operates on the principle of **First-In, First-Out (FIFO)**. In this scenario, the first element inserted into the queue is the first one to be removed. Think of a queue like a line of people waiting for a service: the first person in line is always the next one to be served, and newcomers join at the end of the line.

Key Operations

- **Enqueue:** Add (insert) an element to the rear (end) of the queue.
- **Dequeue:** Remove (delete) an element from the front (head) of the queue and return it.
- **Front (Peek):** Retrieve the element at the front without removing it.
- **IsEmpty:** Check whether the queue is empty.
- **IsFull:** (If using a fixed-size queue) Check whether the queue is at capacity.

Properties

- **Order of Elements**: Strictly FIFO, ensuring the element that has been waiting the longest is served first.
- **Access Complexity**:
  - Enqueuing and dequeuing are typically O(1) operations when using a proper data structure (like a linked list or circular buffer).
  - Access to arbitrary elements in the middle is not efficient and generally O(n).
- **Implementation**: Commonly implemented using linked lists, circular arrays, or specialized data structures like circular buffers.

Common Use Cases

- **Resource Scheduling**: Queues are integral in operating systems for managing tasks and processes, and in networks for buffering packets.
- **Print Job Management**: Printers often use queues to manage multiple documents waiting to be printed.
- **Breadth-First Search (BFS)**: BFS traversal of graphs and trees uses a queue to track nodes as they are visited in layers.
- **Producer-Consumer Problems**: Queues help coordinate work between producers (which generate data) and consumers (which process data).

Common Use Cases

- **Resource Scheduling**: Queues are integral in operating systems for managing tasks and processes, and in networks for buffering packets.
- **Print Job Management**: Printers often use queues to manage multiple documents waiting to be printed.
- **Breadth-First Search (BFS)**: BFS traversal of graphs and trees uses a queue to track nodes as they are visited in layers.
- **Producer-Consumer Problems**: Queues help coordinate work between producers (which generate data) and consumers (which process data).

Queue Coding in Python

ARUN REUNGSINKONKARN, PH.D.

Queue Implementation in Python

1. Using a List (Basic) Python lists can be used to implement a queue. However, removing items from the front (using pop(0)) can be slow because all elements need to shift.

```python
class QueueUsingList:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)  # Add to the end of the list

    def dequeue(self):
        if not self.is_empty():
            return self.queue.pop(0)  # Remove from the front
        else:
            raise IndexError("Dequeue from empty queue")

    def front(self):
        if not self.is_empty():
            return self.queue[0]  # First element
        else:
            return None

    def is_empty(self):
        return len(self.queue) == 0

# Example usage
queue = QueueUsingList()
queue.enqueue(1)
queue.enqueue(2)
print(queue.front())  # Output: 1
print(queue.dequeue())  # Output: 1
print(queue.is_empty())  # Output: False
```

Queue Implementation in Python (Cont.)

2. Using collections.deque (Recommended) deque is optimized for fast appending and popping from both ends, making it ideal for implementing a queue.

```python
from collections import deque

class QueueUsingDeque:
    def __init__(self):
        self.queue = deque()

    def enqueue(self, item):
        self.queue.append(item)  # Add to the end

    def dequeue(self):
        if not self.is_empty():
            return self.queue.popleft()  # Remove from the front
        else:
            raise IndexError("Dequeue from empty queue")

    def front(self):
        if not self.is_empty():
            return self.queue[0]  # First element
        else:
            return None

    def is_empty(self):
        return len(self.queue) == 0

# Example usage
queue = QueueUsingDeque()
queue.enqueue(1)
queue.enqueue(2)
print(queue.front())  # Output: 1
print(queue.dequeue())  # Output: 1
print(queue.is_empty())  # Output: False
```

Queue Implementation in Python (Cont.)

3. Using `queue.Queue` (Thread-Safe) The `queue.Queue` class from Python's standard library is thread-safe and suitable for multithreading.

```python
from queue import Queue

# Initialize the queue
queue = Queue()

# Enqueue elements
queue.put(1)
queue.put(2)

# Get the front element
print(queue.queue[0])  # Output: 1

# Dequeue elements
print(queue.get())  # Output: 1
print(queue.empty())  # Output: False
```

Example 1: Restaurant Waiting List

- **Use Case**: Customers wait in line for their turn to be seated. The first person in the queue is seated first (FIFO).

```
from collections import deque

# Initialize the queue
waiting_list = deque()

# Adding customers to the queue
waiting_list.append("Customer 1")
waiting_list.append("Customer 2")
waiting_list.append("Customer 3")

# Seat customers
print(waiting_list.popleft())  # Output: Customer 1
print(waiting_list.popleft())  # Output: Customer 2
```

Challenge 2: Hot Potato Game

- **Use Case**: Players sit in a circle and pass a "hot potato" (represented as a name). After a fixed number of passes, the player holding the potato is removed, and the game continues.

```python
from collections import deque

def hot_potato(players, num):
    queue = deque(players)
    while len(queue) > 1:
        for _ in range(num):
            queue.append(queue.popleft())  # Pass the potato
        queue.popleft()  # Remove the player holding the potato
    return queue[0]

# Example usage
players = ["A", "B", "C", "D", "E"]
winner = hot_potato(players, 3)
print(f"The winner is: {winner}")  # Output: The winner is: D
```

Example 2: Hot Potato Game

- **Use Case**: Players sit in a circle and pass a "hot potato" (represented as a name). After a fixed number of passes, the player holding the potato is removed, and the game continues.

```python
from collections import deque

def hot_potato(players, num):
    queue = deque(players)
    while len(queue) > 1:
        for _ in range(num):
            queue.append(queue.popleft())  # Pass the potato
        queue.popleft()  # Remove the player holding the potato
    return queue[0]

# Example usage
players = ["A", "B", "C", "D", "E"]
winner = hot_potato(players, 3)
print(f"The winner is: {winner}")  # Output: The winner is: D
```

Individual Assignment

Treasure Hunt Game Using Queues (30 Points)

ARUN REUNGSINKONKARN, PH.D.