**Best Practices for Stacks and Queues**

**Stacks**

* **Use for Reversible or Nested Problems:**
  + Stacks are ideal for problems involving recursion, backtracking, or nested structures (e.g., balanced parentheses, undo functionality).
* **Optimize Stack Size:**
  + Avoid memory overflows by setting a proper size for stacks in fixed-size implementations, or use dynamic structures for scalability.
* **Avoid Infinite Loops in Recursive Algorithms:**
  + Ensure a clear base case in recursive stack operations to prevent stack overflow errors.
* **Push and Pop Atomically:**
  + When dealing with multi-threaded environments, ensure stack operations are atomic to avoid race conditions.
* **Check Stack Underflow and Overflow:**
  + Always validate operations to avoid popping an empty stack or pushing into a full stack.
* **Use Collections Framework for Robustness:**
  + Instead of implementing stacks from scratch, use robust implementations like Deque or LinkedList.
* **Track the Minimum or Maximum Value:**
  + Maintain an auxiliary stack to store these values for O(1) retrieval.

**Queues**

* **Use for FIFO (First In, First Out) Problems:**
  + Queues are well-suited for sequential processing problems like task scheduling, BFS, and producer-consumer scenarios.
* **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.
* **Optimize Memory Usage:**
  + Keep track of head and tail pointers efficiently to avoid wasting memory in circular queues.
* **Handle Concurrency with Thread-Safe Queues:**
  + In multi-threaded environments, use thread-safe implementations like BlockingQueue or ConcurrentLinkedQueue.
* **Validate Queue Underflow and Overflow:**
  + Ensure proper handling of empty and full queues.
* **Lazy Deletion for Priority Queues:**
  + Mark elements as deleted and process cleanup later to avoid immediate restructuring costs.
* **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**

**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.

| using System; using System.Collections.Generic;  public class QueueUsingStacks<T> {  private Stack<T> stack1;  private Stack<T> stack2;   public QueueUsingStacks()  {  stack1 = new Stack<T>();  stack2 = new Stack<T>();  }   // Enqueue operation  public void Enqueue(T item)  {  stack1.Push(item);  }   // Dequeue operation  public T Dequeue()  {  if (stack2.Count == 0)  {  // Transfer elements from stack1 to stack2  while (stack1.Count > 0)  {  stack2.Push(stack1.Pop());  }  }   if (stack2.Count == 0)  {  throw new InvalidOperationException("Queue is empty");  }   return stack2.Pop();  }   // Peek operation to get the front element of the queue  public T Peek()  {  if (stack2.Count == 0)  {  // Transfer elements from stack1 to stack2  while (stack1.Count > 0)  {  stack2.Push(stack1.Pop());  }  }   if (stack2.Count == 0)  {  throw new InvalidOperationException("Queue is empty");  }   return stack2.Peek();  }   // Check if the queue is empty  public bool IsEmpty()  {  return stack1.Count == 0 && stack2.Count == 0;  } }  class Program {  static void Main()  {  QueueUsingStacks<int> queue = new QueueUsingStacks<int>();   queue.Enqueue(1);  queue.Enqueue(2);  queue.Enqueue(3);   Console.WriteLine("Dequeue: " + queue.Dequeue());  Console.WriteLine("Peek: " + queue.Peek());   Console.WriteLine("Dequeue: " + queue.Dequeue());  Console.WriteLine("IsEmpty: " + queue.IsEmpty());  Console.ReadKey();  } } |
| --- |

**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.

| using System; using System.Collections.Generic;  public class SortStack {  public static void Sort(Stack<int> stack)  {  if (stack.Count > 0)  {  // Pop the top element  int temp = stack.Pop();   // Sort the remaining stack  Sort(stack);   // Insert the popped element back in sorted order  InsertSorted(stack, temp);  }  }   private static void InsertSorted(Stack<int> stack, int element)  {  // If stack is empty or element is greater than the top element  if (stack.Count == 0 || element > stack.Peek())  {  stack.Push(element);  }  else  {  // Pop the top element  int temp = stack.Pop();   // Insert element into the sorted stack  InsertSorted(stack, element);   // Push the popped element back  stack.Push(temp);  }  }   public static void PrintStack(Stack<int> stack)  {  foreach (int item in stack)  {  Console.Write(item + " ");  }  Console.WriteLine();  } }  class Program {  static void Main()  {  Stack<int> stack = new Stack<int>();   stack.Push(30);  stack.Push(10);  stack.Push(50);  stack.Push(20);  stack.Push(40);   Console.WriteLine("Original Stack:");  SortStack.PrintStack(stack);   SortStack.Sort(stack);   Console.WriteLine("Sorted Stack:");  SortStack.PrintStack(stack);  Console.ReadKey();  } } |
| --- |

**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.

| using System; using System.Collections.Generic; public class StockSpan {  public static int[] CalculateSpan(int[] prices)  {  int n = prices.Length;  int[] span = new int[n];  Stack<int> stack = new Stack<int>();   for (int i = 0; i < n; i++)  {  // Pop elements from the stack  while (stack.Count > 0 && prices[i] >= prices[stack.Peek()])  {  stack.Pop();  }   // Calculate span  span[i] = (stack.Count == 0) ? (i + 1) : (i - stack.Peek());   // Push current index to the stack  stack.Push(i);  }   return span;  }   public static void Main()  {  int[] prices = { 100, 80, 60, 70, 60, 75, 85 };  int[] span = CalculateSpan(prices);   Console.WriteLine("Stock Span:");  for (int i = 0; i < span.Length; i++)  {  Console.Write(span[i] + " ");  Console.ReadKey();  }  } } |
| --- |

**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.

| using System; using System.Collections.Generic;   public class Deque<T> : LinkedList<T> {  public void AddLast(T item)  {  base.AddLast(item);  }   public void RemoveFirst()  {  base.RemoveFirst();  }   public void RemoveLast()  {  base.RemoveLast();  }   public T GetFirst()  {  return this.First.Value;  }   public T GetLast()  {  return this.Last.Value;  } } public class SlidingWindowMaximum {  public static int[] FindMaxInSlidingWindow(int[] nums, int k)  {  if (nums == null || nums.Length == 0 || k <= 0)  {  return new int[0];  }   int n = nums.Length;  int[] result = new int[n - k + 1];  Deque<int> deque = new Deque<int>();   for (int i = 0; i < n; i++)  {  // Remove elements from the back of the deque while the current element is greater  while (deque.Count > 0 && nums[i] >= nums[deque.GetLast()])  {  deque.RemoveLast();  }   // Add the current element's index to the back of the deque  deque.AddLast(i);   // Remove the front element if its index is outside the current window  if (deque.GetFirst() < i - k + 1)  {  deque.RemoveFirst();  }   // The front element of the deque is the max in the current window  if (i >= k - 1)  {  result[i - k + 1] = nums[deque.GetFirst()];  }  }   return result;  }   public static void Main()  {  int[] nums = { 1, 3, -1, -3, 5, 3, 6, 7 };  int k = 3;  int[] maxInWindows = FindMaxInSlidingWindow(nums, k);   Console.WriteLine("Sliding Window Maximum:");  foreach (int max in maxInWindows)  {  Console.Write(max + " ");  Console.ReadKey();  }  } } |
| --- |

**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.

| using System; using System.Collections.Generic;  public class PetrolPump {  public int Petrol;  public int Distance;   public PetrolPump(int petrol, int distance)  {  Petrol = petrol;  Distance = distance;  } }  public class CircularTour {  public static int FindStartingPoint(List<PetrolPump> petrolPumps)  {  int n = petrolPumps.Count;  int start = 0, end = 1;  int currentPetrol = petrolPumps[start].Petrol - petrolPumps[start].Distance;   while (end != start || currentPetrol < 0)  {  // If currentPetrol becomes less than 0, then remove the starting petrol pump from tour  while (currentPetrol < 0 && start != end)  {  currentPetrol -= petrolPumps[start].Petrol - petrolPumps[start].Distance;  start = (start + 1) % n;   // If 0 is the starting point, then no solution exists  if (start == 0)  {  return -1;  }  }   // Add the petrol pump at end to the tour  currentPetrol += petrolPumps[end].Petrol - petrolPumps[end].Distance;  end = (end + 1) % n;  }   return start;  }   public static void Main()  {  List<PetrolPump> petrolPumps = new List<PetrolPump> {  new PetrolPump(4, 6),  new PetrolPump(6, 5),  new PetrolPump(7, 3),  new PetrolPump(4, 5)  };   int startingPoint = FindStartingPoint(petrolPumps);   if (startingPoint == -1)  {  Console.WriteLine("No solution exists.");  }  else  {  Console.WriteLine("The starting point is: " + startingPoint);  Console.ReadKey();  }  } } |
| --- |

**Sample Problems for Hash Maps & Hash Functions**

**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.

| using System; using System.Collections.Generic;  public class ZeroSumSubarrays {  public static List<Tuple<int, int>> FindZeroSumSubarrays(int[] nums)  {  List<Tuple<int, int>> result = new List<Tuple<int, int>>();  Dictionary<int, List<int>> sumMap = new Dictionary<int, List<int>>();  int cumSum = 0;   sumMap[cumSum] = new List<int> { -1 };    for (int i = 0; i < nums.Length; i++)  {  cumSum += nums[i];   if (sumMap.ContainsKey(cumSum))  {  foreach (int index in sumMap[cumSum])  {  result.Add(new Tuple<int, int>(index + 1, i));  }  }   if (!sumMap.ContainsKey(cumSum))  {  sumMap[cumSum] = new List<int>();  }   sumMap[cumSum].Add(i);  }   return result;  }   public static void Main()  {  int[] nums = { 1, 2, -3, 4, -4, 2, -2, 3, -3 };  List<Tuple<int, int>> zeroSumSubarrays = FindZeroSumSubarrays(nums);   Console.WriteLine("Zero Sum Subarrays:");  foreach (var subarray in zeroSumSubarrays)  {  Console.WriteLine("Start: {0}, End: {1}" , subarray.Item1 , subarray.Item2);  }  } } |
| --- |

**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.

| using System; using System.Collections.Generic;  public class PairWithGivenSum {  public static bool FindPairWithGivenSum(int[] nums, int target)  {  HashSet<int> seenNumbers = new HashSet<int>();   for (int i = 0; i < nums.Length; i++)  {  int difference = target - nums[i];   if (seenNumbers.Contains(difference))  {  Console.WriteLine($"Pair found: ({nums[i]}, {difference})");  return true;  }   seenNumbers.Add(nums[i]);  }   Console.WriteLine("No pair found");  return false;  }   public static void Main()  {  int[] nums = { 10, 15, 3, 7 };  int target = 17;  FindPairWithGivenSum(nums, target);  Console.ReadKey();  } } |
| --- |

**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.

| using System; using System.Collections.Generic;  public class LongestConsecutiveSequence {  public static int FindLongestConsecutiveSequence(int[] nums)  {  HashSet<int> numSet = new HashSet<int>(nums);  int longestStreak = 0;   foreach (int num in nums)  {  if (!numSet.Contains(num - 1))  {  int currentNum = num;  int currentStreak = 1;   while (numSet.Contains(currentNum + 1))  {  currentNum += 1;  currentStreak += 1;  }   longestStreak = Math.Max(longestStreak, currentStreak);  }  }   return longestStreak;  }   public static void Main()  {  int[] nums = { 100, 4, 200, 1, 3, 2 };  int longestStreak = FindLongestConsecutiveSequence(nums);  Console.WriteLine("Longest Consecutive Sequence: " + longestStreak);  Console.ReadKey();  } } |
| --- |

**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.

| using System; using System.Collections.Generic;  public class HashMap<TKey, TValue> {  private const int InitialSize = 16;  private LinkedList<KeyValuePair<TKey, TValue>>[] buckets;   public HashMap()  {  buckets = new LinkedList<KeyValuePair<TKey, TValue>>[InitialSize];  for (int i = 0; i < InitialSize; i++)  {  buckets[i] = new LinkedList<KeyValuePair<TKey, TValue>>();  }  }   // Hash function  private int GetBucketIndex(TKey key)  {  int hash = key.GetHashCode();  int bucketIndex = hash % InitialSize;  return Math.Abs(bucketIndex);  }   // Insert operation  public void Insert(TKey key, TValue value)  {  int bucketIndex = GetBucketIndex(key);  LinkedList<KeyValuePair<TKey, TValue>> bucket = buckets[bucketIndex];   foreach (var pair in bucket)  {  if (pair.Key.Equals(key))  {  throw new ArgumentException("Key already exists");  }  }   bucket.AddLast(new KeyValuePair<TKey, TValue>(key, value));  }   // Delete operation  public void Delete(TKey key)  {  int bucketIndex = GetBucketIndex(key);  LinkedList<KeyValuePair<TKey, TValue>> bucket = buckets[bucketIndex];   var node = bucket.First;  while (node != null)  {  if (node.Value.Key.Equals(key))  {  bucket.Remove(node);  return;  }  node = node.Next;  }   throw new KeyNotFoundException("Key not found");  }   // Retrieve operation  public TValue Retrieve(TKey key)  {  int bucketIndex = GetBucketIndex(key);  LinkedList<KeyValuePair<TKey, TValue>> bucket = buckets[bucketIndex];   foreach (var pair in bucket)  {  if (pair.Key.Equals(key))  {  return pair.Value;  }  }   throw new KeyNotFoundException("Key not found");  } }  class Program  {  static void Main()  {  HashMap<string, int> hashMap = new HashMap<string, int>();  hashMap.Insert("apple", 1);  hashMap.Insert("banana", 2);   Console.WriteLine("Value for 'apple': " + hashMap.Retrieve("apple"));  Console.WriteLine("Value for 'banana': " + hashMap.Retrieve("banana"));   hashMap.Delete("apple");   try  {  Console.WriteLine("Value for 'apple': " + hashMap.Retrieve("apple"));  }  catch (KeyNotFoundException e)  {  Console.WriteLine(e.Message);  Console.ReadKey();  }  } } |
| --- |

**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.

| using System; using System.Collections.Generic;  public class TwoSum {  public static int[] FindTwoSum(int[] nums, int target)  {  Dictionary<int, int> numMap = new Dictionary<int, int>();   for (int i = 0; i < nums.Length; i++)  {  int difference = target - nums[i];    if (numMap.ContainsKey(difference))  {  return new int[] { numMap[difference], i };  }   if (!numMap.ContainsKey(nums[i]))  {  numMap[nums[i]] = i;  }  }  return new int[] {};  }   public static void Main()  {  int[] nums = { 2, 7, 11, 15 };  int target = 9;  int[] result = FindTwoSum(nums, target);  Console.WriteLine("Indices: {0}, {1}", result[0] , result[1]);  } } |
| --- |