## Questions

1. Use a stack to test for balanced parentheses, when scanning the following expressions. Your solution should show the state of the stack each time it is modified. The "state of the stack" must indicate which is the top element.

   Only consider the parentheses `[,],(,),{,}` . Ignore the variables and operators.

   (a) `[ a + { b / ( c - d ) + e / (f + g ) } - h ]`

   (b) `[ a { b + [ c ( d + e ) - f ] + g }`

2. Suppose you have a stack in which the values 1 through 5 must be pushed on the stack in that order, but that an item on the stack can be popped at any time. Give a sequence of push and pop operations such that the values are popped in the following order:

   (a) 2, 4, 5, 3, 1

   (b) 1, 5, 4, 2, 3

   (c) 1, 3, 5, 4, 2

   It might not be possible in each case.

3. (a) Suppose you have three stacks s1, s2, s3 with starting configuration shown on the left, and finishing condition shown on the right. Give a sequence of push and pop operations that take you from start to finish. For example, to pop the top element of `s1` and push it onto `s3`, you would write `s3.push( s1.pop())`.

```
               start                                      finish
     A                                                                    A
     B                                                                    B
     C                                                                    D
     D                                                                    C
    ---        ---        ---                      ---       ---        ---
     s1        s2         s3                        s1       s2         s3
```

   (b) Same question, but now suppose the finish configuration on s3 is `BDAC` (with `B` on top) ?

4. Consider the following sequence of stack commands:

   `push(a), push(b), push(c), pop(), push(d), push(e), pop(), pop(), pop(), pop().`

   (a) What is the order in which the elements are popped ? (Give a list and indicate which was popped first.)

   (b) Change the position of the `pop()` commands in the above sequence so that the items are popped in the following order: `b,d,c,a,e`.

   You are *not* allowed to change the ordering of the `push` commands.

5. Given a queue of integers, how would you reverse the order of the first k elements, leaving the other elements in the same relative order? For example: if k=4 and queue is [10, 20, 30, 40, 50, 60, 70, 80, 90], then, the output should be [40, 30, 20, 10, 50, 60, 70, 80, 90].

   Assume that the size of queue is denoted by size(q) and it is available to you. Also, assume you can perform usual queue operations on your queue. Finally, and very important, assume you have access to a stack and its usual operations of push and pop.

   **I discussed the next question in the queue lecture (slides only).**

6. Assume you have a stack with operations: `push()`, `pop()`, `isEmpty()`. How would you use these stack operations to simulate a queue, in particular, the operations `enqueue()` and `dequeue()`?

   One solution was presented in the lecture. See if you can come up with another.

7. Assume you have a queue with operations: `enqueue()`, `dequeue()`, `isEmpty()`. How would you use the queue methods to simulate a stack, in particular, `push()` and pop() ?

   Hint: use two queues, one of which is the main one and one is temporary.

## Answers

1. (a)
```
-        means empty stack
[
[{
[{(      TOP OF STACK IS ON THE RIGHT
[{
[{(
[{
[
-        empty stack, so brackets match
```
   (b)
```
-
[
[{
[{[         TOP OF STACK IS ON THE RIGHT
[{[(
[{[
[{
[           stack not empty, so brackets don't match
```

2.

```
24531           15423               13542

push 1          push 1              push 1
push 2          pop                 pop
pop             push 2              push 2
push 3          push 3              push 3
push 4          push 4              pop
pop             push 5              push 4
push 5          pop                 push 5
pop             pop                 pop
pop              x                  pop
pop            (not possible)       pop
```

3. (a)

```
s2.push( s1.pop() )
s2.push( s1.pop() )
s3.push( s1.pop() )
s3.push( s1.pop() )
s3.push( s2.pop() )
s3.push( s2.pop() )
```

   (b)

```
s2.push( s1.pop() )
s2.push( s1.pop() )
s3.push( s1.pop() )
s1.push( s2.pop() )
s3.push( s2.pop() )
s2.push( s1.pop() )
s3.push( s1.pop() )
s3.push( s2.pop() )
```

4. (a) c (popped first), e, d, b, a

   (b) push(a), push(b), pop(), push(c), push(d), pop(), pop(), pop() push(e), pop()

5.

```
reverseQueueFirstKElements(int k, Queue q)
{
    if(q is empty or k > size(q)){
        return //do nothing
    }
    else if(k>0){
        tmp = new empty stack
        for(int i = 0; i < k: i++){
            tmp.push(dequeue(q))
        }
        while(!isEmptyStack(tmp)){
            enqueue(q,pop(tmp))
        }
        for(int i = 0; i<size(q) - k; i++){
            enqueue(q, dequeue(q))
        }
    }
}
```

6. Here I present two solutions. Note that each requires that at least one of the `enqueue()` or `dequeue()` operation is $O(N)$ where $N$ is the size of the queue.

   **Solution 1**

   The first solution was presented in the lecture. Implement `enqueue(e)` simply by pushing the new element onto the main stack `s`.

   ```
     enqueue(e){
       s.push(e)
     }
   ```

   In this solution, the bottom of the stack would be the front of the queue (containing the element that has been in the queue longest) and the top of the stack is the back of the queue (containing the least recently added element). How can we implement `dequeue`, that is, how do we remove the bottom element of the stack?

   The idea is to use a second stack `tmpS`. We first pop all items from the main stack `s`, and push each popped element directly onto the second stack `tmpS`. We then pop the top element of the second stack (which is the oldest element in the set). Finally, we refill the main stack `s` by popping all elements from the second stack and pushing each back on to the first stack.

   ```
   dequeue(){
     tmpS <-  new empty stack
     while !s.isEmpty(){
       tmpS.push( s.pop() )
     }
     returnValue <- tmpS.pop()               //  the dequeued element
     while  !(tmpS.isEmpty()){
       s.push( tmpS.pop() )
     }
     return  returnValue
   }
   ```

   **Solution 2**

   Here we let `dequeue()` be simple and just pop the stack. For this to work, the stack needs to store the elements such that the oldest element is on top of the stack and the most recently added element is at the bottom of the stack.

   ```
   dequeue(){   return s.pop()   }
   ```

   With this solution, `enqueue(e)` needs to do the heavy lifting: `enqueue` uses a temporary stack to invert the order of elements currently in the main stack, so that the newest element is on top of this temporary stack. Then it pushes the new element onto the empty main stack. Then it copies all the elements back from the temporary stack to the main stack, using pop-push. This recreates the original stack, but now the newest element has been added on the bottom.

```
enqueue(e){
  tmp <- new empty stack
  while  ! (s.isEmpty()){
    tmpS.push( s.pop() )
  }
  s.push(e)
  while ! (tmpS.isEmpty()){
    s.push( tmpS.pop())
  }
}
```

7. The concepts are similar to the previous question.

**Solution 1**

```
push(e){    q.enqueue(e)   }

pop(){
  tmpQ <-  new empty queue
  while   ! (q.isEmpty()){
    tmpE  <-   q.dequeue()
    if   ! (q.isEmpty())
      tmpQ.enqueue( tmpE )
    else{
      while   !( tmpQ.isEmpty())
          q.enqueue( tmpQ.dequeue())
      return tmpE
    }
  }
}
```

**Solution 2**

Here the idea is similar, but now push does most of the work.

```
pop(){
  q.dequeue()
}
```

```
push(e){
  make a new empty queue qTmp
  qTmp.enqueue(e)
  while !q.isEmpty(){
     qTmp.enqueue( q.dequeue() )
  }
  q <- qTmp
  return qTmp
}
```