CodInClub

Powered By BridgeLabz

Best Practices for Stacks and Queues

Stacks

1. **Use for Reversible or Nested Problems**:
   Stacks are ideal for problems involving recursion, backtracking, or nested structures (e.g., balanced parentheses, undo functionality).
2. **Optimize Stack Size**:
   Avoid memory overflows by setting a proper size for stacks in fixed-size implementations, or use dynamic structures (like Java's `Stack` class) for scalability.
3. **Avoid Infinite Loops in Recursive Algorithms**:
   Ensure a clear base case in recursive stack operations to prevent stack overflow errors.
4. **Push and Pop Atomically**:
   When dealing with multi-threaded environments, ensure stack operations are atomic to avoid race conditions. Use synchronized stacks like `java.util.concurrent.ConcurrentLinkedDeque` in Java.
5. **Check Stack Underflow and Overflow**:
   Always validate operations to avoid popping an empty stack or pushing into a full stack (if the stack has a fixed size).
6. **Use Collections Framework for Robustness**:
   Instead of implementing stacks from scratch, use robust implementations like `Deque` or `LinkedList` from Java's Collections Framework for better performance and maintainability.
7. **Track the Minimum or Maximum Value**:
   For problems where you frequently need the minimum or maximum element, maintain an auxiliary stack to store these values for O(1) retrieval.

Queues

1. **Use for FIFO (First In, First Out) Problems**:
   Queues are well-suited for sequential processing problems, like task scheduling, breadth-first search (BFS), and producer-consumer scenarios.
2. **Choose the Right Type of Queue**:
   - **Simple Queue**: For basic FIFO needs.
   - **Deque (Double-Ended Queue)**: For flexibility to add/remove from both ends.
   - **Priority Queue**: When elements must be processed based on priority rather than order.

3. **Optimize Memory Usage**:
   When using circular queues, keep track of head and tail pointers efficiently to avoid wasting memory.
4. **Handle Concurrency with Thread-Safe Queues**:
   In multi-threaded environments, use thread-safe implementations like `BlockingQueue` or `ConcurrentLinkedQueue`.
5. **Validate Queue Underflow and Overflow**:
   Ensure proper handling of scenarios where the queue is empty (during dequeue operations) or full (in fixed-size queues).
6. **Lazy Deletion for Priority Queues**:
   When frequent deletions are involved, mark elements as deleted and process cleanup later to avoid immediate restructuring costs.
7. **Avoid Polling Empty Queues**:
   Always check if the queue is empty before dequeue operations to avoid exceptions or errors.

Sample Problems for Stacks and Queues

1. **Implement a Queue Using Stacks**
   - **Problem:** Design a queue using two stacks such that enqueue and dequeue operations are performed efficiently.
   - **Hint:** Use one stack for enqueue and another stack for dequeue. Transfer elements between stacks as needed.

```
import java.util.*;
class QueueUsingStacks {
    Stack<Integer> stack1 = new Stack<>();
    Stack<Integer> stack2 = new Stack<>();

    void enqueue(int x) {
        stack1.push(x);
    }

    int dequeue() {
        if (stack2.isEmpty()) {
            while (!stack1.isEmpty()) {
                stack2.push(stack1.pop());
            }
        }
```

```
        return stack2.isEmpty() ? -1 : stack2.pop();
    }

    public static void main(String[] args) {
        QueueUsingStacks q = new QueueUsingStacks();
        q.enqueue(10); q.enqueue(20); q.enqueue(30);
        System.out.println(q.dequeue());
        System.out.println(q.dequeue());
    }
}
```

2. **Sort a Stack Using Recursion**
    - **Problem:** Given a stack, sort its elements in ascending order using recursion.
    - **Hint:** Pop elements recursively, sort the remaining stack, and insert the popped element back at the correct position.

```
import java.util.*;
class SortStackRecursively {
    static void sortedInsert(Stack<Integer> s, int x) {
        if (s.isEmpty() || x > s.peek()) {
            s.push(x);
            return;
        }
        int temp = s.pop();
        sortedInsert(s, x);
        s.push(temp);
    }

    static void sortStack(Stack<Integer> s) {
        if (!s.isEmpty()) {
            int x = s.pop();
            sortStack(s);
            sortedInsert(s, x);
        }
    }

    public static void main(String[] args) {
```

```
        Stack<Integer> s = new Stack<>();
        s.push(30); s.push(10); s.push(20); s.push(5);
        sortStack(s);
        while (!s.isEmpty()) System.out.println(s.pop());
    }
}
```

3. **Stock Span Problem**
   - **Problem:** For each day in a stock price array, calculate the span (number of consecutive days the price was less than or equal to the current day's price).
   - **Hint:** Use a stack to keep track of indices of prices in descending order.

```
import java.util.*;
class StockSpan {
    public static void calculateSpan(int[] price) {
        Stack<Integer> stack = new Stack<>();
        int[] span = new int[price.length];
        for (int i = 0; i < price.length; i++) {
            while (!stack.isEmpty() && price[stack.peek()] <= price[i])
stack.pop();
            span[i] = (stack.isEmpty()) ? (i + 1) : (i - stack.peek());
            stack.push(i);
        }
        for (int i = 0; i < span.length; i++) System.out.print(span[i] + "
");
    }
    public static void main(String[] args) {
        int[] prices = {100, 80, 60, 70, 60, 75, 85};
        calculateSpan(prices);
    }
}
```

4. **Sliding Window Maximum**
   - **Problem:** Given an array and a window size k, find the maximum element in each sliding window of size k.
   - **Hint:** Use a deque (double-ended queue) to maintain indices of useful elements in each window.

```
import java.util.*;
```

```
class SlidingWindowMax {
    public static void printMax(int[] arr, int k) {
        Deque<Integer> dq = new LinkedList<>();
        for (int i = 0; i < arr.length; i++) {
            while (!dq.isEmpty() && dq.peek() <= i - k) dq.poll();
            while (!dq.isEmpty() && arr[dq.peekLast()] < arr[i])
dq.pollLast();
            dq.offer(i);
            if (i >= k - 1) System.out.print(arr[dq.peek()] + " ");
        }
    }
    public static void main(String[] args) {
        int[] arr = {1,3,-1,-3,5,3,6,7};
        printMax(arr, 3);
    }
}
```

5. **Circular Tour Problem**
   - **Problem:** Given a set of petrol pumps with petrol and distance to the next pump, determine the starting point for completing a circular tour.
   - **Hint:** Use a queue to simulate the tour, keeping track of surplus petrol at each pump.

```
import java.util.*;
class PetrolPump {
    int petrol, distance;
    PetrolPump(int p, int d) { petrol = p; distance = d; }
}
class CircularTour {
    public static int tour(PetrolPump[] arr) {
        int start = 0, deficit = 0, balance = 0;
        for (int i = 0; i < arr.length; i++) {
            balance += arr[i].petrol - arr[i].distance;
            if (balance < 0) {
                start = i + 1;
                deficit += balance;
                balance = 0;
```

```
            }
        }
        return (balance + deficit >= 0) ? start : -1;
    }
    public static void main(String[] args) {
        PetrolPump[] arr = {
            new PetrolPump(6, 4),
            new PetrolPump(3, 6),
            new PetrolPump(7, 3)
        };
        System.out.println(tour(arr));
    }
}
```

Sample Problems for Hash Maps & Hash Functions

1. **Find All Subarrays with Zero Sum**
    - **Problem:** Given an array, find all subarrays whose elements sum up to zero.
    - **Hint:** Use a hash map to store the cumulative sum and its frequency. If a sum repeats, a zero-sum subarray exists.

```
import java.util.*;

public class ZeroSumSubarrays {
    public static void main(String[] args) {
        int[] arr = {6, 3, -1, -3, 4, -2, 2, 4, 6, -12, -7};
        Map<Integer, List<Integer>> map = new HashMap<>();
        int sum = 0;
        map.put(0, new ArrayList<>());
        map.get(0).add(-1);

        for (int i = 0; i < arr.length; i++) {
            sum += arr[i];
            if (map.containsKey(sum)) {
                List<Integer> list = map.get(sum);
                for (int j = 0; j < list.size(); j++) {
                    System.out.println("Subarray: " + (list.get(j) + 1) + "
```

```
to " + i);
                }
                list.add(i);
            } else {
                List<Integer> newList = new ArrayList<>();
                newList.add(i);
                map.put(sum, newList);
            }
        }
    }
}
```

2. **Check for a Pair with Given Sum in an Array**
    - **Problem:** Given an array and a target sum, find if there exists a pair of elements whose sum is equal to the target.
    - **Hint:** Store visited numbers in a hash map and check if `target - current_number` exists in the map.

```
import java.util.*;

public class PairWithSum {
    public static void main(String[] args) {
        int[] arr = {10, 15, 3, 7};
        int target = 17;
        Set<Integer> seen = new HashSet<>();
        for (int i = 0; i < arr.length; i++) {
            if (seen.contains(target - arr[i])) {
                System.out.println("Pair found: " + arr[i] + ", " + (target
- arr[i]));
                return;
            }
            seen.add(arr[i]);
        }
        System.out.println("No pair found");
    }
}
```

3. **Longest Consecutive Sequence**
    - **Problem:** Given an unsorted array, find the length of the longest consecutive elements sequence.
    - **Hint:** Use a hash map to store elements and check for consecutive elements efficiently.

```java
import java.util.*;

public class LongestConsecutiveSeq {
    public static void main(String[] args) {
        int[] nums = {100, 4, 200, 1, 3, 2};
        Set<Integer> set = new HashSet<>();
        for (int i = 0; i < nums.length; i++) set.add(nums[i]);

        int maxLen = 0;
        for (int i = 0; i < nums.length; i++) {
            if (!set.contains(nums[i] - 1)) {
                int curr = nums[i];
                int len = 1;
                while (set.contains(curr + 1)) {
                    curr++;
                    len++;
                }
                if (len > maxLen) maxLen = len;
            }
        }
        System.out.println("Longest sequence length: " + maxLen);
    }
}
```

4. **Implement a Custom Hash Map**
    - **Problem:** Design and implement a basic hash map class with operations for insertion, deletion, and retrieval.
    - **Hint:** Use an array of linked lists to handle collisions using separate chaining.

```java
import java.util.*;
```

CodInClub

Powered By BridgeLabz

```
class Entry {
    int key, value;
    Entry next;
    Entry(int k, int v) { key = k; value = v; }
}

class MyHashMap {
    int size = 10;
    Entry[] table = new Entry[size];

    int hash(int key) { return key % size; }

    void put(int key, int value) {
        int idx = hash(key);
        Entry head = table[idx];
        while (head != null) {
            if (head.key == key) { head.value = value; return; }
            head = head.next;
        }
        Entry newNode = new Entry(key, value);
        newNode.next = table[idx];
        table[idx] = newNode;
    }

    Integer get(int key) {
        int idx = hash(key);
        Entry head = table[idx];
        while (head != null) {
            if (head.key == key) return head.value;
            head = head.next;
        }
        return null;
    }

    void remove(int key) {
        int idx = hash(key);
        Entry head = table[idx], prev = null;
        while (head != null) {
            if (head.key == key) {
                if (prev == null) table[idx] = head.next;
```

```
                else prev.next = head.next;
                return;
            }
            prev = head;
            head = head.next;
        }
    }

    void display() {
        for (int i = 0; i < size; i++) {
            Entry head = table[i];
            System.out.print(i + ": ");
            while (head != null) {
                System.out.print("[" + head.key + "=" + head.value + "] ");
                head = head.next;
            }
            System.out.println();
        }
    }
}

public class CustomHashMap {
    public static void main(String[] args) {
        MyHashMap map = new MyHashMap();
        map.put(1, 10);
        map.put(2, 20);
        map.put(12, 30);
        map.display();
        System.out.println("Get 2: " + map.get(2));
        map.remove(2);
        map.display();
    }
}
```

5. **Two Sum Problem**
    - **Problem:** Given an array and a target sum, find two indices such that their values add up to the target.

- **Hint:** Use a hash map to store the index of each element as you iterate. Check if `target - current_element` exists in the map.

```
import java.util.*;

public class TwoSum {
    public static void main(String[] args) {
        int[] arr = {2, 7, 11, 15};
        int target = 9;
        Map<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < arr.length; i++) {
            int comp = target - arr[i];
            if (map.containsKey(comp)) {
                System.out.println("Indices: " + map.get(comp) + ", " + i);
                return;
            }
            map.put(arr[i], i);
        }
        System.out.println("No valid pair found.");
    }
}
```