Section C – Stack and Queues

1. (**createQueueFromLinkedList**) Write a C function `createQueueFromLinkedList()` to create a queue (linked-list-based) by enqueuing all integers which are stored in the linked list. The first node of the linked list is enqueued first, and then the second node, and so on. Remember to empty the queue at the beginning, if the queue is not empty.

   The function prototypes are given as follows:

   ```
   void createQueueFromLinkedList(LinkedList *ll , Queue *q);
   ```

   A sample input and output session is given below (if the current linked list is **1, 2, 3, 4, 5**):

   ```
   The resulting linked list is: 1 2 3 4 5
   Please input your choice(1/2/3/0): 2
   The resulting queue is: 1 2 3 4 5
   ```

2. **(createStackFromLinkedList)** Write a C function `createStackFromLinkedList()` to create a stack (linked-list-based) by pushing all integers that are storing in the linked list. The first node of the linked list is pushed first, and then the second node, and so on. Remember to empty the stack at the beginning, if the stack is not empty.

   The function prototypes are given as follows:

   ```
   void createStackFromLinkedList(LinkedList *ll , Stack *stack);
   ```

   A sample input and output session is given below (if the current linked list is **1, 3, 5, 6, 7**):

   ```
   The resulting linked list is: 1 3 5 6 7
   Please input your choice(1/2/3/0): 2
   The resulting stack is: 7 6 5 3 1
   ```

3. (**isStackPairwiseConsecutive**) Write a C function write a function `isStackPairwiseConsecutive()` that checks whether numbers in the stack are pairwise consecutive or not. Note that the `isStackPairwiseConsecutive()` function **only** uses `push()` and `pop()` when adding or removing integers from the stack.

   The function prototype is given as follows:

   ```
   int isStackPairwiseConsecutive(Stack *s);
   ```

   Sample test cases are given below:

   For example, if the stack is (**16, 15, 11, 10, 5, 4**):

   ```
   The stack is: 16 15 11 10 5 4
   The stack is pairwise consecutive
   ```

   For example, if the stack is (**16, 15, 11, 10, 5, 1**)

   ```
   The stack is: 16 15 11 10 5 1
   The stack is not pairwise consecutive
   ```

   For example, if the stack is (**16, 15, 11, 10, 5**)

```
The stack is: 16 15 11 10 5
The stack is not pairwise consecutive
```

4. **(reverseQueue)** Write a C function `reverseQueue()` to reverse a queue using **a stack**. Note that the `reverseQueue()` function only uses `push()` and `pop()` when adding or removing integers from the stack and only uses `enqueue()` and `dequeue()` when adding or removing integers from the queue. Remember to empty the stack at the beginning, if the stack is not empty.

   The function prototype is given as follows:
   ```
   void reverseQueue(Queue *q);
   ```

   For example, if the queue is (**1, 2, 3, 4, 5**) the resulting queue will be (**5, 4, 3, 2, 1**).

5. **(recursiveReverseQueue)** Write a **recursive** C function `recursiveReverseQueue()` that reverses the order of items stored in a queue of integers.

   The function prototype is given as follows:
   ```
   void recursiveReverseQueue(Queue *q);
   ```

   For example, if the queue is (**1, 2, 3, 4, 5**), then the resulting queue will be (**5, 4, 3, 2, 1**).

6. **(removeUntilStack)** Write a C function `removeUntilStack()` that pops all values off a stack of integers until the first occurrence of the chosen value.

   The function prototype is given as follows:
   ```
   void removeUntilStack(Stack *s, int value);
   ```

   Given a stack (**1, 2, 3, 4, 5, 6, 7**) with the topmost number displayed on the left, calling `removeUntilStack()` with **value = 4** will produce the stack (**4, 5, 6, 7**).

   Given a stack (**10, 20, 15, 25, 5**) with the topmost number displayed on the left, calling `removeUntilStack()` with **value = 15** will produce the stack (**15, 25, 5**).

7. **(balanced)** Write a C function `balanced()` that determines if an expression comprised of the characters `()[]{}` is balanced. The prototype for the `balanced()` function is given below:

   The function prototype is given as follows:
   ```
   int balanced(char *expression);
   ```

   For example, the following expressions are balanced because the order and quantity of the parentheses match:
   ```
   ()
   ([])
   {[]()[]}
   ```

   A sample input and output session is given below:
   ```
   1: Enter a string:
   ```

```
2: Check whether expressions comprised of the characters ()[]{}
is balanced:
0: Quit:

Please input your choice(1/0): 1
Enter expressions without spaces to check whether it is
balanced or not: {[]()[]}
{[]()[]}
balanced!

Please input your choice(1/0): 0
```

The following expressions are not balanced:

```
{{)]
[({{)])
```

A sample input and output session is given below:

```
1: Enter a string:
2: Check whether expressions comprised of the characters ()[]{}
   is balanced:
0: Quit:

Please input your choice(1/0): 1
Enter expressions without spaces to check whether it is
balanced or not: [({{)])
[({{)])
not balanced!

Please input your choice(1/0): 0
```