Assignment 7

Reminder: Your code is to be designed and written by only you and not to be shared with anyone else. See the Course Outline for details explaining the policies on Academic Integrity. Submissions that violate the Academic Integrity policy will be forwarded directly to the Computer Science Academic Integrity Committee.
All materials provided to you for this work are copyrighted, these and all solutions you create for this work cannot be shared in any form (digital, printed or otherwise). Any violations of this will be investigated and reported to Academic Integrity.

Objectives

- More practice implementing linked list data structure
- Creating/using a stack abstract data type.
- Use exceptions for error handling
- Creating/using generic data types

Introduction

In this assignment we continue to explore different data structures in this course – and in particular concept of abstract data types by implementing a generic Stack.

Additionally, you will do some problem solving to complete an application that will make use of your generic stack as well as Java's built-in generic `List` and `LinkedList` abstract data types.

More specifically:

1. You will create the class named `StackRefBased` which will implement the interface named `Stack` (ie., `push`, `pop`, `isEmpty`, etc.) using the same kind of nodes we have seen in linked lists. As part of this step you will test your implementation by working with a stack containing Integer objects.
2. Using your `StackRefBased` and several other classes provided to you, you will complete the `reorderTrains` method in the `Solver` class.

Submission and Grading

Submit your `StackRefBased.java` with the completed methods through the assignment link in BrightSpace.

- You **must** name the methods in `StackRefBased.java` as specified in the given interface and as used in `A7Tester.java` or you will receive a **zero grade** as the tester will not compile.
- If you chose not to complete some of the methods required, you **must at least provide a stub for the incomplete method** in order for our tester to compile.
- If you submit files that do not compile with our tester (ie. an incorrect filename, missing method, etc) you will receive a **zero grade** for the assignment.
- Your code must **not** be written to specifically pass the test cases in the testers, instead, it must work on other inputs. We may change the input values when we run the tests and we will inspect your code for hard-coded solutions.
- **ALL late** and **incorrect** submissions will be given a **ZERO** grade.

Getting Started

1) Download all java files provided in the Assignment folder on BrightSpace.

2) Add stubs for the missing methods in `StackRefBased` class.

3) Compile and run the test program with the `-Xlint:unchecked` flag as follows:
   `javac -Xlint:unchecked A7Tester.java`

   You will see there is a warning for an unchecked cast:

   ```
   ./StackRefBased.java:94: warning: [unchecked] unchecked cast
           StackRefBased<T> otherStack = (StackRefBased<T>)other;
   ```

   You can ignore this warning and continue with your assignment, the issue is mentioned in the method's precondition as well.

   You should NOT ignore any of the other warnings. To get a report of the line numbers of the warnings, delete all .class files in your folder and recompile with:

   `javac -Xlint:unchecked A7Tester.java`

4) Implement each method in `StackRefBased.java` until all of the tests in the `testBasicStack` method pass in `A7Tester.java`

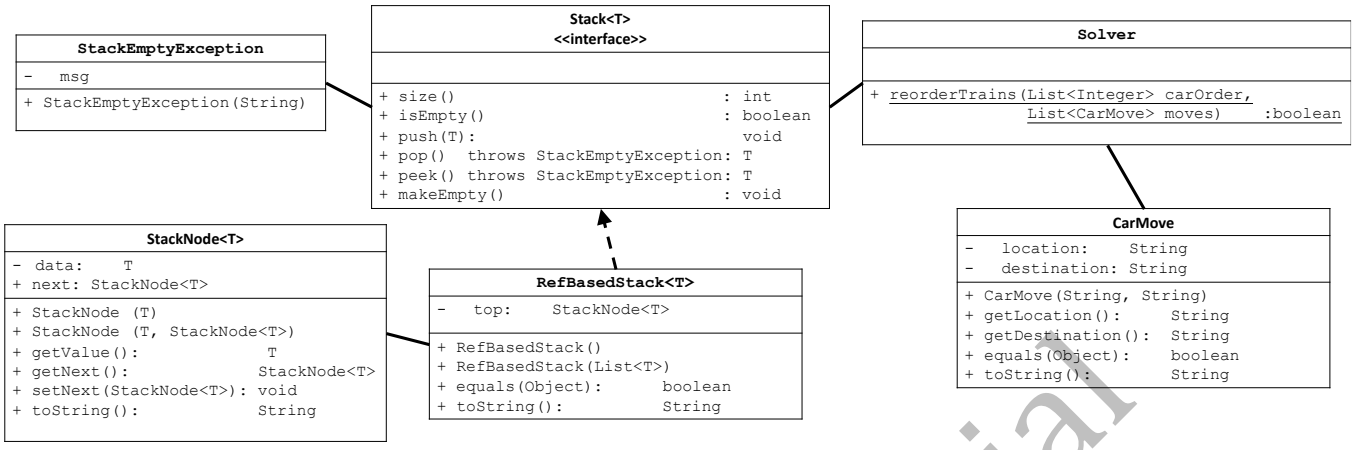
5) Implement the `reorderTrains` method in the Solver class. Uncomment the call to `testTrains` in the `main` of `A7Tester.java` to test your implementation.

The following UML diagram gives you an overview of the classes involved in this assignment. The diagram does not include the Java classes from `java.util` package that you will use in this assignment.

You can find the Java documentation for the `List` interface here:
https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/List.html

Within the `List` documentation are links to ***All Known Implementing Classes*** including the `LinkedList` class.

This means you can use these classes already written for you in the `java.util` package!



NOTE:
the underlining indicates the feature is static

Problem Solving – Train switching yard

In a train-switching yard, sometimes the order of cars must be changed to coincide with the order of the train destinations enroute. The switching is done using a small side-track called a *spur* where the operations are automated as long as the order of the operations is provided.
You are to write a method that will determine if the reordering is possible and if it is, provide a list of the operations to be performed to achieve the proper ordering of train cars.

Your method will also be passed an empty `List` of `CarMove` objects that you will store the operations to be performed for the reordering.
The `CarMove` class has been written for you to use.
Java's `List` and `LinkedList` have been imported to the `Solver` class for you to use.
Examples of their use can be found in the `A7Tester` class.
See the Java API for the documentation of the methods available in the `List` interface:
https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/List.html

In your method you will create separate stacks to represent the input, spur and output tracks and a list to hold `CarMove` objects that represent the operations to be performed for the reordering.

- You must initially push all of the `Integer` objects in the initial ordering of train cars onto an input track.
- Train cars can then be moved from the input track to:
  - the output track (if it is the next car to be outputted)
  - the spur (if it is not the next car to be outputted)
- A car can also be moved from the top of the spur to the output track.
- A car **cannot** be moved to the input track from the spur OR from the output track.
- The reordering is only possible if the values in the given list are unique and range from 1 to the length of the given input list.
  For example:
  `{5, 4, 3, 2, 1}` can be reordered
  `{5, 4, 3, 2}` cannot be reordered
  `{4, 3, 2, 1}` can be reordered
  `{1, 3, 5, 4, 2}` can be reordered
  `{1, 3, 6, 4, 2}` cannot be reordered
  `{18, 3, 56, 24, 2}` cannot be reordered

There are many ways to solve this problem, feel free to design your own algorithm as long as the tracks (input, spur and output) are represented as stacks.

If you are having difficulty designing your own algorithm, the following pages describe an approach with example runs and explanations of a potential algorithm. The algorithm on the following pages provide a description that leverages a fourth stack that keeps track of operations as they are considered. Note, it is not a hard requirement that your solution use this fourth stack but your input, spur and output must be represented with a stack to receive marks for this part of the assignment.

Sample runs explained

The images for each example given show the three tracks: input, spur and output tracks at the switching station. In your program, these tracks should be represented by three Stack objects.
The numbers on the input track are in random order based on the given input list, and we want the numbers on the output track to be in ascending order, from left to right.
That is, if the output track is represented as a stack, the top should be the smallest number and the bottom should be the biggest number.
Your method is given the initial order the train cars as a `List` of `Integers`

In Example 1 on the next page, the given input list is `{5, 4, 3, 2, 1}`
After the given list is pushed onto the input stack, it will look like this:

1 ← top of the stack
2
3
4
5 ← bottom of the stack

Therefore, in the image in Example 1, what you see on the input track (input stack) is:

where the value at the top of the stack is **1**

Your algorithm should attempt to reorder these numbers using only the spur and the output.

In this example

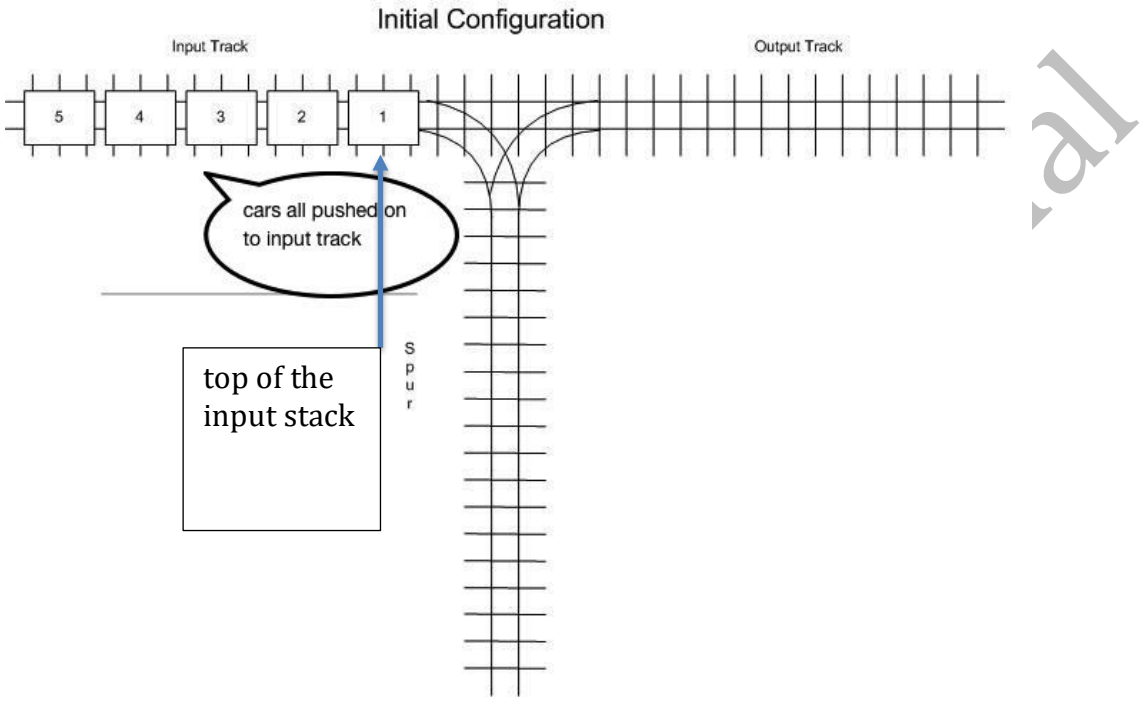
- the input stack contains the values shown above
- the spur and output tracks are empty.
- since we know there are 5 values in the input stack we know the 5 needs to be pushed to the output stack first.
- 1 is at the top of the input, it cannot go to the output so we push this 1 to the spur: in -> spur
- 2 is at the top of the input, it cannot go to the output so we push this 2 to the spur: in -> spur
- 3 is at the top of the input, it cannot go to the output so we push this 3 to the spur: in -> spur
- 4 is at the top of the input, it cannot go to the output so we push this 4 to the spur: in -> spur
- 5 is at the top of the input, it can go to the output so we push this 5 to the output: in -> out
- 4 is at the top of the spur, it can go to the output so we push this 4 to the output: spur -> out
- 3 is at the top of the spur, it can go to the output so we push this 3 to the output: spur -> out
- 2 is at the top of the spur, it can go to the output so we push this 2 to the output: spur -> out
- 1 is at the top of the spur, it can go to the output so we push this 1 to the output: spur -> out
- The input and spur are both empty and since all the values have been pushed to the output track the method should return `true` and the result list `moves` should contain the ordered list of `CarMove` instances that make the reordering of the train cars possible:
  { in -> spur, in -> spur, in -> spur, in -> spur, in -> out, spur -> out, spur -> out, spur -> out, spur -> out}

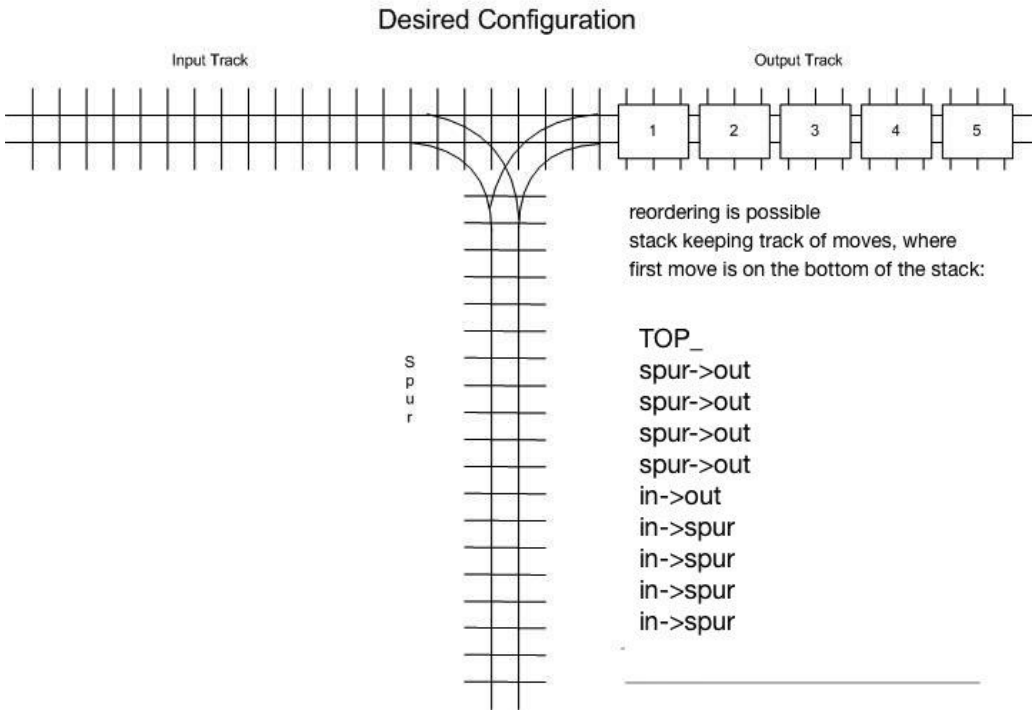
Example 1

incoming carOrder List is {5,4,3,2,1}

In the image given below, the attempted CarMove operations are stored in an additional temporary stack, where the bottom of the stack holds the first attempted move and the top of the stack holds the last attempted move.

Since the reordering is possible, the attempted CarMove operations should be taken from the stack and added to the moves result List and the method should return true.





In Example 2 on the next page, the given input list is `{1, 3, 5, 4, 2}`
After the given list is pushed onto the input stack, it will look like this:

2 ← top of the stack
4
5
3
1 ← bottom of the stack

Therefore, in the image in Example 2, what you see on the input track (input stack) is:

where the value at the top of the stack is **2**

Your algorithm should attempt to reorder these numbers using only the spur and the output.

In this example

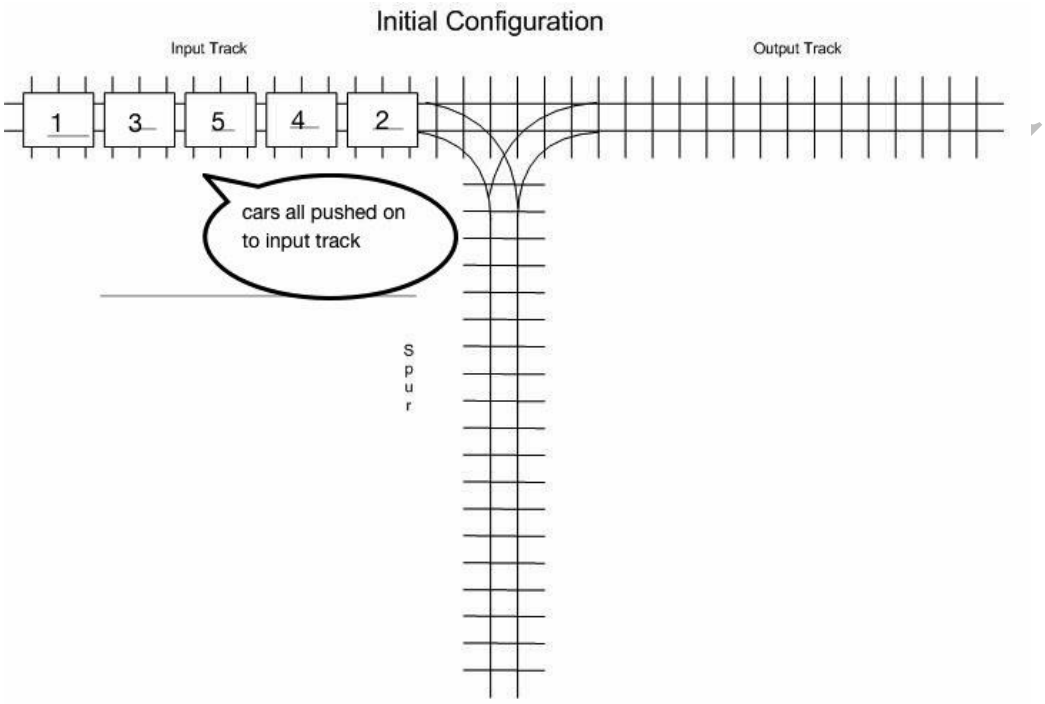
- the input stack contains the values shown above
- the spur and output tracks are empty
- since we know there are 5 values in the input stack we know the 5 needs to be pushed to the output stack first.
- 2 is at the top of the input, it cannot go to the output so we push this 2 to the spur: in -> spur
- 4 is at the top of the input, it cannot go to the output so we push this 4 to the spur: in -> spur
- 5 is at the top of the input, it can go to the output so we push this 5 to the output: in -> out
- 4 is at the top of the spur, it can go to the output so we push this 4 to the output: spur -> out
- 3 is at the top of the input, it can go to the output so we push this 3 to the output: in -> out
- 2 is at the top of the spur, it can go to the output so we push this 2 to the output: spur -> out
- 1 is at the top of the input, it can go to the output so we push this 1 to the output: in -> out
- The input and spur are both empty and since all the values have been pushed to the output track the method should return `true` and the result list `moves` should contain the ordered list of `CarMove` instances that make the reordering of the train cars possible:
  { in -> spur, in -> spur, in -> out, spur -> out, in -> out, spur -> out, in -> out}

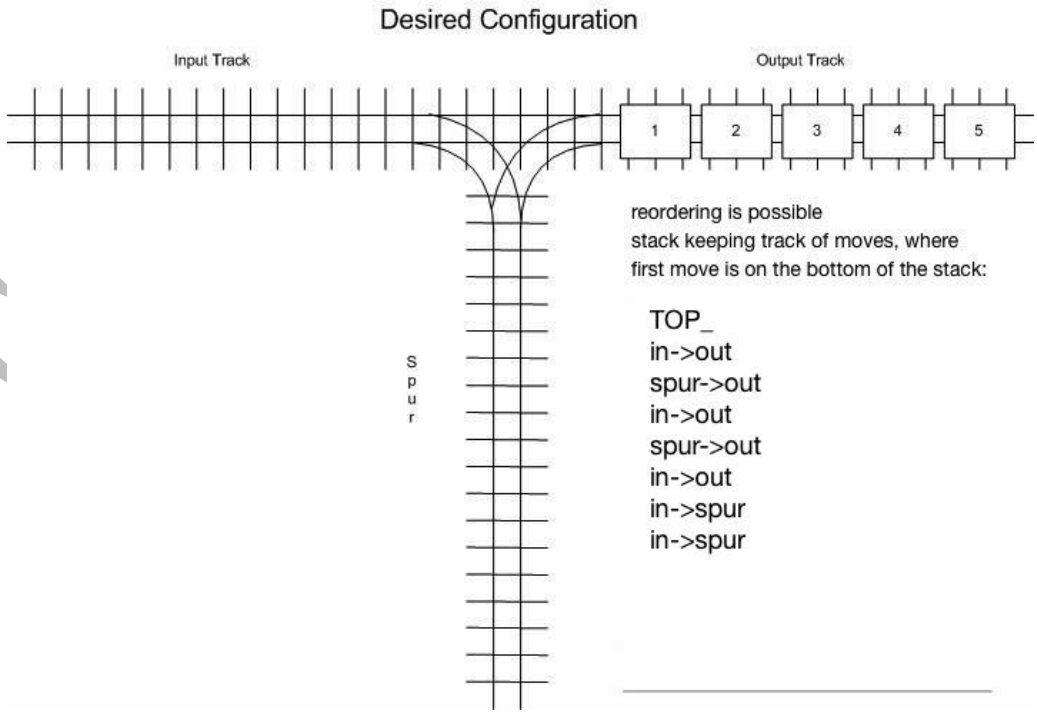
Example 2

incoming carOrder List is {1,3,5,4,2}

In the image given below, the attempted CarMove operations are stored in an additional temporary stack, where the bottom of the stack holds the first attempted move and the top of the stack holds the last attempted move.

Since the reordering is possible, the attempted CarMove operations should be taken from the stack and added to the moves result List and the method should return true.





In the Example 3 on the next page, the given input list is `{4,3,5,1,2}`
After the given list is pushed onto the input stack, it will look like this:

2 ← top of the stack
1
5
3
4 ← bottom of the stack

Therefore, in the image in Example 3, what you see on the input track (input stack) is:

where the value at the top of the stack is **2**

Your algorithm should attempt to reorder these numbers using only the spur and the output.

In this example

- the input stack contains the values shown above
- the spur and output tracks are empty
- since we know there are 5 values in the input stack we know the 5 needs to be pushed to the output stack first.
- 2 is at the top of the input, it cannot go to the output so we push this 2 to the spur: in -> spur
- 1 is at the top of the input, it cannot go to the output so we push this 1 to the spur: in -> spur
- 5 is at the top of the input, it can go to the output so we push this 5 to the output: in -> out
- 3 is at the top of the input, it cannot go to the output so we push this 3 to the spur: in -> spur
- 4 is at the top of the input, it can go to the output so we push this 4 to the output: in -> out
- 3 is at the top of the spur, it can go to the output so we push this 3 to the output: spur -> out
- 1 is at the top of the spur, it cannot go to the output and it is not allowed to go back to the input
- since the input is empty there are no other possible moves
- since all the values have not been pushed to the output track the method should return `false` and the `moves` result list remains unchanged.

Example 3

incoming carOrder List is {4,3,5,1,2}

In the image given below, the attempted CarMove operations are stored in an additional temporary stack, where the bottom of the stack holds the first attempted move and the top of the stack holds the last attempted move.

Since the reordering is not possible, the moves result List remains unchanged and the method should return false.