4. Stack and Queue

Problem 4.1

1. Write the method `public <T> boolean isReverse(DoubleLinkedList <T> l, Queue<T> q)` that accepts a double-linked list `l` and a queue `q`. The method should return true if and only if the elements of `l` are in the reverse order of the elements of `q`. Use the method `equals` for checking for equality. The content of `l` and `q` must not change after the call.
2. Write the method `isReverse`, but this time as a member of the class `ArrayQueue` which accepts a list `l` (**not a double linked list**) to compare against (Do not call any method of the class `ArrayQueue`). The content of `l` and the queue must not change after the call. The method signature is: `public boolean isReverse(List <T> l)`.
3. Write the method `public < T> boolean isReverse(Queue<T> q1, Queue<T> q2)`, that checks if q_2 is the reverse of q_1.
4. Write the method `public <T> void exchange(Queue<T> q1, Queue<T> q2)` that exchanges the content of the two queues **without using any auxiliary data structures**.
5. Write the method `concat` that takes as input a queue of lists and concatenates them into a single list. The queue and the lists must not be changed after the call. The method signature is `public<T> List <T> concat(Queue<List<T>> l)`.
6. Write the method `concat` that takes as input a queue of queues and concatenates them into a single queue. All queues must not be changed after the call. The method signature is `public<T> Queue<T> concat(Queue<Queue<T>> q)`.
7. Given a Queue `q`, we would like to search the queue for an element `e` and delete it while keeping the order of elements intact. Do not use any auxiliary data structures. Write the method void `removeElement(Queue <T> q, T e)`. For example, if we have a queue $10 \to 8 \to 6 \to 7 \to 2$ and want to delete 7, it will be $10 \to 8 \to 6 \to 2$.
8. Write the method `public static <T> void remove(Queue<T> q, int[] pos, int k)`, which removes all the elements of q located at the positions indicated in pos (k is the size of pos). Assume that pos is sorted in increasing order with no duplicates and contains only valid positions. The numbering of the positions starts from 0 at the head. The method must run in $O(n)$, where n is the size of q (not $O(kn)$).

   ■ **Example 4.1** If $q: A, B, C, D, E, F, G, H$ and $pos: 1, 2, 5$, then after calling `remove(q, pos, 3)`, q becomes A, D, E, G, H. ■

9. Write a static method (User of ADT) named `exchange` that accepts a queue q and two integers i and j, and exchanges the elements at positions i and j (the first element in the queue has position 0). The queue order should not change otherwise. Assume that $0 \leq i < j < n$, where n is the length of the queue.
10. Write the method `intersect`, user of Queue ADT, that accepts two queues q_1 and q_2, and returns the intersection of the two queues as a new queue. There shouldn't be any duplicate elements in the new queue. The elements in the returned queue must have the same order as in q_1. The inputs q_1 and q_2 must not change after the method. The method signature is: `public <T> Queue <T> intersect( Queue <T> q1, Queue <T> q2)`.

    ■ **Example 4.2**

$$\begin{aligned} q_1 &: B \to A \to C \to D \to E \to G \\ q_2 &: G \to U \to D \to P \to C \\ \text{Returned queue} &: C \to D \to G \end{aligned}$$

■

11. Write the method `public static <T> void swapAdj(Queue<T> q)` which swaps adjacent elements in the queue starting from the first element. Do not use any extra data structures.

    ■ **Example 4.3** If $q : A, B, C, D, E$, then after calling the method `swapAdj(q)`, q becomes B, A, D, C, E. ■

12. Write the method `public static boolean firstEqLast(Queue<T> q)`, which returns **true** if the first and last elements are equal, and **false** if they are not equal. The queue `q` must not change after the call. Do not use any other data structure. Assume the queue `q` is not empty.

    ■ **Example 4.4** If `q` contains: A, B, C, A, E the method will return **false**. If `q` contains: A, B, C, F, A the method will return **true**. ■

13. Write the method `symDiff`, user of Queue ADT, that accepts two queues q_1 and q_2, and returns the symmetric difference of the two queues as a new queue (the set of elements that are in q_1 or q_2 but not in their intersection). There shouldn't be any duplicate elements in the new queue. In the returned queue, all the elements belonging to q_1 appear in their respective order before all elements belonging to q_2 also in their respective order. The inputs q_1 and q_2 must not change after the method. The method signature is: `public <T> Queue <T> symDiff( Queue <T> q1, Queue <T> q2)`.

    ■ **Example 4.5**

$$\begin{aligned} q_1 &: B \to A \to C \to D \to E \to G \\ q_2 &: G \to U \to D \to P \to C \\ \text{Returned queue} &: B \to A \to E \to U \to P \end{aligned}$$

■

14. Write the method `public static boolean isOrdered(Queue<Integer> q)` which accepts a non empty queue of integers. The method should return true if and only if the queue is sorted in an increasing order (from the lowest to the highest), false otherwise. The content of q must not change after the call. Also, no extra data structures should be used.

    ■ **Example 4.6** If q : $1, 3, 5, 5, 10$, then calling `isOrdered(q)` returns true, whereas if $q : 5, 4, 10$ then calling `isOrdered(q)` returns false. ■

15. Write the method `public <T> void removeItem(Queue<T> q, T e,int i)`, user of ADT Queue, that removes all occurrences of `e` from the queue if `e` appears `i` times or more in the queue. Do not use any auxilary data structures.

   ■ **Example 4.7** If $q : A, B, C, B, D, A, B, C$, after calling `removeItem(q,'B',2)`, q becomes $q : A, C, D, A, C$.
   If $q : A, B, C, B, D, A, B, C$, after calling `removeItem(q,'B',4)`, q does not change since `B` appears only 3 times.

   ■

Problem 4.2

1. Write the method `reverse`, member of the class `LinkedQueue` which reverses the content of the queue. The method signature is: `public void reverse()`.
2. Write the method `reverse`, member of the class `ArrayQueue` which reverses the content of the queue. The method signature is: `public void reverse()`.
3. Write the method `void remove(int k)`, member of the class `LinkedQueue`, that removes the first k elements (assume that k has a valid value). Your method must be $O(k)$.
4. Write the method `void remove(int k)`, member of the class `ArrayQueue`, that removes the first k elements (assume that k has a valid value). Your method must be $O(1)$.
5. Rewrite the method described in Problem 4.1.9 as a member of the class `LinkedQueue` and also as a member of the class `ArrayQueue`. Give th big-oh notation for each of the previous three methods. Which one is the fastest?
6. Write the method `swapWithFirst` (member of `ArrayQueue`), that takes as parameter an integers i, and swaps the element at positions i with the first element in its corresponding half of the queue. If i is in the first half, it will swap with head. If it is in the second half, it will swap with the first element of that half. Assume the queue starts at position 0, and $0 \leq i < n$. Assume the number of elements in the queue is even. Do not use any auxiliary data structure. The method signature is `public void swapWithFirst(int i)`.
7. Suppose you are given the values of `head`, `tail` and `maxSize`, members of `ArrayQueue`, compute the size of the queue.

Problem 4.3

1. Write the method `boolean canBeInserted(PQueue<T> q1, PQueue<T> q2)` that accepts two priority queues, q_1, q_2 and checks whether we can insert all the elements of q_2 between any two elements in the first priority queue q_1. The method either returns true if the operation is possible or false if it is not (notice that the method only performs a check, neither q_1 nor q_2 are actually changed).
2. Using the two previous methods, write the method `removeLowest` (user of the ADT) that takes as input a linked priority queue (with the two additional methods `removePr` and `lowestPr`) and removes all the elements having the lowest priority. The method signature is `public void removeLowest(LinkedPQ<T> q)`.

   ■ **Example 4.8** If the queue q contains $2 \to 3 \to 5 \to 7 \to 7 \to 9 \to 9 \to 9$, then, after the call to `removeLowest(q)`, the queue becomes $2 \to 3 \to 5 \to 7 \to 7$. ■

3. Write the method `changePriority()`, user of the ADT PQueue, that changes the priority of the elements with the highest priority (with the biggest number) to the lowest priority (with the smallest number). The method signature is: `public<T> void changePriority(PQueue<T> pq)`.

■ **Example 4.9** If pq: $(C,7) \to (B,7) \to (A,7) \to (G,6) \to (F,4) \to (D,1)$, then after calling `changePriority(pq)`, pq becomes: $(G,6) \to (F,4) \to (D,1) \to (C,1) \to (B,1) \to (A,1)$. ■

Problem 4.4

1. Write the method `public static <T> void print(LinkedPQ<T> q)` which print the content of `q` in decreasing order of priority. The method must not use any auxiliary data structures (space complexity $O(1)$).
2. Write the method `public void mergePQ(LinkedPQ<T> q)`, member of the class `LinkedPQ`, that merges the priority q with the current one (keeping q unchanged). The method must have a **linear** performance.
3. Write the method `removePr`, member of the class `LinkedPQ` (linked priority queue) that takes as input an integer pr and removes all the elements having the priority pr. The method signature is: `public void removePr(int pr)`. Do not call any methods and do not use any auxiliary data structure.

   ■ **Example 4.10** If the queue contains $2 \to 3 \to 5 \to 7 \to 7 \to 9 \to 9 \to 9$, then, after the call to `removePr(7)`, the queue becomes $2 \to 3 \to 5 \to 9 \to 9 \to 9$. ■

4. Write the method `lowestPr`, member of the class `LinkedPQ` (linked priority queue) that returns the lowest priority in the queue. The method signature is: `public int lowestPr()`. Do not call any methods and do not use any auxiliary data structure.

   ■ **Example 4.11** If the queue contains $2 \to 3 \to 5 \to 7 \to 7 \to 9 \to 9 \to 9$, then, the call to `lowestPr` returns 9. ■

5. Write an algorithm that uses a priority queue to sort an array. If you use the implementation `LinkedPQueue` seen in class, what would be the performance of this sorting algorithm?

Problem 4.5

1. Consider the following code:

```
public class Test {
    public static void f1(int n) {
        n++;
        f2(n);
        f3(n);
        f2(n);
    }
    public static void f2(int n) {
        n++;
        f3(n);
    }
    public static void f3(int n) {
        System.out.println(n);
    }
    public static void main(String[] args) {
        int n = 3;
        f1(n);
        n++;
        f2(n);
    }
}
```

Which of the following snapshots of the call stack are valid for this code?

				f2	
f3				f3	f2
f2	f3	f3	f3	f2	f3
f1	f1	f2	f2	f1	f1
main	main	f1	main	main	main

2. Suppose you want to check parentheses balance for expressions that contain a single type of parentheses. Would you need a stack for this task? Write a pseudo-code solution for this problem.
3. Trace the execution of the evaluation of the following expression: **2 9 3 1 + * 5 4 3 % 1 - - + $>$ 35 14 8 + =** $||$. Show the content of the data structure(s) **after** parsing each operation.

<table>
<tr><td>+</td><td>*</td><td>%</td><td>-</td><td>-</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>+</td><td>></td><td>+</td><td>=</td><td>||</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td></tr>
</table>

4. Trace the execution of the evaluation of the following expression: **4 + (9 -(3 * 2)) % 3 + 5 * (2 +(6 / 3)) -1**. Draw the content of the data structure(s) after parsing each operation.

+	-	*	%	+
*	+	/	-	$

5. Trace the execution of the conversion of the following expression into infix notation: **2 9 3 1 + * 5 4 3 % 1 - - + > 35 14 8 + =** ||. Show the content of the data structure(s) **after** parsing each operation.

+	*	%	-	-
+	>	+	=	||

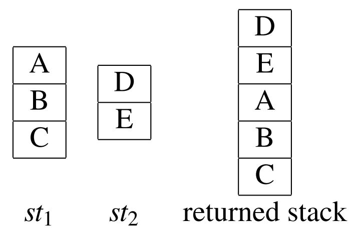
6. Trace the execution of the conversion of the following expression: **4 + (9 -(3 * 2)) % 3 + 5 * (2 +(6 / 3)) -1** into postfix notation. Draw the content of the data structure(s) after parsing each operation.

+	-	*	%	+
*	+	/	-	$

Problem 4.6

1. Write the static method `public static <T> Stack<T> concat(Stack <T> st1, Stack<T> st2)` (user of the ADT stack) that takes as input two stacks st_1, st_2 and returns their concatenation as a new stack (the two stacks st_1 and st_2 must not change).

   ■ **Example 4.12** This is an example:

   ■

2. Write the method `pushBack` (user of the Stack ADT), that takes a stack `st` and an integer `p`, and pushes `the top of the stack` to the p^{th} position. The top element in the stack is in position 1. Assume that $1 \leq p \leq n$, where n is the length of `st`. The method signature is `public <T> void pushBack(Stack<T> st, int p)`.

   ■ **Example 4.13** Assuming the stack st (from top to bottom): $1, 2, 5, 3, 10, 6$. Calling `s.pushBack(st, 4)` will result in $st : 2, 5, 3, 1, 10, 6$. ■

3. As a user of the ADT Stack, write the following method that checks if the top element of a stack of Integers is equal to the total sum of all lower elements in the stack. It returns true if they are equal and false otherwise. The stack should not be changed after you call the method. The method signature is `public boolean checkTotalTop(Stack<Integer> st)`.
4. Write the static method `moveAfter` (user of the Stack ADT), that takes as input two stacks st_1, st_2 and an index i. It moves the elements of stack st_2 after the element at position i in

stack st_1. Assume that i is within the range of stack st_1, and that the top element has an index 0. The signature is `public static <T> void moveAfter(Stack<T> st1, Stack<T> st2, int i)`.

■ **Example 4.14** If st_1 (top to bottom): 5, 2, 4, 1 and st_2 (top to bottom): 8, 9. After calling `moveAfter(st1, st2, 1)`, st_1 will be (top to bottom): 5, 2, 8, 9, 4, 1. ■

5. Write the static method `countEquals` (user of the Stack ADT), that takes as input a stack st, and an element e. It returns the number of elements of stack st matching e. The stack st should **not change** after calling the method. The method signature is `public static <T> int countEquals(Stack<T> st, T e)`.

■ **Example 4.15** If st (top to bottom): 5, 2, 4, 1, 4, 2, 4. Then `countEquals(st, 4)` returns 3, `countEquals(st, 2)` returns 2, and `countEquals(st, 7)` returns 0. ■

6. Write the static method `public static <T> void removeLast(Stack<T> st)` (user of Stack ADT) that takes a stack st as input, and removes the bottom element of st.

■ **Example 4.16** Assuming st (top-to-bottom): $5,7,5,3,2$. After calling `removeBottom(st)` st will be: $5,7,5,3$. ■

7. Write the method `public static <T> boolean topEqualsBottom(Stack<T> st)` that checks if the top element of the stack is equal to bottom element. Return true is that is the case. The stack st should not change after the method has been called.
8. Write the method `pullUpBottom`, user of the ADT Stack, that moves the element in the bottom of the stack to the top without changing the order of the other elements. The method signature is: `public<T> void pullUpBottom(Stack<T>)`.

■ **Example 4.17** If st (top to bottom): $A \to B \to E \to C$, after calling `pullUpBottom(st)`, st becomes $C \to A \to B \to E$. ■

9. Write the method `public static Stack<Character> replace(Stack<Character> st, char a, char b)`, which replaces all occurrences of the char `a` in the stack `st` by the char `b` and returns the result as a new stack. The stack `st` must not change after the call.

■ **Example 4.18** If `st` before the call contains: `'A', 'B','C', 'A', 'E'` (from top to bottom), and we called: `replace(st, 'A', 'B')`, then the returned stack contains: `'B', 'B', 'C', 'B', 'E'`, and `st` remains unchanged. ■

10. Write the method `public static <T> int nbCommon(Stack<T> st1, Stack<T> st2)` which returns the number of elements that appears in both stacks. Assume that elements are unique within each stack.

■ **Example 4.19** If $st_1 : A,B,C,D,E,F$ and $st_2 : F,B,C,J$, then `nbCommon(st1, st2)` returns 2. ■

Problem 4.7

1. Write a **recursive** method that removes all elements from a stack.
2. Write a **recursive** method that removes all elements from a queue.
3. Write the **recursive** method `void removeEle(Stack<T> st, T e)` that deletes **all the occurrences** of the element e from the stack st keeping all other elements in their order.
4. Write the **recursive** method `int stackSum(Stack<Integer> st)` that sums all the elements in the stack and returns the total result. The stack must not changed at then end of method.
5. Write a method that counts the number of occurrences of an element in a stack.
6. Write a method that counts the number of occurrences of an element in a queue (the queue

should not change).

7. Write a **recursive** method that removes all occurrences of an element `e` from a queue (the queue should not change).
8. Write a **recursive** method that computes the sum of all elements in a queue of integers (the queue should not change).
9. Write a **recursive** method that inserts an element `e` at the bottom of a stack.
10. Write a **recursive** method that inserts an element `e` at the head of a queue.
11. Write a **recursive** method that reverses a stack.
12. Write a **recursive** method that reverses a queue.
13. Write a **recursive** method `public static <T> int compareLength(Stack<T> st1, Stack<T> st2)`. The return value follows the same convention as `compareTo`.
14. Write a recursive method $merge(q_1, q_2)$ that merges the queues q_1 and q_2 into a new queue. After the call, q_1 and q_2 become empty (**Do not use any loops**). The method signature is: `public <T> Queue<T> merge(Queue<T> q1, Queue<T> q2)`.
15. Rewrite the method `merge` so that the two queues q_1 and q_2 do not change after the call (**Do not use any loops**).
16. Write the **recursive** method `public static <T> void copyAtEnd(Stack<T> st, Queue<T> q)` that copies all elements of `st` top to bottom at the end of `q`. The stack `st` must not change.
17. Write the **recursive** method `public static <T> void copyAtEnd(Stack<T> st, Queue<T> q)` that copies all elements of `st` bottom to top at the end of `q`. The stack `st` must not change.
18. Write a **recursive** method that copies a queue head to tail at the top of a stack.
19. Write a **recursive** method that copies a queue head to tail at the bottom of a stack.

Problem 4.8

Suppose you want to implement the ADT Stack and Queue using a list for internal storage.

1. Which of the two implementations of list, `ArrayList` and `LinkedList`, would you to implement each of the two ADTs? Justify your answer.
2. Write the implementations corresponding to your choice.

Problem 4.9

The goal in this problem is to implement a generalization of the ADT queue called `MultiQueue`, which consists in a set of queues numbered from 0 to $n-1$. The number of queues n is fixed and specified by the user at the time of the creation of the data structure (see example below).

This data structure is used as follows:

- An element in enqueued in one of the n queues as specified by the user.
- The `Serve` operation chooses the queue from which the element is removed circularly (the queue 0 is selected first after the creation of the multiqueue). If the queue in question is empty, the next non empty queue is selected for the serve.

■ **Example 4.20** This is an example of a multiqueue consisting of four queues. Notice that queue 1 is empty.

0	$\rightarrow 3 \rightarrow 5 \rightarrow 4$
1	$\rightarrow$
2	$\rightarrow 1 \rightarrow 5 \rightarrow 2$
3	$\rightarrow 2 \rightarrow 1 \rightarrow 3 \rightarrow 6$

Assuming that the turn now is for queue 0 to be served, the next 6 serve operations return in order: 3, 1, 2, 5, 5, 1. After that, the multiqueue becomes:

0	$\rightarrow 4$
1	$\rightarrow$
2	$\rightarrow 2$
3	$\rightarrow 3 \rightarrow 6$

■

The following is the specification of the ADT MultiQueue. All operations are performed on a multiqueue called *mq* having *n* queues.

- Procedure length (l: int, i: int). Requires: $0 \leq i < n$. Results: l is set to the length of the queue i.
- Procedure full ($flag$: boolean, i: int). Requires: $0 \leq i < n$. Results: $flag$ is set to true if queue i is full, to true otherwise.
- Procedure enqueue (val: T, i: int). Requires: $0 \leq i < n$ and queue i is not full. Results: val is in enqueued in queue i.
- Procedure serve (val: T). Requires: At least one queue is not empty. Results: val is set to the element to be served.

Use the ADT queue (class `LinkedQueue<T>`) to implement the ADT MultiQueue.

Problem 4.10

1. Write an array implementation of the ADT PQueue. The `serve` method must run in $O(1)$, `enqueue` in $O(n)$.
2. Rewrite the interface `PQueue` so that both data and priority are generic. Give a linked implementation of this interface.
3. Rewrite the interface `PQueue` so that the data itself is comparable and plays the role of the priority. Give a linked implementation of this interface.

Problem 4.11

Consider the interface `Dequeue` below:

```
public interface Dequeue<T> {
    int length();
    boolean full();
    void addFirst(T e);
    void addLast(T e);
    T removeFirst();
    T removeLast();
}
```

1. Give a linked implementation of `Dequeue`.
2. Give an array implementation of `Dequeue`.
3. Compare the performance of the two implementations.
4. Write an implementation of the interface `Queue` that uses a `Dequeue` to store data. Compare the performance of this implementation with `LinkedQueue` and `ArrayQueue`.
5. Write an implementation of the interface `Stack` that uses a `Dequeue` to store data. Compare the performance of this implementation with `LinkedStack` and `ArrayStack`.

Problem 4.12

A store announces a sale campaign whereby any customer who buys two items gets 50% off on the cheaper one. If the customer buys more than two items, he/she must group them into pairs of two to indicate the items that the offer should apply to.

1. Suppose you want to buy n items in total. Write a method that will give you the best pairing of the items (the one with the minimum price). The method's signature is:
   `public static LinkedList<ItemPair> minPairing(LinkedList<Item> items)`.
2. If you leave it up to the store owner, he/she will try to pair the items in order to obtain the maximum price. Write a method that will help the store owner achieve this. The method's signature is:
   `public static LinkedList<ItemPair> maxPairing(LinkedList<Item> items)`.
3. How much will you gain if you use your method (instead of the shop owner's method) for the following list of item prices: 60 SAR, 100 SAR, 400 SAR, 600 SAR, 200 SAR, 80 SAR.

```java
public class Item {
    private int id;
    private double price;
    public Item(int id, double price) {
        this.id = id;
        this.price = price;
    }
    int getId() {
        return id;
    }
    double getPrice() {
        return price;
    }
}
```

```java
public class ItemPair {
    public Item first;
    public Item second;
    public ItemPair(Item first, Item second) {
        this.first = first;
        this.second = second;
    }
}
```

Problem 4.13

In this problem, do not use any auxiliary data structures (in particular, do not use a stack).

1. Write a recursive method, `eval`, to evaluate a postfix expression. The expression is represented as a String and contains the following operators: +, -, * and /. For simplicity, assume that all the numbers are single digit and unsigned, for instance 5, or 6 but not 23, 124 or -4. An example of an input is: ”873-*4+23-*58-+”.
   Programming hint: in order to transform a single character located at position i in a string *exp* to its numerical value, you may use:

   ```java
   val = Character.getNumericValue(exp.charAt(i));
   ```

2. Write a recursive method, `infix`, to transform a postfix expression into an infix one. Use the same assumptions as in the previous question. For simplicity, put all operation between parentheses. For instance, the postfix expression ”23+” is transformed to ”(2+3)”, and ”873-*4+23-*58-+” is transformed to ”((((8*(7-3))+4)*(2-3))+(5-8))”.

```
public class Postfix {

    // Private recursive method.
    private static double recEval(...) {
        ...
    }

    // Public non-recursive
    public static double eval(String exp) {
        ...
    }

    // Private recursive method.
    private static String recInfix(...) {
        ...
    }

    // Public non-recursive
    public static String infix(String exp) {
        ...
    }
}
```

Problem 4.14

★ In prefix notation, the operation precedes the operands like: `+ a b`. Write an algorithm that evaluates an expression written in infix notation.