Assignment 4

Due: November 15th, 2015 before 11:59pm

Objectives

- Introduction to stack containers
- Introduction to exceptions
- More practice with reference-based data structures and templates
- Exposure to applications of stacks

Introduction

This assignment uses stack containers (both list-based and vector-based)

Quick Start:

1. Read this entire document
2. Download all of the A4 source files from conneX.
3. Complete Part I:
   (a) Read the `vector_stack.h` file carefully. It contains a complete implementation of a templated, vector-based stack, with exceptions used in error conditions, such as invalid pop operations in an empty stack.
   (b) Compile and run the vector stack tester with the following commands:
   Compile: `g++ -Wall vector_stack_tester.cpp -o vector_stack_tester`
   Run: `./vector_stack_tester`
   (c) Read the `sll_stack.h` file carefully. It contains empty methods for a (non-templated) list-based stack.
   (d) Change the `node` and `sll_stack` classes in `sll_stack.h` to use templates.
   (e) Compile and run the list stack tester with the following commands:
   Compile: `g++ -Wall sll_stack_tester.cpp -o sll_stack_tester`
   Run: `./sll_stack_tester`
   (f) If all tests pass, move on to Part II.
   (g) If a test fails, fix the implementation of the tested function and go back to (e).
4. Complete Part II:
   (a) Read the `part2.h` and `part2.cpp` files carefully. You are not permitted to change any aspect of `part2.h` in your submission.

(b) Compile and run the part 2 tester with the following commands:

Compile: g++ -Wall part2.cpp part2_tester.cpp -o part2
Run: ./part2

(c) If a test fails, fix the implementation of the tested function and go back to (b).

Part I – Stack Implementation

The `vector_stack.h` file contains a complete implementation of a vector-based stack. Your task in Part I is to implement the singly-linked-list based stack in `sll_stack.h` such that all of the tests in the Part 1 tester pass. The comments in the `sll_stack.h` file describe the purpose and specification of each method. When your implementation is complete and correct, the output of the Part 1 tester will be the following.

```
test_push_empty_size passed.
test_pop_all_elements passed.
test_peek passed.
exceptions working for pop
exceptions working for peek
Passed: 4
```

Part II – Two stack-based exercises

Part II involves implementing two algorithms using stacks.

Problem 1: Bracket Matching

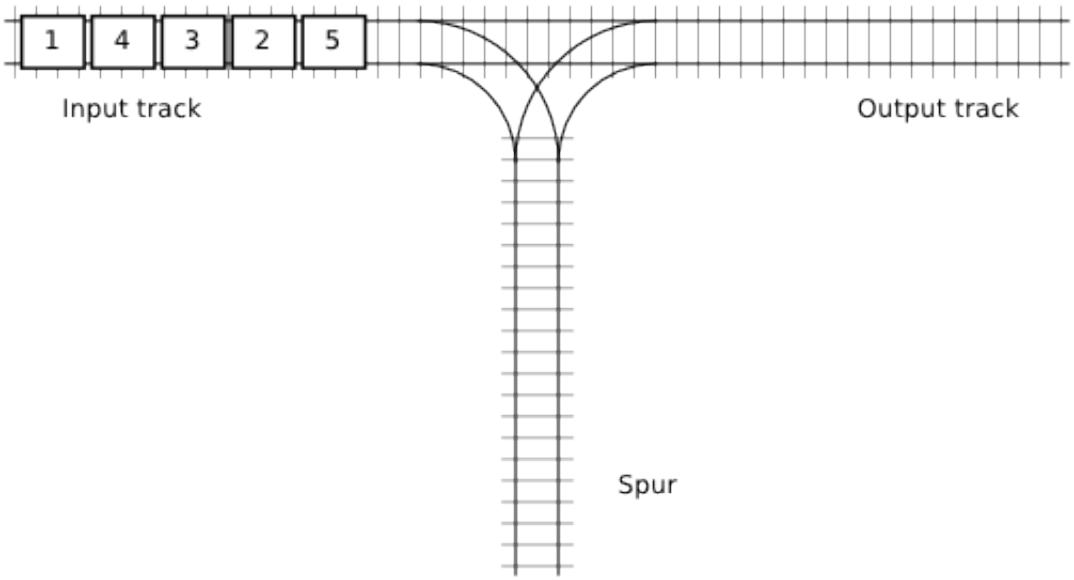
The first problem is checking whether parentheses and brackets in a string of text are balanced. We will use the following types of brackets: ( ), < >, [ ] and { }. For example, the string "(x+ [5*y – {2/z}] + 6)" has balanced brackets, as does the string "3 + 4 + (5*{2-(10-6)}+100)", since every opening bracket has a matching closing bracket, and brackets are properly nested. The string "(3 + x" is not balanced, since a closing bracket is missing; the string "x + (y/3]" is not balanced since the opening and closing brackets do not match; and the string "x + (y + [z-6) + 10]" is not balanced since the brackets are not correctly nested.

One possible algorithm to decide whether a string of text has balanced brackets is given in the pseudocode on the next page.

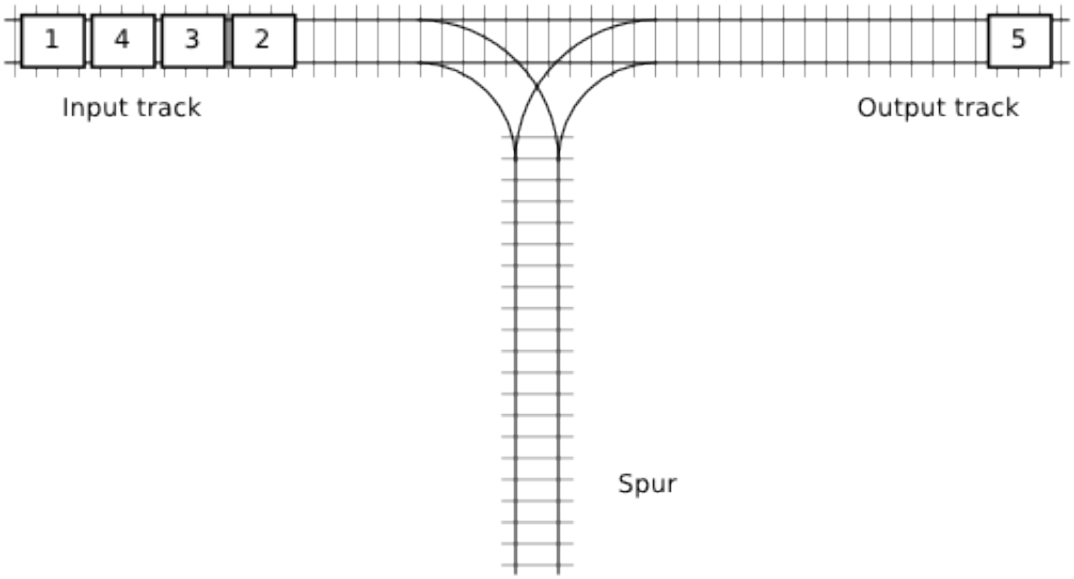
```
Create an empty stack
For each character in the input string:
     If the character is an opening bracket:
          push the character onto the stack.
     If the character is a closing bracket:
          If the stack is empty, return false.
          Otherwise, pop the top element off the stack.
          If the top element is not the matching opening bracket
          for the current character of the string, return false.
After the loop above finishes, if the stack is empty, return true.
Otherwise, return false.
```

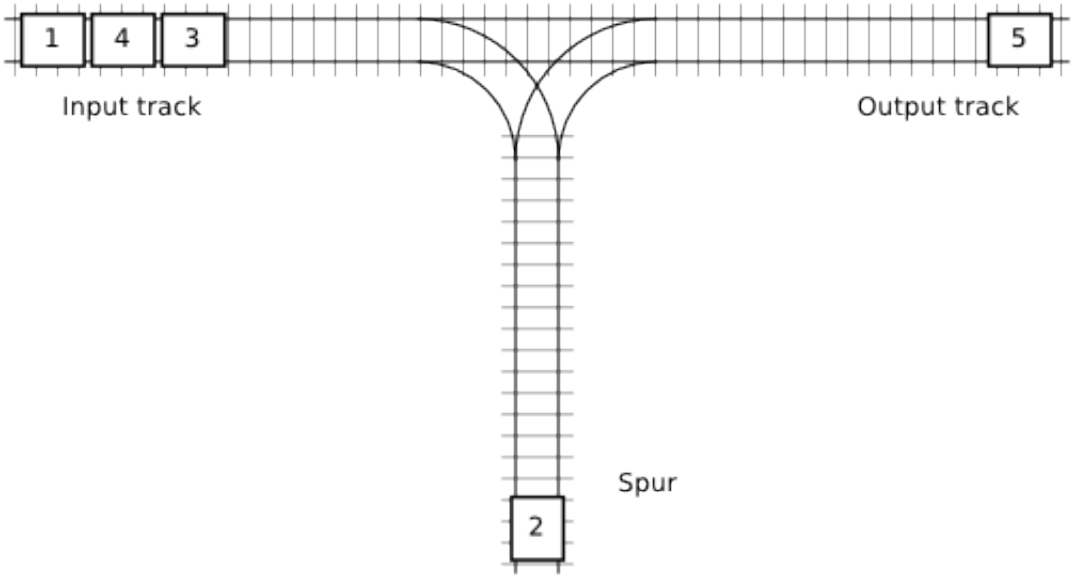
Problem 2: Knuth's train-switching yard

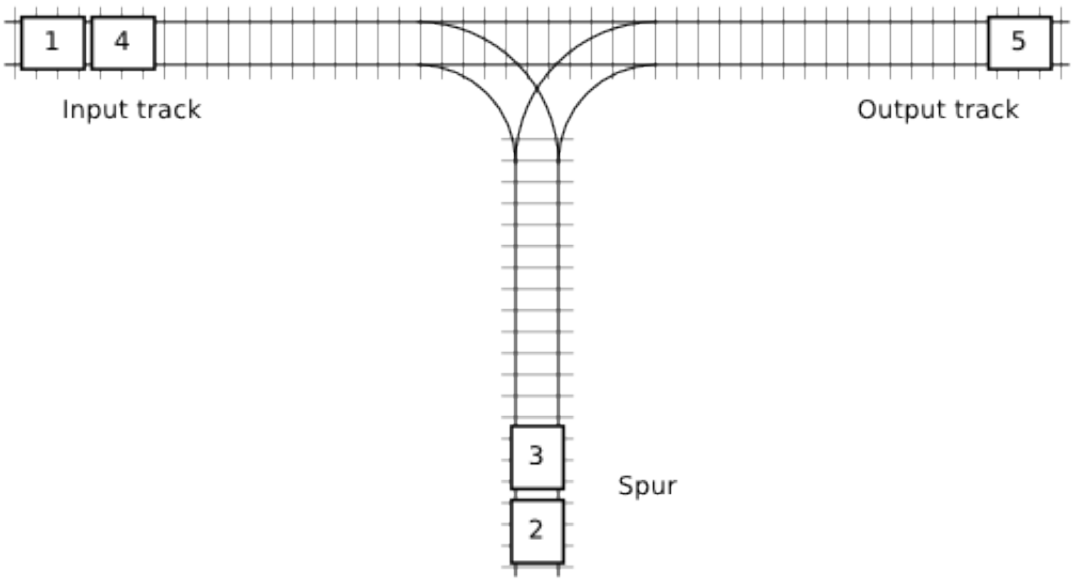
The second problem is from the classic Computer Science textbook *The Art of Computer Programming* by D. E. Knuth (appearing as an exercise in section 2.2.1). Consider a train with N cars, numbered 1 through N. Sometimes, the order of cars within the train needs to be changed. This is accomplished in a rail-switching yard using an extra track called a *spur*, as in the diagram below.

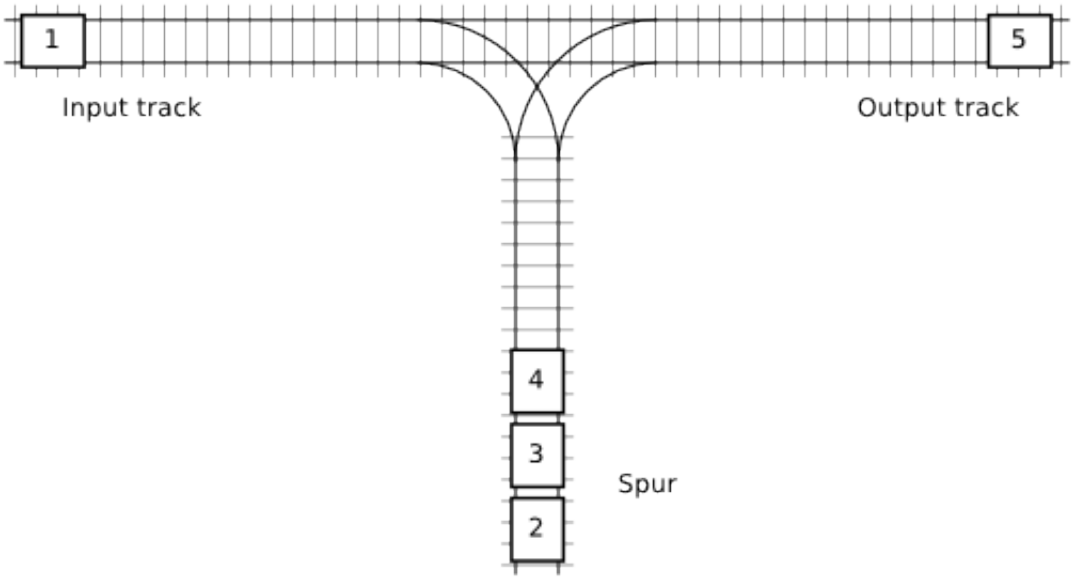


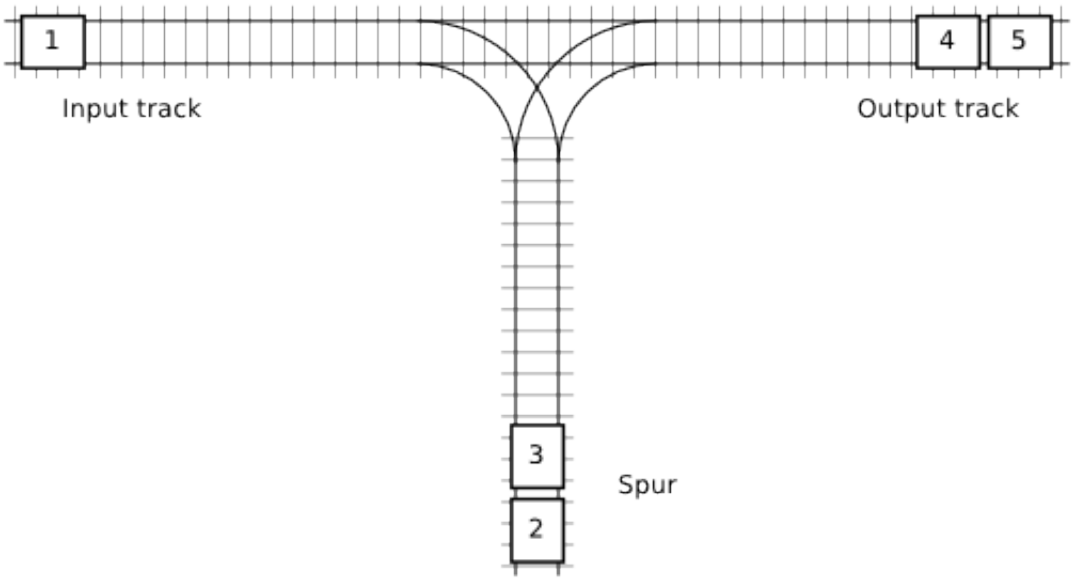
The goal of this problem is to move all cars from the input track to the output track such that the cars are in ascending order (with the highest numbered car is on the right). Cars on the input track can be moved to either the output track or the spur, and cars on the spur can be moved to the output track. Cars cannot be moved back onto the input track from either of the other tracks. Starting from the initial configuration above, the following sequences of moves would arrange the cars into sorted order:

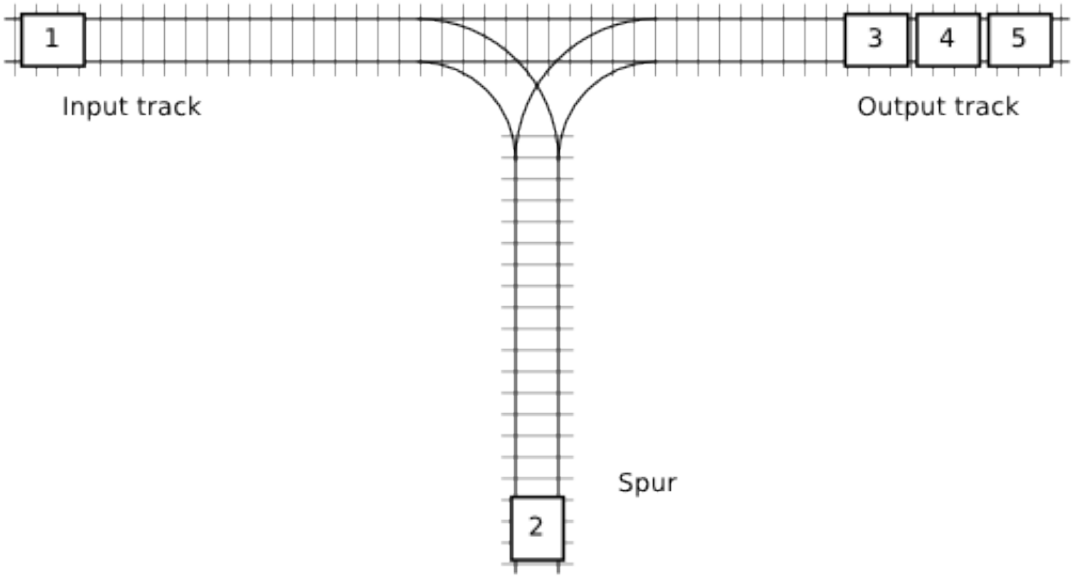


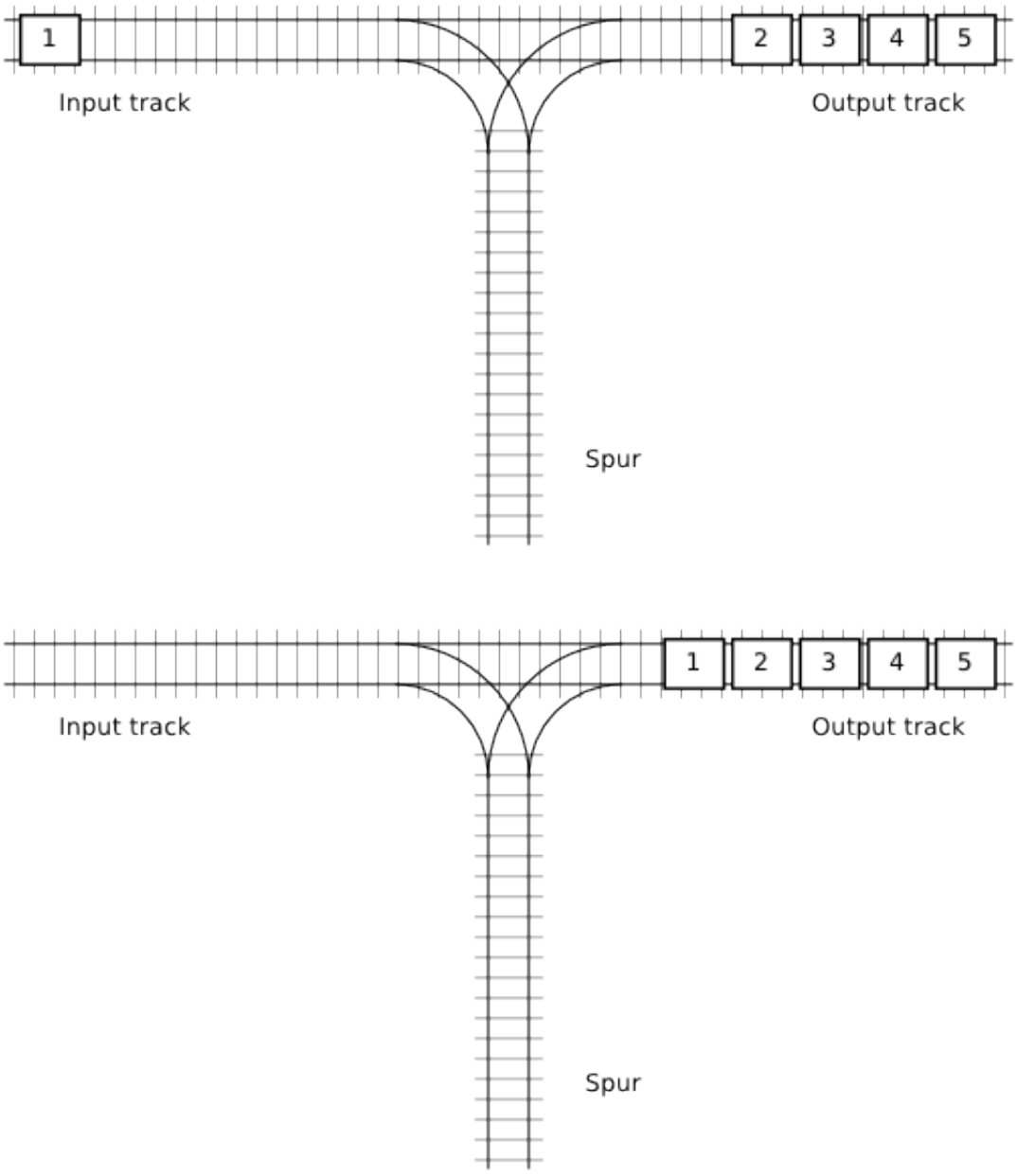



1
4
5
Input track
Output track
3
Spur
2


1
5
Input track
Output track
4
3
Spur
2


1
4
5
Input track
Output track
3
Spur
2

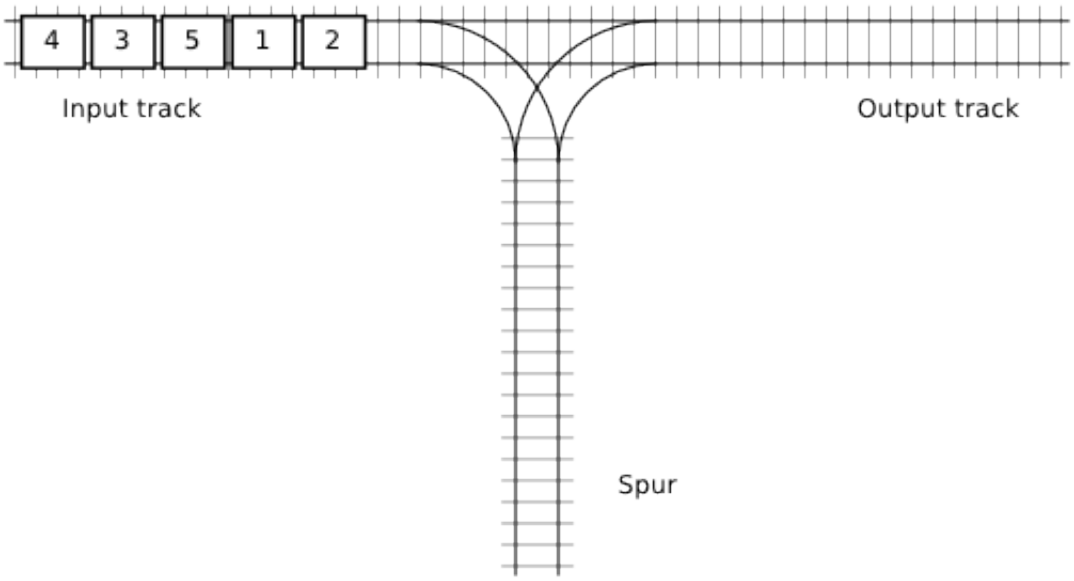

1
3
4
5
Input track
Output track
Spur
2



Each of the three tracks can be represented by a stack, with the bottom of each stack corresponding to the outside edge of the diagram. For the initial configuration [1, 4, 3, 2, 5] in the example above, the stack representing the input track would initially have element 1 at the bottom and element 5 on top. In the final configuration shown in the last diagram, the stack representing the output track would have element 5 on top and element 1 on the bottom. Cars can be moved between tracks with push and pop operations on the corresponding stacks.

Your task is to implement an algorithm to determine whether a given input sequence of train cars can be reordered into ascending order, and if such a reordering is possible, to list the sequence of moves

necessary to achieve it. Your algorithm should begin by pushing every element of the input sequence onto the stack corresponding to the input track. The comments in the `part2.cpp` file describe the expected format of the algorithm's input and output. Note that for some configurations, like the one pictured below, reordering is not possible.



After your implementation for both problems is complete, the output of the Part 2 tester will be the following.

```
test_bracket_matching_one_type passed.
test_bracket_matching_multiple_types passed.
test_railway_1 passed.
test_railway_2 passed.
Passed: 4
```

Submission

Submit your `sll_stack.h` and `part2.cpp` using conneX.

As usual, it is acceptable (and encouraged) for you to talk about your assignment with your classmates, and you are encouraged to design solutions together, but each student must implement their own solution. Plagiarism detection software will be run on all assignment submissions.

Grading

If you submit something that does not compile, you will receive a grade of 0 for the assignment. It is your responsibility to make sure you submit the correct files.

Part I

Requirement	Marks
Your submission compiles	1
Your code passes the test cases in `sll_stack_tester.cpp`	4

Part II

Requirement	Marks
Your submission compiles	1
Your code passes the test cases in `part2_tester.cpp`	4

Total 10