King Saud University
Department of Computer Science
CSC 212 Homework # 4
Stack & Recursion

Guidelines:	This is an individual assignment. The homework must be **submitted electronically through LMS**. **Submissions by email and hard copy submissions are not accepted**.

Problem 1

1. Convert the following expression to postfix notation, show the stacks after each push: `b - c + d / d * a ^ f + b`.
2. Trace the evaluation of the following expression, show the stack after each push: `8 5 2 ^ 3 4 + 6 * + 2 - *`.
3. Convert the following expression to infix notation, show the stack after each push: `6 5 1 ^ 3 2 + 6 * - 8 - *`.
4. Trace the evaluation of the following expression, show the stacks after each push: `6 + 3 ^ 2 / 3 / 3 - 2 * 3 * 4`.

. . . . . .

Problem 2

1. Write the static method, user of the ADT Stack, **public static** `<T extends Comparable<T>> Stack<T> mergeSortedStacks(Stack<T> s1, Stack<T> s2)`, that takes two sorted stacks `s1` and `s2` (minimum on top) and returns one stack that is sorted (min on top). Do not use any auxiliary data structures other than stacks.

Example 2.1. *If s1(top to bottom):* $1, 4, 6, 8$ *and s2:* $2, 3, 9$*, then after calling mergeSortedStacks (s1,s2) it will return stack contains :* $1, 2, 3, 4, 6, 8, 9$.

2. Write a static method `pushElement`, user of the Stack ADT, that pushes all occurrences of the element `e` to the bottom of the stack `st`, keeping the order of the other elements unchanged. The method signature is **public static** `<T>` **void** `pushElement(Stack<T> st, T e)`.

   Example 2.2. *If st (top to bottom):* $23, 10, 14, 10, 7, 10, 9, 14$. *After calling* `pushElement(st,10)`*, the stack st becomes* `st` *:* $23, 14, 7, 9, 14, 10, 10, 10$.

. . . . . .

Problem 3

1. Write the **recursive** method **public static** `<T extends Comparable<T>> sort(Stack <T> st)` that sorts `st` in increasing order.

2. Write the recursive method **public boolean** `recSearch(T k)` member of the class `LinkedList` that searches the list for element `k`. It should return "true" if if found false otherwise. Do not use other data structures. You can add a private member method as needed.

   Remark 3.1. *Recursive member functions are private in general, since their parameters may depend on the internal representation of the data structure. Consequently, when one talks about a recursive public method, it is understood that the method itself is* ***not recursive****, but calls a private recursive method that does the job.*

3. Write **recursive** method **public** `<T>` **void** `reverse(Queue<T> q)` (user of the Queue ADT), that reverses the order of the elements of `q`.

4. Write the **recursive** method **public static** `<T> merge(Queue<T> q1, Queue<T>q2)` that merges the queues `q1` and `q2` into a new queue. After the call, `q1` and `q2` must not change (**Do not use any loops**).

   Example 3.1. *If the queue* `q1` *contains:* A, B, C*, and* `q2` *contains:* D, E, F, G, H*, then the result of* merge(`q1`, `q2`) *is:* A, D, B, E, C, F, G, H.

. . . . . .

Problem 4

The parts of this problem are related.

1. We add to the interface `List` the method `T car()`, which returns the first element of the list, and the method `List<T> cdr()`, which returns the rest of the elements as a new list. The method `car` cannot be called on an empty list. On the other hand, if the list is empty, the method `cdr` returns an empty list.

   Example 4.1. *If $l : A, B, C, D$, then `l.car()` returns the element A, and `l.cdr()` returns the list B, C, D.*

   Implement these two methods as members of the class `LinkedList`.

2. Write the method **public static** `<T> List<T> list(T e)`, which returns a list containing the single element `e`.

3. Write the method **public static** `<T> List<T> concat(List<T> l1, List<T> l2)`, which returns the concatenation of the lists `l1` and `l2`. The two input lists must not change.

   Example 4.2. *If $l1 : A, B$ and $l2 : C, D$, then `concat(l1, l2)` returns the list A, B, C, D.*

4. Using the methods: `car`, `cdr`, `list` and `concat`, write the following **recursive methods**[1]:

   (a) The method **public static** `<T>` **void** `print(List<T> l)`, which prints the list `l`.

   (b) The method **public static** `<T> List<T> inverse(List<T> l)`, which returns the inverse of the list `l`. The list `l` must not change.

   (c) The method **public static** `<T> List<T> remove(List<T> l, T e)`, which returns the list obtained by removing all occurrences of `e` from `l`. The list `l` must not change.

. . . . . .

[1]The names of the methods `car` and `cdr` are those of two primitive operations in the Lisp language which were named this way for historical reasons (see https://en.wikipedia.org/wiki/CAR_and_CDR). Most programs in Lisp consist in clever use of these two primitives to recursively solve problems.