6 Circular Data; State Change

Practice Problems

Practice problems help you get started, if some of the lab and lecture material is not clear. You are not required to do these problems, but make sure you understand how you would solve them. Solving them on paper is a great preparation for the exams.

1. Design the classes that represent the transit system that consists of several train lines, with stations on each line.

   Every line has a name (usually a color name) and a list of stations it serves. Every station has a name and a list of lines that go through the station.

   (a) Make an example of a transit system that looks like the MBTA *Green Line, Red Line Blue Line, and Orange Line* and include two terminal stops and at least two transit stations for each line.

   (b) Design the method `isTransfer` that determines whether a station is a transfer station between one or more lines.

   (c) Design the method `sameLine` in the class `Station` that determines whether this station is on the same line as the given station.

   (d) Design the method `oneChange` that determines whether we can travel from this station to the given station making exactly one change at a transfer station.

Pair Programming Assignment

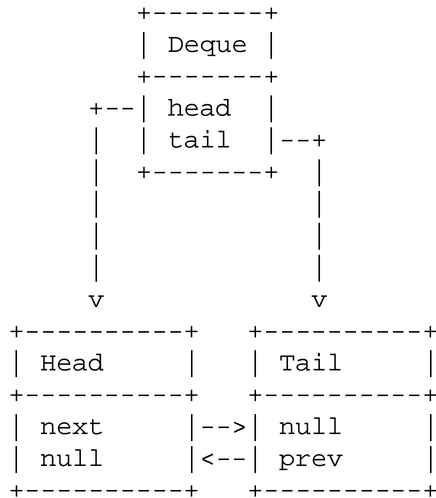
6.1 Problem

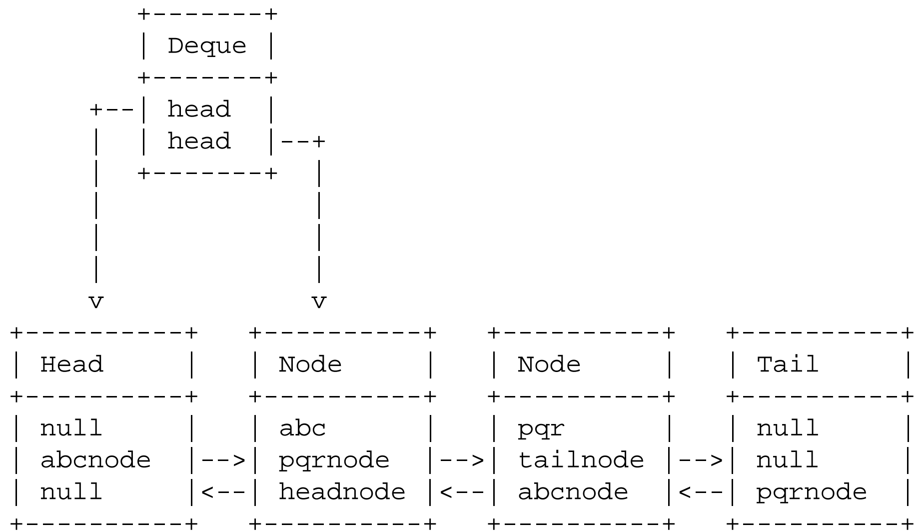
Complete problems 6.1 and 6.2 from Lab 6 and turn in the complete solution.

6.2 Problem

In this problem we extend what we learned in problem 7.3 from Lab 7. We would like to build a list in such a way that we can start either at the front,

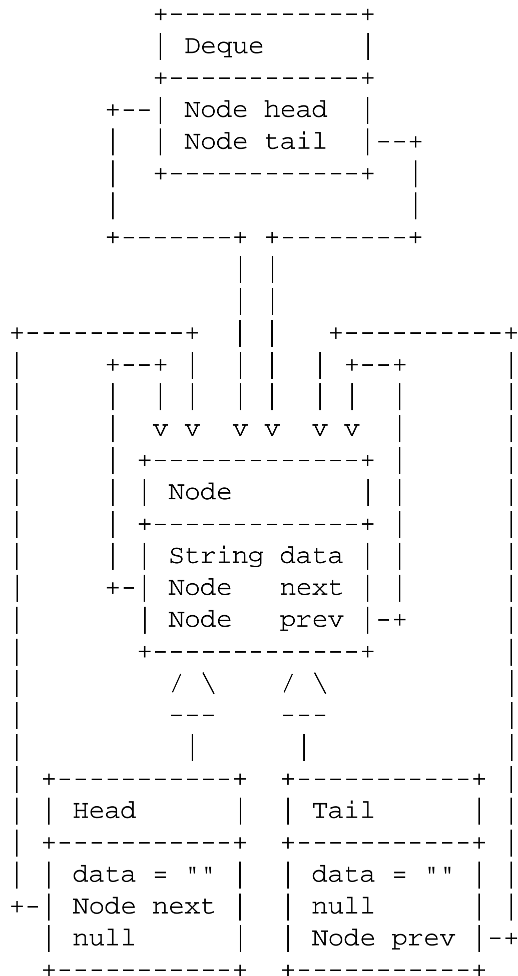
or at the end, and move across the list in either direction. To do so, we have decided on the following structure:





Instead of one *sentinel* node, we have two of them, one marking the head of the queue and the other marking the tail of the queue. Additionally, we have a new field in each node, a reference to the previous item in the list.

The class diagram would then be:



A. Define the classes `Node`, `Head`, `Tail`, and `Deque` that implement the *doubly-linked* list of `String`s. Use the code from Lab 6 as your model.

B. Make examples of three lists: the empty list, a list with the two values shown in the drawing at the beginning of this problem, and a list with three values, ordered lexicographically from the head to the tail.

C. Design the method `size` that counts the number of nodes in a list (`Deque`), not including the two sentinel nodes (the head and the tail).

D. Design the method `addAtHead` for the class `Deque` that consumes a `String` and inserts it at the *head* of this list.

E. Design the method `addAtTail` for the class `Deque` that consumes a `String` and inserts it at the *tail* of this list.

F. Design the method `removeFromHead` for the class `Deque` that removes the first node from this `Deque`. Besides this *effect* it produces the `String` that was in the removed `Node`.

   Throw an exception, if an attempt is made to remove from an empty list.

G. Design the method `removeFromTail` for the class `Deque` that removes the last node from this `Deque`. Besides this *effect* it produces the `String` that was in the removed `Node`.

   Throw an exception, if an attempt is made to remove from an empty list.

H. Design the method `insertSorted` for the class `Deque` that consumes a `String` and inserts it into this sorted list in correct order.

I. Design the method `removeSorted` for the class `Deque` that removes the node that contains the given `String` from this `Deque`.

   Throw an exception, if the there is no such node.

J. Design the method `toLowerCase` for the class `Deque` that changes the `String` in every `Node` to lower case.

   The class `String` defines the method `toLowerCase` that produces a new `String` with all letters changed to lower case.

   Note: Do not use this method with special characters without looking up the formal documentation for the method.