30/04/2023

Q1 Reverse a queue.

i/p →	3	6	9	2	8
o/p →	8	2	9	6	3

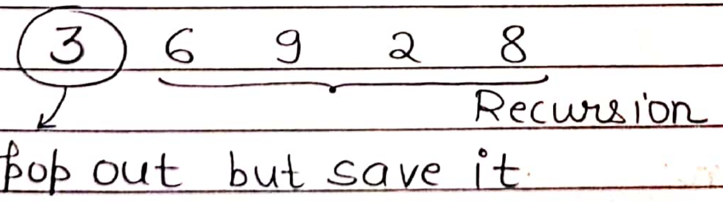
Approach-1

Simply create a stack and pop each element of the queue & push in stack. Now pop element from stack & push in the queue.

Time complexity = O(n)
Space complexity = O(n)

Approach-2

We can reverse the queue with the help of recursion also. We will just solve one case & rest recursion will handle.



1) Save front element and pop it.
2) Recursive call for rest of elements.

8 2 9 6

3) Now simply push the front element which was saved.

8	2	9	6	3

Hence the queue has been reversed.

Code

(i) Using stack

```
void reverseQueue (queue <int> & q) {
        // Create stack
        stack <int>st;
       // Push queue elements to stack
       while (! q.empty ()) {
              int element  = q.front ();
              st.push (element); // Insert in stack
              q.pop(); // Remove from queue
     }
    // Push elements from stack to queue
    while (!st.empty ()) {
            int element  = st.top();
            st.pop();
            q.push (element);
    }
}
```

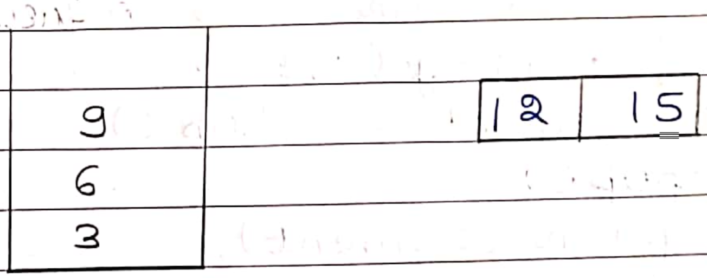
(ii) Using recursion

```
void reverseQueue (queue <int> & q){
        // Base case
        if (a.empty()){
              return;
      }
      //Save front element
      int element = q.front();
      q.pop();
    // Recursive call
    reverseQueue(q);
   // Push back the element
   q.push(element);
}
```

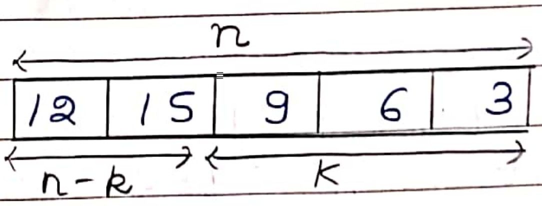
Q2 Reverse first k elements of queue.

i/p →	3	6	9	12	15	, k = 3
o/p →	9	6	3	12	15	

1) First insert the k elements in the stack.



2) Simply pop elements from stack & push in queue.



3) First pop n-k elements & then pop in the queue.

9	6	3	12	15

Code

```
void reversekQueue (queue <int> &q, int k){
    //Creation of stack
    stack <int> s;
    int count = 0;
    int n= q.size();
   //First insert k elements into stack
   while (!q.empty()){
       // Fetch front element
       int temp = q.front();
       q.pop();
       s.push(temp);
       count++;
       if (count == k){ //k elements to be
               break;      pushed
       }
   }
  // Push elements from stack to queue
  while (!s.empty()){
       int element = s.top();
       s.pop();
       q.push(element);
  }
  //Push the (n-k) elements
  count = 0;
                                  && (n-k)!=0
  while (!q.empty()){                ↳ To handle k=size
       int temp = q. front();                   case
       q.pop();
       q.push(temp);
```

```
            count ++;
            if (count == n-k){ // n-k elements to
                    break;          be pushed
            }
        }
    }
```

Time complexity = O(n)
Space complexity = O(n)

Extra condition

```
if (k<=0 || k>n)
        return;      } Do nothing in this
                       case.
```

Q3 Interleave 1st and 2nd half of the queue.

↓ mid (can't be accessed)

i/p →	10	20	30	40	50	60	70	80
o/p →	10	50	20	60	30	70	40	80

Interleaving means 1st element of queue will come and then element after mid will come & so on.

1st half → 10, 20, 30, 40
2nd half → 50, 60, 70, 80

1) Seperate out the 2 halfs.
New queue → 10 20 30 40 } 1st half of original queue
original queue → 50 60 70 80

2) Pop & push element from new queue to original queue. Then pop & push element from original queue to the original queue.

(i) original queue → 50 60 70 80 10
new queue → 20 30 40

(ii) OQ → 60 70 80 10 50
NQ → 20 30 40

(iii) OQ → 60 70 80 10 50 20
NQ → 30 40

(iv) OQ → 70 80 10 50 20 60
NQ → 30 40

(v) OQ → 70 80 10 50 20 60 30
NQ → 40

(vi) OQ → 80 10 50 20 60 30 70
NQ → 40

(vii) OQ → 80 10 50 20 60 30 70 40
NQ →

(viii) OQ → 10 50 20 60 30 70 40 80

Code

```
void interLeave Queue (queue <int> & oQ) {
      //Nothing to do if empty queue
      if (oQ.empty()) {
            return;
      }
      int n = oQ.size();
      int k = n/2
      int count = 0;
```

```
        queue <int> & nQ;
       // Push half elements of oQ to nQ
       while (! oQ. empty ()) {
            int temp = oQ.front ();
            oQ.pop ();
            nQ.push (temp);
            count ++;
            //Done half elements → break
            if(count == k) {
                  break;
            }
       }
      // Start interleaving
      while (! oQ.empty () && !nQ.empty ()){
          // Push from nQ to oQ
          inf first  = nQ.front ();
          nQ.pop ();
          oQ. push (first);
         // Push from oQ to nQ
         int second = oQ.front ();
         oQ.pop ();
         oQ. push (second);
      }
     //Handling odd case
     if (n & 1) {//oQ has one extra element than nQ
           int element = oQ.front ();
           oQ.pop ();
           oQ.push (element);
     }
}
```

Note → Checking odd or even with the bitwise & operator

3 → 11

11 & 01 = 1 } True → Odd

4 → 100

100 & 001 = 000 = 0 } False → Even

Q4 First negative integer in every window of size k. (Process first window and then process other windows via insertion & deletion)

i/p →

12	-1	-7	8	-15	30	16	28

k = 3

o/p → {-1, -1, -7, -15, -15, 0}

↳ no negative integer

{12, -1, -7} → -1

{-1, -7, 8} → -1

{-7, 8, -15} → -7

{8, -15, 30} → -15

{-15, 30, 16} → -15

{30, 16, 28} → 0

Approach-1

If the input is a vector, then we can do the question by using 2 nested loops but the time complexity will be O(n × k).

Approach-2

Here when we are going to the new window, then front element is getting removed whereas a new element is pushed from rear & this is possible in queue data structure.

First window of k size we need to process.

While processing simply push the index of negative elements in the queue.

queue →	1	2

q.front () has some value, then arr [q.front()] is the answer else 0 is the answer

* Remove the front element of i/p array
* Now new element insertion only if negative.

Note - We have stored index in queue so that we can get to know whether it lies in the window of size k or not. This can be know via indexes & not by storing the elements.

Formulae for removal of element from the queue

$i - q.front() >= k$

12 -1 -7 8 -15 30 16 28

↗ q.front() ↗ i

$4 - 1 >= k$

$3 >= 3$ } True & hence index of -1 to be removed from the queue.

Code

```
void solve (int arr [ ], int size, int k) {

        vector<int> ans;
        //Create a deque
```

```
deque <int> q;
//Process first window of size = k
for (int i=0; i<k; i++){
    //Push -ve number index in deque
    if (arr [i] <0){
            q.push_back (i);
    }
}
// Process the remaining window
for (int i = k ; i< size; i++){
        //Store answer of previous window
        if (q.empty ()){
                ans.push_back (0);
        }
        else { //Push the front element in answer
                int temp = arr [q.front ()];
                ans.push_back (temp);
        }
        //Out of window element to be removed
        while (!q.empty () && (i-q.front()>=k){
                q.pop_front ();
        }
        // Push index of the -ve number
        if (arr[i] < 0){
                q.push_back (i);
        }
}
// Process the last window
if (q.empty ()){
        ans.push_back (0);
}
else {
        int temp= arr[q.front()];
        ans.push_back(temp);
    }
    // Print the ans vector
    for(int i=0; i<ans.size(); i++){
            cout <<       [i] << "  ";
    }
}
```

Time complexity = O(n)

Space complexity = O(k) → Can be done in constant space.