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.

```java
import java.util.*;

public class QueueUsingStacks {

    static class Queue<D> {

        Stack<D> stk1 = new Stack<>();
        Stack<D> stk2 = new Stack<>();

        public void add(D data) {
            System.out.println("Adding");
            while (!stk1.isEmpty()) {
                stk2.push(stk1.pop());
            }

            stk1.push(data);

            while (!stk2.isEmpty()) {
                stk1.push(stk2.pop());
            }
        }

        public D remove() {
            return stk1.pop();
        }

    }

    public static void main(String[] args) {

        Queue<Integer> q = new Queue<>();
        q.add(1);
        q.add(2);
        q.add(3);
```

```
        q.add(4);

        System.out.println(q.remove());
        System.out.println(q.remove());
        System.out.println(q.remove());

        q.add(5);
        System.out.println(q.remove());

    }

}
```

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.

```java
import java.util.*;

public class SortStack {

    public static void sort(Stack<Integer> stk) {
        if (stk.isEmpty()) {
            return;
        }
        int ele = stk.pop();
        sort(stk);

        Stack<Integer> dummy = new Stack<>();
        while (!stk.isEmpty() && stk.peek() < ele) {
            dummy.push(stk.pop());
        }
        stk.push(ele);
        while(!dummy.isEmpty()){
            stk.push(dummy.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.*;

public class StockSpan {
    public static void stockSpan(int stock[], int span[]) {
        Stack<Integer> s = new Stack<>();
        s.push(0);
        for (int i = 1; i < stock.length; i++) {
            int currPrice = stock[i];
            while (!s.isEmpty() && currPrice >= stock[s.peek()]) {
                s.pop();
            }
            if (s.isEmpty()) {
                span[i] = i + 1;
            } else {
                span[i] = i - s.peek();
            }

            s.push(i);
        }
    }

    public static void main(String args[]) {
        int stock[] = { 100, 80, 60, 70, 60, 75, 85 };
        int span[] = new int[stock.length];
        span[0] = 1;
        stockSpan(stock, span);
        for (int i = 0; i < span.length; i++) {
            System.out.print(span[i] + " ");
        }
    }
}
```

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 SlidingWindowMaximum {

    public static void main(String[] args) {

        int arr[] = { 3, 6, 7, 1, 0, 6, 7, 9 };
        int k = 4;

        displayMax(arr, k);
    }

    public static void displayMax(int arr[], int k){
        Deque<Integer> dq = new LinkedList<Integer>();

        for (int i = 0; i < arr.length; i++) {
            if (!dq.isEmpty() && (i - dq.peek()) >= k)
                dq.poll();

            while (!dq.isEmpty() && arr[i] > arr[dq.peekLast()])
                dq.pollLast();

            dq.addLast(i);

            if (i >= k - 1)
                System.out.println("Maximum in the window is : " +
arr[dq.peek()]);
        }
    }

}
```

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.*;

public class CircularTour {

    static class Node {
        int petrol;
        int distance;

        Node(int petrol, int distance) {
            this.petrol = petrol;
            this.distance = distance;
        }
    }

    public static int findStartPoint(int petrolPump[] , int distanceCoverage[])
{

        Queue<Integer> q = new LinkedList<>();
        int len = petrolPump.length;
        for (int i = 0; i < len; i++) {
            q.add(i);
        }

        int dis = 0;
        boolean b = true;
        while (!q.isEmpty()) {
            b = true;
            int idx = q.poll();
            int copy = idx;
            for (int i = 0; i < len; i++) {
                dis += petrolPump[idx % len];
                int nextIdx = (idx + 1) % len;
                if (dis < distanceCoverage[nextIdx]) {
                    b = false;
                    break;
                }
                idx++;
```

```
            }
            if (b) {
                return copy;
            }
        }
        return -1;

    }
}
```

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 SubarraySumZero {

    public static int countSubarray(int arr[]) {
        int ans = 0;
        int preSum = 0;
        HashMap<Integer, Integer> map = new HashMap<>();
        map.put(0, 1);
        for (int i = 0; i < arr.length; i++) {
            preSum += arr[i];
            if (map.containsKey(preSum)) {
                ans += map.get(preSum);
                map.put(preSum, map.get(preSum) + 1);
            } else {
                map.put(preSum, 1);
            }
        }

        return ans;
```

```
    }

    public static void main(String[] args) {
        int arr[] = { 1, 2, -1, 3, -2, -3, 4, 5, -2, -7 };
        //            1  3   2  5   3   0  4  9   7   0
        System.out.println(countSubarray(arr));
    }
}
```

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 CheckPair {

    public static boolean checkTwoSum(int arr[], int target) {
        HashMap<Integer, Integer> map = new HashMap<>();
        for (int i = 0; i < arr.length; i++) {
            if (map.containsKey(target - arr[i])) {
                return true;
            } else {
                map.put(arr[i], i);
            }
        }
        return false;
    }

    public static void main(String[] args) {
        int arr[] = { 3, 5, 6, 2, 1 };
        int target = 4;

        System.out.println(checkTwoSum(arr, target));
    }
}
```

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.

```
import java.util.*;

public class LongestConsecutiveSequence {


    public static int longestConsecutive(int[] nums) {
        HashMap<Integer, Boolean> map = new HashMap<>();
        for (int i = 0; i < nums.length; i++) {
            if (map.containsKey(nums[i] - 1)) {
                map.put(nums[i], false);
            } else {
                map.put(nums[i], true);
            }
            if (map.containsKey(nums[i] + 1)) {
                map.put(nums[i] + 1, false);
            }
        }
        int ans = 0;
        for (int key : map.keySet()) {
            if (map.get(key)) {
                int count = 0;
                while (map.containsKey(key)) {
                    count++;
                    key++;
                }
                ans = Math.max(ans, count);
            }
        }
        return ans;
    }
    public static void main(String[] args) {
        // int arr[] = { 1, 9, 3, 10, 4, 20, 2 };
        int arr[] = { 2, 6, 1, 9, 4, 5, 3 };
```

```
        System.out.println(longestConsecutive(arr));
    }

}
```

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.

```
import java.util.*;

public class HashMapImplement {

    static class HashMap<K, V> {

        private class Node {
            K key;
            V value;

            Node(K key, V value) {
                this.key = key;
                this.value = value;
            }
        }

        private int n;
        private int N;
        private LinkedList<Node> bucket[];

        @SuppressWarnings("unchecked")
        public HashMap() {
            this.n = 0;
            this.N = 4;
            this.bucket = new LinkedList[N];

            for (int i = 0; i < N; i++) {
                bucket[i] = new LinkedList<>();
            }
        }
```

```
public void put(K key, V value) {
    int bi = hashFunction(key);

    int di = searchInLL(bi, key);

    if (di == -1) {
        bucket[bi].add(new Node(key, value));
        n++;
    } else {
        Node n = bucket[bi].get(di);
        n.value = value;
    }

    double lambda = (double) n / N;
    if (lambda > 2) {
        rehash();
    }
}

public int hashFunction(K key) {
    int bi = key.hashCode();
    bi = Math.abs(bi);
    return bi % N;
}

public int searchInLL(int bi, K key) {
    int di = 0;
    LinkedList<Node> ll = bucket[bi];
    for (Node n : ll) {
        if (n.key == key) {
            return di;
        }
        di++;
    }
    return -1;
}

@SuppressWarnings("unchecked")
public void rehash() {
    LinkedList<Node> oldBucket[] = bucket;
    bucket = new LinkedList[N * 2];
    N = N * 2;
```

```
        for (int i = 0; i < N; i++) {
            bucket[i] = new LinkedList<>();
        }

        for (LinkedList<Node> ll : oldBucket) {
            for (Node n : ll) {
                put(n.key, n.value);
            }
        }
    }

    public V get(K key) {
        int bi = hashFunction(key);
        int di = searchInLL(bi, key);

        if (di == -1) {
            return null;
        }

        return bucket[bi].get(di).value;
    }

    public V remove(K key) {
        int bi = hashFunction(key);
        int di = searchInLL(bi, key);

        if (di == -1) {
            return null;
        }
        Node nn = bucket[bi].remove(di);
        n--;
        return nn.value;
    }

    public boolean containsKey(K key) {
        int bi = hashFunction(key);
        int di = searchInLL(bi, key);

        return di != -1;
    }

    public ArrayList<K> keySet() {
```

```
            ArrayList<K> list = new ArrayList<>();
            for (LinkedList<Node> ll : bucket) {
                for (Node nn : ll) {
                    list.add(nn.key);
                }
            }

            return list;
        }

    }

}
```

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 int[] twoSum(int arr[], int target) {
        HashMap<Integer, Integer> map = new HashMap<>();
        int idx1 = -1;
        int idx2 = -1;
        for (int i = 0; i < arr.length; i++) {
            if (map.containsKey(target - arr[i])) {
                idx1 = map.get(target - arr[i]);
                idx2 = i;
                break;
            } else {
                map.put(arr[i], i);
            }
        }
        return new int[] { idx1, idx2 };
    }

    public static void main(String[] args) {
        int arr[] = { 3, 5, 6, 2, 1 };
        int target = 4;
        int ans[] = twoSum(arr, target);
        System.out.println("" + ans[0] + " " + ans[1]);
    }
}
```