Lesson:

Queue in JAVA

Pre Requisites:

- Arrays
- Basic java syntax

List of concepts involved :

- Introduction to queue
- Types of queue
- Queue class in JAVA
- Implementation of various functions of a queue
- Implement queue using stacks
- Implement stack using queues
- Introduction to deque
- Sliding window
- Problem based on the concept of sliding window

What is Queue?

A queue is a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.
For example : standing in a queue for cash withdrawal, the person who came first will be withdrawing the money first.

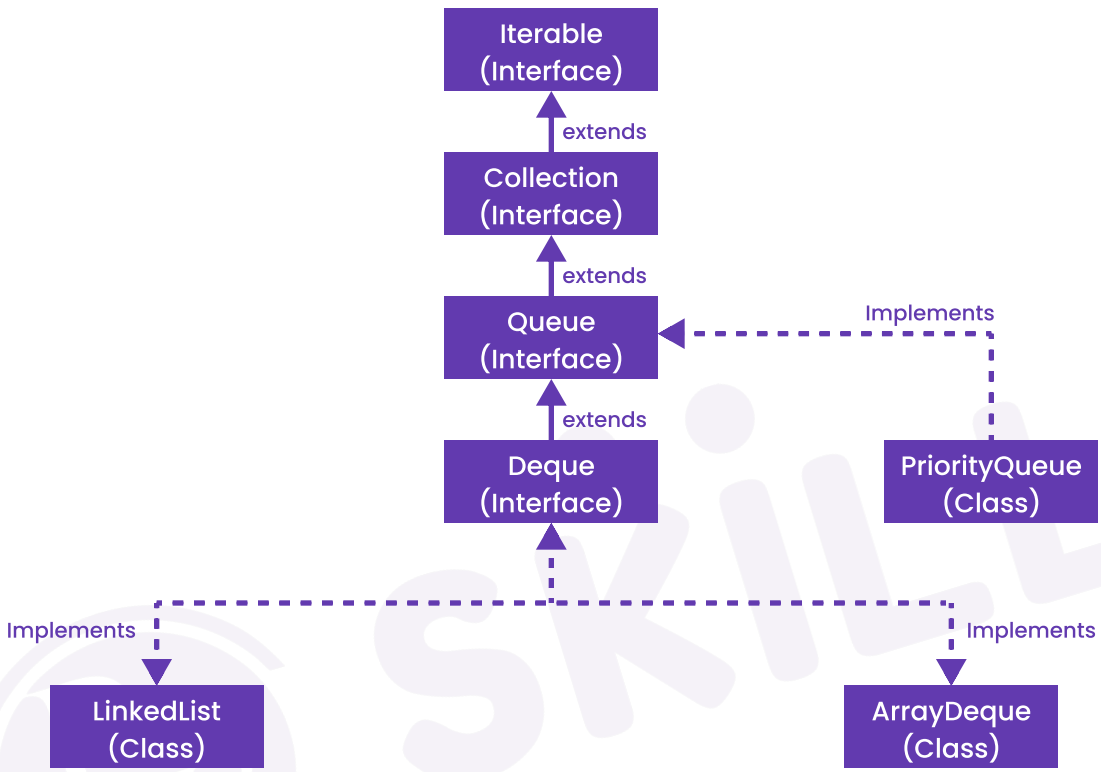
Terminologies associated with a queue :

- Front : this denotes the index from where remove/delete operation will take place.
- Rear : this denotes the index from where add/insert operation will take place.
- Add: adding an element to a queue is known as an add operation. Add operation is not possible if the queue is full.
- Remove : removing an element from a queue is known as remove operation and is not possible if the queue is empty.
- front(): viewing the element present at the front index in a queue is returned by front() function. Its functionality is similar to pop, the only difference is that in pop operation we remove the front indexed element but in front(), we just return the front indexed element's value and we do not remove it. This operation is also not feasible if the queue is empty.

Types of queues :

- **Simple queue** : It is a normal queue that is solely based upon the FIFO principle.

- **Circular Queue :** It is similar to the simple queue. The only difference between a circular queue and a simple queue is the way it is implemented.
- **Double Ended Queue (Deque) :** It is the type of queue where an element can be inserted or deleted from both the front and the rear.
- **Priority Queue :** In this type of queue, the elements are assigned a priority value and depending upon that value, each element is given a position in the queue. The order in which the elements are placed may be in increasing or decreasing order of the priority value depending upon how we want to use it.



Queue class in java :

The Queue interface is present in java.util package and extends the Collection interface is used to hold the elements about to be processed in FIFO(First In First Out) order. It is an ordered list of objects with its use limited to inserting elements at the end of the list and deleting elements from the start of the list, (i.e.), it follows the FIFO or the First-In-First-Out principle.
Being an interface the queue needs a concrete class for the declaration and the most common classes are the PriorityQueue and LinkedList in Java.

Syntax to define a queue :

- Queue<data_type> queue = new PriorityQueue<> ();
- Queue<data_type> queue = new LinkedList<> ();

Q1. Write a program to implement the various functions of a queue.

Solution :

Code : LP_Code1.java

Output :

```
Elements of queue [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
removed element-0
[1, 2, 3, 4, 5, 6, 7, 8, 9]
head of queue-1
Size of queue-9
```

Q2. Write a program to implement the functionalities of a stack using the methods of queue only.
Solution :

Code : LP_Code2.java

Output :

```
current size: 6
The peek element is :
6
The peek element is :
5
The peek element is :
4
current size: 4
```

Approach :

- The idea behind this approach is to make one queue and push the first element in it.
- After the first element, we push the next element and then push the first element again and finally pop the first element.
- So, according to the FIFO rule of the queue, the second element that was inserted will be at the front and then the first element as it was pushed again later and its first copy popped out.
- So, this acts as a Stack and we do this at every step i.e. from the initial element to the second last element, and the last element will be the one that we are inserting and since we will be pushing the initial elements after pushing the last element, our last element becomes the first element.

Time Complexity:

- Push operation: O(N)
- Pop operation: O(1)

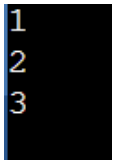
Space complexity: O(N) since 1 queue is used.

Q3. Write a program to implement the functionalities of a queue using the methods of stack only.

Solution :

Code : LP_Code3.java

Output :



Approach :

- Method 1 (By making enQueue/add operation costly): This method makes sure that the oldest entered element is always at the top of stack 1, so that deQueue operation just pops from stack1. To put the element at top of stack1, stack2 is used.

- enQueue(q, x):
- While stack1 is not empty, push everything from stack1 to stack2.
- Push x to stack1 (assuming size of stacks is unlimited).
- Push everything back to stack1.
- Here time complexity will be O(n)

- deQueue(q):
- If stack1 is empty then error
- Pop an item from stack1 and return it.

Time Complexity:

- Push operation: O(N).
- In the worst case we have an empty whole of stack 1 into stack 2.
- Pop operation: O(1).
- Same as pop operation in stack.

Space Complexity : O(N).
Use of stack for storing values.

Method 2 (By making deQueue operation costly):

- In this method, in en-queue operation, the new element is entered at the top of stack1. In de-queue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

- enQueue(q, x)
  1) Push x to stack1 (assuming size of stacks is unlimited).
  Here time complexity will be O(1)
- deQueue(q)
  1) If both stacks are empty then error.
  2) If stack2 is empty while stack1 is not empty, push everything from stack1 to stack2.
  3) Pop the element from stack2 and return it.
- Here time complexity will be O(n)

Code : LP_Code4.java

Output :

```
1 2 3
```

What is a Deque (or double-ended queue) ?

The deque stands for Double Ended Queue. Deque is a linear data structure where the insertion and deletion operations are performed from both ends. Though the insertion and deletion in a deque can be performed on both ends, it does not follow the FIFO rule. The representation of a deque is given as follows -

Representation of deque

Fixed Window

In a fixed window we have a fixed length in which we have to traverse and find the solution. Now imagine if we have an array of 10 elements and a fixed window of size 3. Now we first check the first 3 elements, then we check 2,3,4 elements and so on. For this process we need two for loops.

This problem can be solved with the Sliding window technique.

Another example is when we need to check a certain property among all the sizes of an array then Sliding window technique also comes into picture.

Sliding window:

The Sliding window is a problem-solving technique of data structure and algorithm for problems that apply arrays or lists. These problems are painless to solve using a brute force approach in **O(n²) or O(n³)**. However, the **Sliding window** technique can reduce the time complexity to **O(n).**

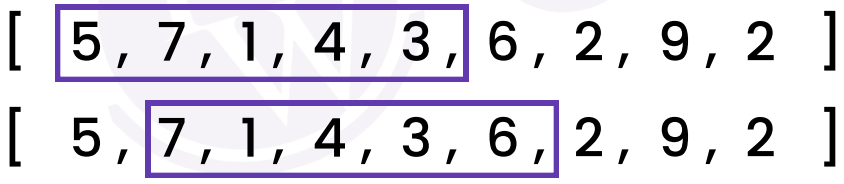


Figure 1: Sliding window technique to find the largest sum of 5 consecutive numbers.

The basic idea behind the sliding window technique is to transform two nested loops into a single loop.

Below are some fundamental clues to identify such kind of problem:

The problem will be based on an array, list or string type of data structure.

- It will ask to find subranges in that array or string and will have to give longest, shortest, or target values.
- Its concept is mainly based on ideas like the longest sequence or shortest sequence of something that satisfies a given condition perfectly.

Let's say that if you have an array like below:
[a b c d e f g h]

A sliding window of **size 3** would run over it like below:

```
[a b c]
  [b c d]
    [c d e]
      [d e f]
        [e f g]
          [f g h]
```

Q4 : 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]

Solution :

Code : LP_Code5.java

Output :

```
The desired output is :
5  5  5  9  9
```

Approach :

- We scan the array from 0 to n-1, keeping only useful elements in the deque. The algorithm is amortized O(n) as each element is put and polled once.
- At each i, we keep "promising" elements, which are potentially max numbers in window [i-(k-1),i] or any subsequent window. This means
- If an element in the deque is out of i-(k-1), we discard them. We just need to poll from the head, as we are using a deque and elements are ordered as the sequence in the array
- Now only those elements within [i-(k-1),i] are in the deque. We then discard elements smaller than a[i] from the tail. This is because if a[x] <a[i] and x<i, then a[x] has no chance to be the "max" in [i-(k-1),i], or any other

subsequent window: a[i] would always be a better candidate.

- As a result elements in the deque are ordered in both sequence in array and their value. At each step the head of the deque is the max element in [i-(k-1),i].

Time complexity : O(n) where n = size of the array
Space complexity : O(n)