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.

Sol:

```
import java.util.*;

class QueueUsingStacks {

Stack<Integer> stack1 = new Stack<>();

Stack<Integer> stack2 = new Stack<>();

void enqueue(int value){

stack1.push(value);

}

int dequeue(){
```

```
if(stack2.isEmpty()){

if(stack1.isEmpty())throw new NoSuchElementException("Queue is empty");

while(!stack1.isEmpty())stack2.push(stack1.pop());

}

return stack2.pop();

}

void display(){

System.out.println("Queue contents: "+stack2);

}

}

public class Main {

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

QueueUsingStacks queue = new QueueUsingStacks();

int choice;

do{

choice = sc.nextInt();

switch(choice){

case 1:

queue.enqueue(sc.nextInt());

break;

case 2:

System.out.println("Dequeued: " + queue.dequeue());

break;
```

```
case 3:

queue.display();

break;

}}while(choice != 0);

}

}
```

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.

Sol:

```
import java.util.*;

class StackSorter {

    static void sortStack(Stack<Integer> stack) {

        if (!stack.isEmpty()) {

            int temp = stack.pop();

            sortStack(stack);

            insertSorted(stack, temp);

        }

    }

    static void insertSorted(Stack<Integer> stack, int element) {

        if (stack.isEmpty() || stack.peek() <= element) {

            stack.push(element);

        } else {

            int temp = stack.pop();
```

```
            insertSorted(stack, element);
            stack.push(temp);
        }
    }
    static void displayStack(Stack<Integer> stack) {
        System.out.println("Sorted Stack: " + stack);
    }
}
public class Main {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter stack elements (type 'done' to stop): ");
        while (sc.hasNextInt()) {
            stack.push(sc.nextInt());
        }
        StackSorter.sortStack(stack);
        StackSorter.displayStack(stack);
    }
}
```

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.

Sol:

```
import java.util.*;

class StockSpan {

    static void calculateSpan(int[] prices) {

        int n = prices.length;

        int[] span = new int[n];

        Stack<Integer> stack = new Stack<>();

        for (int i = 0; i < n; i++) {

            while (!stack.isEmpty() && prices[stack.peek()] <= prices[i]) {

                stack.pop();

            }

            span[i] = (stack.isEmpty()) ? i + 1 : i - stack.peek();

            stack.push(i);

        }

        for (int s : span) {

            System.out.print(s + " ");

        }

        System.out.println();

    }

}

public class Main {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        int n = sc.nextInt();

        int[] prices = new int[n];
```

```
        for (int i = 0; i < n; i++) {
            prices[i] = sc.nextInt();
        }
        StockSpan.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.

Sol:

```
import java.util.*;

class SlidingWindowMaximum {

    static void findMaxInWindow(int[] arr, int k) {

        Deque<Integer> deque = new LinkedList<>();

        int n = arr.length;

        for (int i = 0; i < n; i++) {

            while (!deque.isEmpty() && arr[deque.peekLast()] <= arr[i]) {

                deque.pollLast();

            }

            deque.offerLast(i);

            if (deque.peekFirst() <= i - k) {

                deque.pollFirst();

            }
```

```
            if (i >= k - 1) {
                System.out.print(arr[deque.peekFirst()] + " ");
            }
        }
        System.out.println();
    }
}
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int k = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }
        SlidingWindowMaximum.findMaxInWindow(arr, k);
    }
}
```

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.

Sol:

```
class CircularTour {

    static int getStartingPoint(int[] petrol, int[] distance) {

        int start = 0, end = 1, currPetrol = petrol[0] - distance[0], n = petrol.length;

        while (end != start || currPetrol < 0) {

            while (currPetrol < 0 && start != end) {

                currPetrol -= petrol[start] - distance[start];

                start = (start + 1) % n;

            }

            currPetrol += petrol[end] - distance[end];

            end = (end + 1) % n;

        }

        return start;

    }

}

public class Main {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        int n = sc.nextInt();

        int[] petrol = new int[n];

        int[] distance = new int[n];

        for (int i = 0; i < n; i++) {

            petrol[i] = sc.nextInt();

        }

        for (int i = 0; i < n; i++) {
```

```
        distance[i] = sc.nextInt();
      }
      System.out.println(CircularTour.getStartingPoint(petrol, distance));
  }
}
```

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.

Sol:

```
import java.util.*;

class ZeroSumSubarrays {
  static void findSubarrays(int[] arr) {
    Map<Integer, List<Integer>> map = new HashMap<>();
    int sum = 0;
    map.put(0, new ArrayList<>(Arrays.asList(-1)));
    for (int i = 0; i < arr.length; i++) {
      sum += arr[i];
      if (map.containsKey(sum)) {
        for (int index : map.get(sum)) {
          System.out.println("Subarray found from index " + (index + 1) + " to " + i);
        }
```

```
            }
            map.computeIfAbsent(sum, k -> new ArrayList<>()).add(i);
        }
    }
}
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }
        ZeroSumSubarrays.findSubarrays(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.

Sol:

```
import java.util.*;
class PairWithGivenSum {
    static boolean hasPairWithSum(int[] arr, int target) {
```

```
        Set<Integer> set = new HashSet<>();
        for (int num : arr) {
            if (set.contains(target - num)) {
                return true;
            }
            set.add(num);
        }
        return false;
    }
}
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        int target = sc.nextInt();
        int[] arr = new int[n];
        for (int i = 0; i < n; i++) {
            arr[i] = sc.nextInt();
        }
        System.out.println(PairWithGivenSum.hasPairWithSum(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.

Sol:

```
import java.util.*;

class LongestConsecutiveSequence {

    static int longestConsecutive(int[] nums) {

        Set<Integer> set = new HashSet<>();

        for (int num : nums) {

            set.add(num);

        }

        int longestStreak = 0;

        for (int num : set) {

            if (!set.contains(num - 1)) {

                int currentNum = num;

                int currentStreak = 1;

                while (set.contains(currentNum + 1)) {

                    currentNum++;

                    currentStreak++;

                }

                longestStreak = Math.max(longestStreak, currentStreak);

            }

        }

        return longestStreak;

    }
```

```
}

public class Main {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        int n = sc.nextInt();

        int[] nums = new int[n];

        for (int i = 0; i < n; i++) {

            nums[i] = sc.nextInt();

        }

        System.out.println(LongestConsecutiveSequence.longestConsecutive(nums));

    }

}
```

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.

Sol:

```
class CustomHashMap {

    class Node {

        int key, value;

        Node next;

        Node(int key, int value) {

            this.key = key;

            this.value = value;

            this.next = null;
```

```
    }
}
private final int SIZE = 100;
private Node[] table;
public CustomHashMap() {
    table = new Node[SIZE];
}
private int hash(int key) {
    return key % SIZE;
}
public void put(int key, int value) {
    int index = hash(key);
    Node newNode = new Node(key, value);
    if (table[index] == null) {
        table[index] = newNode;
    } else {
        Node current = table[index];
        while (current != null) {
            if (current.key == key) {
                current.value = value;
                return;
            }
            current = current.next;
        }
```

```
        newNode.next = table[index];
        table[index] = newNode;
    }
}
public Integer get(int key) {
    int index = hash(key);
    Node current = table[index];
    while (current != null) {
        if (current.key == key) {
            return current.value;
        }
        current = current.next;
    }
    return null;
}
public void remove(int key) {
    int index = hash(key);
    Node current = table[index];
    Node prev = null;
    while (current != null) {
        if (current.key == key) {
            if (prev == null) {
                table[index] = current.next;
            } else {
```

```
                prev.next = current.next;
            }
            return;
        }
        prev = current;
        current = current.next;
    }
  }
}
public class Main {
    public static void main(String[] args) {
        CustomHashMap map = new CustomHashMap();
        map.put(1, 100);
        map.put(2, 200);
        map.put(3, 300);
        System.out.println(map.get(2));
        map.remove(2);
        System.out.println(map.get(2));
        map.put(1, 150);
        System.out.println(map.get(1));
    }
}
```

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.

Sol:

```
import java.util.HashMap;

public class TwoSum {

    public static int[] twoSum(int[] nums, int target) {

        HashMap<Integer, Integer> map = new HashMap<>();

        for (int i = 0; i < nums.length; i++) {

            int complement = target - nums[i];

            if (map.containsKey(complement)) {

                return new int[]{map.get(complement), i};

            }

            map.put(nums[i], i);

        }

        return new int[]{-1, -1};

    }

    public static void main(String[] args) {

        int[] nums = {2, 7, 11, 15};

        int target = 9;

        int[] result = twoSum(nums, target);

        System.out.println(result[0] + " " + result[1]);

    }

}
```