PW SKILLS

Lesson Plan

Queue - 2

Java

PW SKILLS

Q. Reverse the first K elements of a Queue

Given an integer k and a queue of integers, The task is to reverse the order of the first k elements of the queue, leaving the other elements in the same relative order.

Code:

```
class Solution {
    public void reverseQueueFirstKElements(int k, Queue<Integer> queue) {
        if (queue.isEmpty() || k > queue.size() || k <= 0)
            return;

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

        // Push the first K elements from the queue into a stack
        for (int i = 0; i < k; i++) {
            stack.push(queue.poll());
        }

        // Enqueue the contents of the stack to the back of the queue
        while (!stack.isEmpty()) {
            queue.add(stack.pop());
        }

        // Remove the remaining elements and enqueue them at the end of the queue
        for (int i = 0; i < queue.size() - k; i++) {
            queue.add(queue.poll());
        }
    }
}
```

Time Complexity: O(N + k), Where 'n' is the total number of elements in the queue and 'k' is the number of elements to be reversed. This is because firstly the whole queue is emptied into the stack and after that first 'k' elements are emptied and enqueued in the same way.

Auxiliary Space: O(k) where k is no of elements to be reversed since stack is being used to store values for the purpose of reversing.

Explanation:

The idea is to use an auxiliary stack. Store the first k elements of the queue in a stack and pop it from the queue, then push it back to the queue and perform pop operation for n-k times and again push the popped element.

Follow the below steps to implement the idea:

Create an empty stack.
One by one dequeue first K items from given queue and push the dequeued items to stack.
Enqueue the contents of stack at the back of the queue
Dequeue (size-k) elements from the front and enqueue them one by one to the same queue.

Q. Number of Students Unable to eat Lunch [LC - 1700]

The school cafeteria offers circular and square sandwiches at lunch break, referred to by numbers 0 and 1 respectively. All students stand in a queue. Each student either prefers square or circular sandwiches.

The number of sandwiches in the cafeteria is equal to the number of students. The sandwiches are placed in a stack. At each step:

If the student at the front of the queue prefers the sandwich on the top of the stack, they will take it and leave the queue.
Otherwise, they will leave it and go to the queue's end.
This continues until none of the queue students want to take the top sandwich and are thus unable to eat.

You are given two integer arrays students and sandwiches where sandwiches[i] is the type of the ith sandwich in the stack (i = 0 is the top of the stack) and students[j] is the preference of the jth student in the initial queue (j = 0 is the front of the queue). Return the number of students that are unable to eat.

Example 1:

Input: students = [1,1,0,0], sandwiches = [0,1,0,1]
Output: 0

Explanation:

- Front student leaves the top sandwich and returns to the end of the line making students = [1,0,0,1].
- Front student leaves the top sandwich and returns to the end of the line making students = [0,0,1,1].
- Front student takes the top sandwich and leaves the line making students = [0,1,1] and sandwiches = [1,0,1].
- Front student leaves the top sandwich and returns to the end of the line making students = [1,1,0].
- Front student takes the top sandwich and leaves the line making students = [1,0] and sandwiches = [0,1].
- Front student leaves the top sandwich and returns to the end of the line making students = [0,1].
- Front student takes the top sandwich and leaves the line making students = [1] and sandwiches = [1].
- Front student takes the top sandwich and leaves the line making students = [] and sandwiches = [].

Hence all students are able to eat.

Example 2:

Input: students = [1,1,1,0,0,1], sandwiches = [1,0,0,0,1,1]
Output: 3

Code:

```
import java.util.LinkedList;
import java.util.Queue;

class Solution {
    public int countStudents(int[] students, int[] sandwiches) {
        int size = students.length;
        Queue<Integer> studentChoice = new LinkedList<>();

        for (int i = 0; i < size; ++i) {
            studentChoice.add(students[i]);
        }

        int rotations = 0, i = 0;
        while (!studentChoice.isEmpty() && rotations < studentChoice.size()) {
            if (studentChoice.peek() == sandwiches[i]) {
                studentChoice.poll();
                i++;
                rotations = 0;
            } else {
                int choice = studentChoice.poll();
                studentChoice.add(choice);
                rotations++;
            }
        }
        return studentChoice.size();
    }
}
```

Time: O(N), Iterating through students and queue operations.
Space: O(N), Queue space for storing student choices.

Explanation: Sure, here's a more concise breakdown of the approach:

1. Use a queue to store the students' sandwich choices.
2. Initialize a counter rotations to track front elements being moved.
3. Iterate through the queue until all rotations are completed or until the queue is empty.
4. If the front student's choice matches the sandwich at index i, remove them from the queue, increment i, and reset rotations to 0.
5. If there's no match, move the student to the back of the queue, increment rotations.
6. Finally, return the size of the queue.

This approach ensures students receive their preferred sandwiches and tracks the number of rotations in the queue until either all students are served or until the rotations match the queue's size.

Q. Implement Queue using Stacks [Leetcode - 232]

Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (push, peek, pop, and empty).

Implement the MyQueue class:

void push(int x) Pushes element x to the back of the queue.
int pop() Removes the element from the front of the queue and returns it.
int peek() Returns the element at the front of the queue.
boolean empty() Returns true if the queue is empty, false otherwise.

Notes:

You must use only standard operations of a stack, which means only push to top, peek/pop from top, size, and is empty operations are valid.
Depending on your language, the stack may not be supported natively. You may simulate a stack using a list or deque (double-ended queue) as long as you use only a stack's standard operations.

Example 1:

Input:
["MyQueue", "push", "push", "peek", "pop", "empty"]
[[], [1], [2], [], [], []]

Output:
[null, null, null, 1, 1, false]

Explanation:

MyQueue myQueue = new MyQueue();
myQueue.push(1); // queue is: [1]
myQueue.push(2); // queue is: [1, 2] (leftmost is front of the queue)
myQueue.peek(); // return 1
myQueue.pop(); // return 1, queue is [2]
myQueue.empty(); // return false

Code:

```
class MyQueue {
    private Stack<Integer> inStack;
    private Stack<Integer> outStack;

    public MyQueue() {
        inStack = new Stack<>();
        outStack = new Stack<>();
    }

    // Push element x to the back of the queue...
    public void push(int x) {
        inStack.push(x);
    }
```

```
    // Remove the element from the front of the queue and returns it...
    public int pop() {
        peek();
        int val = outStack.pop();
        return val;
    }

    // Get the front element...
    public int peek() {
        if (outStack.isEmpty()) {
            while (!inStack.isEmpty()) {
                outStack.push(inStack.pop());
            }
        }
        return outStack.peek();
    }

    // Return whether the queue is empty...
    public boolean empty() {
        return inStack.isEmpty() && outStack.isEmpty();
    }
}
```

Explanation: implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (push, peek, pop, and empty).

Implement the MyQueue class:

void push(int x) Pushes element x to the back of the queue.
int pop() Removes the element from the front of the queue and returns it.
int peek() Returns the element at the front of the queue.
boolean empty() Returns true if the queue is empty, false otherwise.

Time: Peek is O(n), rest are are O(1)
Space: O(n), to store the elements

Q. Given an array and a positive integer k, find the first negative integer for each window(contiguous subarray) of size k. If a window does not contain a negative integer, then print 0 for that window.

Examples:

Input : arr[] = {-8, 2, 3, -6, 10}, k = 2
Output : -8 0 -6 -6

First negative integer for each window of size k
{-8, 2} = -8
{2, 3} = 0 (does not contain a negative integer)
{3, -6} = -6
{-6, 10} = -6

Input : arr[] = {12, -1, -7, 8, -15, 30, 16, 28} , k = 3
Output : -1 -1 -7 -15 -15 0

Code: static void printFirstNegativeInteger(int[] arr, int n, int k) {
Deque<Integer> Di = new LinkedList<>(); // Double Ended Queue to store indexes of negative integers in the current window

```
// Process the first k elements of the array
        for (int i = 0; i < k; i++) {
            // Add the current element's index to the rear of Di if it's negative
            if (arr[i] < 0)
                Di.addLast(i);
        }

        // Process the remaining elements from arr[k] to arr[n-1]
        for (int i = k; i < n; i++) {
            // If Di isn't empty, print the first negative integer of the previous window
            if (!Di.isEmpty())
                System.out.print(arr[Di.peekFirst()] + " ");
            else
                System.out.print("0 "); // If no negative integer exists in the window

            // Remove elements out of the current window from the front of the queue
            while (!Di.isEmpty() && Di.peekFirst() < (i - k + 1))
                Di.pollFirst();

            // Add the index of the current element to the rear of Di if it's negative
            if (arr[i] < 0)
                Di.addLast(i);
        }

        // Print the first negative integer of the last window
        if (!Di.isEmpty())
            System.out.print(arr[Di.peekFirst()] + " ");
        else
            System.out.print("0 ");
    }
```

Time Complexity: O(n), to traverse the whole array

Auxiliary Space: O(k), for the deque

Explanation: We create a Dequeue, Di of capacity k, that stores only useful elements of the current window of k elements. An element is useful if it is in the current window and it is a negative integer. We process all array elements one by one and maintain Di to contain useful elements of current window and these useful elements are all negative integers. For a particular window, if Di is not empty then the element at front of the Di is the first negative integer for that window, else that window does not contain a negative integer.

Q. Sliding Window Maximum [Leetcode - 239]

You are given an array of integers nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.

Return the max sliding window.

Example 1:

Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [3,3,5,5,6,7]

Explanation:

```
Window position                Max
---------------               -----
[1  3  -1] -3  5  3  6  7      3
 1 [3  -1  -3] 5  3  6  7      3
 1  3 [-1  -3  5] 3  6  7      5
 1  3  -1 [-3  5  3] 6  7      5
 1  3  -1  -3 [5  3  6] 7      6
 1  3  -1  -3  5 [3  6  7]     7
```

Example 2:

Input: nums = [1], k = 1
Output: [1]

Code:

```
class Solution {
public:
   class Solution {
    public List<Integer> maxSlidingWindow(int[] nums, int k) {
        int n = nums.length;
        Deque<Integer> dq = new ArrayDeque<>();
        List<Integer> ans = new ArrayList<>();

        for (int i = 0; i < n; i++) {
            while (!dq.isEmpty() && dq.peekFirst() ⩽ i - k)
                dq.pollFirst();

            while (!dq.isEmpty() && nums[dq.peekLast()] ⩽ nums[i])
                dq.pollLast();

            dq.offerLast(i);

            if (i ⩾ k - 1)
                ans.add(nums[dq.peekFirst()]);
        }
        return ans;
    }
}
```

Explanation:

The algorithm uses a deque data structure to store indices of elements in the sliding window. The deque is maintained in such a way that the front element always represents the maximum element in the current window. The algorithm iterates through the input array and updates the deque based on the sliding window's size. At each step, it checks if the front element is within the window's range and removes it if necessary. It also removes any elements from the back of the deque that are smaller than or equal to the current element. Finally, it adds the maximum element from the deque to the answer vector when the window size is reached.

Time complexity: O(n), to traverse the array

Space complexity: O(k), queue will has at max k element at any time

Q. Dota2 Senate [Leetcode - 649]

The Dota2 senate consists of senators coming from two parties. Now the Senate wants to decide on a change in the Dota2 game. The voting for this change is a round-based procedure. In each round, each senator can exercise one of the two rights:

Ban one senator's right: A senator can make another senator lose all his rights in this and all the following rounds.
Announce the victory: If this senator found the senators who still have rights to vote are all from the same party, he can announce the victory and decide on the change in the game.
Given a string senate representing each senator's party belonging. The character 'R' and 'D' represent the Radiant party and the Dire party. Then if there are n senators, the size of the given string will be n.

The round-based procedure starts from the first senator to the last senator in the given order. This procedure will last until the end of voting. All the senators who have lost their rights will be skipped during the procedure.

Suppose every senator is smart enough and will play the best strategy for his own party. Predict which party will finally announce the victory and change the Dota2 game. The output should be "Radiant" or "Dire".

Example 1:

Input: senate = "RD"
Output: "Radiant"

Explanation:
The first senator comes from Radiant and he can just ban the next senator's right in round 1.
And the second senator can't exercise any rights anymore since his right has been banned.
And in round 2, the first senator can just announce the victory since he is the only guy in the senate who can vote.

Example 2:

Input: senate = "RDD"
Output: "Dire"

Explanation:
The first senator comes from Radiant and he can just ban the next senator's right in round 1.
And the second senator can't exercise any rights anymore since his right has been banned.
And the third senator comes from Dire and he can ban the first senator's right in round 1.
And in round 2, the third senator can just announce the victory since he is the only guy in the senate who can vote.

Code:

```
class Solution {
    public String predictPartyVictory(String senate) {
        Queue<Integer> q1 = new LinkedList<>();
        Queue<Integer> q2 = new LinkedList<>();
        int n = senate.length();

        for (int i = 0; i < n; i++) {
            if (senate.charAt(i) == 'R') {
                q1.offer(i);
            } else {
                q2.offer(i);
            }
        }

        while (!q1.isEmpty() && !q2.isEmpty()) {
            int rIndex = q1.poll();
            int dIndex = q2.poll();

            if (rIndex < dIndex) {
                q1.offer(rIndex + n);
            } else {
                q2.offer(dIndex + n);
            }
        }

        return (q1.size() > q2.size()) ? "Radiant" : "Dire";
    }
}
```

Time: O(n), The code iterates through the senate string once, where n is the length of the string. It uses queues to process characters and performs operations based on the senate length, resulting in linear time complexity.

Space: O(n), Two queues (q1 and q2) are used to store the indexes of characters in the senate string. The space used by these queues depends on the length of the senate string, resulting in linear space complexity.

Explanation: this is obliviously a greedy algorithm problem. Each Senate R must ban its next closest Senate D who is from another party, or else D will ban its next Senate from R's party.
The idea is to use two queues to save the index of each senate from R's and D's parties, respectively. During each round, we delete the banned senate's index; and the remaining senate's index with n(the length of the input string senate), then move it to the back of its respective queue.

Q. Reorder Queue (Interleave 1st Half with 2nd Half)
[Do this by using one Stack only]

Given a queue of integers of even length, rearrange the elements by interleaving the first half of the queue with the second half of the queue.

Examples:

Input : 1 2 3 4
Output : 1 3 2 4

Input : 11 12 13 14 15 16 17 18 19 20
Output : 11 16 12 17 13 18 14 19 15 20

Code:

```
class Solution {
    public void interLeaveQueue(Queue<Integer> q) {
        if (q.size() % 2 != 0)
            System.out.println("Input even number of integers.");

        Stack<Integer> s = new Stack<>();
        int halfSize = q.size() / 2;

        // Push the first half elements of the queue into the stack
        for (int i = 0; i < halfSize; i++) {
            s.push(q.poll());
        }

        // Enqueue the stack elements back into the queue
        while (!s.isEmpty()) {
            q.offer(s.pop());
        }

        // Dequeue and enqueue back the first half elements of the queue
        for (int i = 0; i < halfSize; i++) {
            q.offer(q.poll());
        }

        // Again, push the first half elements of the queue into the stack
        for (int i = 0; i < halfSize; i++) {
            s.push(q.poll());
        }

        // Interleave the elements of the queue and the stack
        while (!s.isEmpty()) {
            q.offer(s.pop());
            q.offer(q.poll());
        }
    }
}
```

Time complexity: O(n), traversing the queue
Auxiliary Space: O(n), for the new stack

Explanation: Following are the steps to solve the problem:
Push the first half elements of the queue to stack.
Enqueue back the stack elements.
Dequeue the first half of elements of the queue and enqueue them back.
Again push the first half elements into the stack.
Interleave the elements of queue and stack.

Q. Reveal Cards in Increasing Order [Leetcode - 950]

You are given an integer array deck. There is a deck of cards where every card has a unique integer. The integer on the ith card is deck[i].

You can order the deck in any order you want. Initially, all the cards start face down (unrevealed) in one deck.

You will do the following steps repeatedly until all cards are revealed:

Take the top card of the deck, reveal it, and take it out of the deck.
If there are still cards in the deck then put the next top card of the deck at the bottom of the deck.
If there are still unrevealed cards, go back to step 1. Otherwise, stop.
Return an ordering of the deck that would reveal the cards in increasing order.

Note that the first entry in the answer is considered to be the top of the deck.

Example 1:

Input: deck = [17,13,11,2,3,5,7]
Output: [2,13,3,11,5,17,7]

Explanation:

- We get the deck in the order [17,13,11,2,3,5,7] (this order does not matter), and reorder it.
- After reordering, the deck starts as [2,13,3,11,5,17,7], where 2 is the top of the deck.
- We reveal 2, and move 13 to the bottom. The deck is now [3,11,5,17,7,13].
- We reveal 3, and move 11 to the bottom. The deck is now [5,17,7,13,11].
- We reveal 5, and move 17 to the bottom. The deck is now [7,13,11,17].
- We reveal 7, and move 13 to the bottom. The deck is now [11,17,13].
- We reveal 11, and move 17 to the bottom. The deck is now [13,17].
- We reveal 13, and move 17 to the bottom. The deck is now [17].
- We reveal 17.

Since all the cards revealed are in increasing order, the answer is correct.

Example 2:

Input: deck = [1,1000]
Output: [1,1000]

Code:

```
class Solution {
    public int[] deckRevealedIncreasing(int[] deck) {
        Arrays.sort(deck);
        Deque<Integer> d = new ArrayDeque<>();
        d.offer(deck[deck.length - 1]);

        for (int i = deck.length - 2; i >= 0; i--) {
            d.offerFirst(d.pollLast());
            d.offerFirst(deck[i]);
        }

        int[] res = new int[deck.length];
        int index = 0;
        for (int val : d) {
            res[index++] = val;
        }

        return res;
    }
}
```

Time complexity: O(NlogN), O(NlogN) to sort, O(N) to construct using deque or queue.
Space: O(N), for the deque

Explanation: We simulate the reversed process. Initial an empty list or deque or queue, each time rotate the last element to the first, and append the next biggest number to the left.

PW SKILLS

THANK YOU !

PW SKILLS