The `SinglyLinkedList<E>` ADT continued

Data Structures & Algorithms @ Dwight-Englewood School

overview

Now that you have implemented the `SinglyLinkedList` ADT and the basic operations that were originally specified, we move on to add some more functionality to this class. Specifically, you will add the following methods to the class (if you have not already done so!):

- `void reverse()`: reverses the list while only using constant memory (in other words, the amount of additional memory used in this method is **not** a function of the length of the list).
- `SinglyLinkedList<E> concat(SinglyLinkedList<E> other)`: returns a new list that results from concatenating this list with list `other`. Leaves this list and the other list unchanged.
- `boolean equals(Object other)`: overrides inherited method from `Object`. Returns `true` if this list and `other` are *equal* (see algorithm below).

additional methods

Here are some algorithms for a few of the methods listed above.

- `concat(SinglyLinkedList<E> other)`: This method creates and returns an entirely new list that is the result of concatenating the elements of `other` onto the end of this list. The algorithm is one that you will derive, although we did sketch it out in class.
- `reverse()`: This method reverses this list (modifies it), and does so using constant additional memory that is not dependent upon the length of our list. The algorithm is one that you will derive, and we also sketched it out in class recently.
- `equals(Object other)`: This method is an override of the method inherited from the `Object` class that takes as input an `Object`. For this class, we want to compare for equality with other singly linked lists. The algorithm:
  i. if `other` is `null`, return `false`.
  ii. if the class of `other` does not equal our class, return `false`. (Here you would use the inherited method `getClass()` to get the class of each object.) Note that the `getClass` method will only tell you if the object reference is that of a `SinglyLinkedList`, *not* what the

datatype is that it holds! The check of equality between individual elements below will take care of that. The algorithm below is the one that we went over in class recently.

iii. create a `SinglyLinkedList` reference called `otherList` and set it to `other` using a cast to a `SinglyLinkedList`.

iv. if the length of this list differs from the length of `otherList`, return `false`

v. create a `Node` reference `A` and set to `head`

vi. create a `Node` reference `B` and set to `otherList.head`

vii. while `A` is not `null` ...

(a) if the element referenced by `A` differs from the element referenced by `B`, return `false`

(b) advance `A` to its next node

(c) advance `B` to its next node

viii. return `true`

For now, create your own client program to test out the new additions. When I want you to turn in any output I will give you a specific client for you to run.

some questions to ponder

Here are some additional questions for you to think about with respect to the `SinglyLinkedList` class:

i. In the class we have a field called `length` that tracks the number of nodes in the list. Suppose we did NOT maintain that field. How would you implement the `size()` method in that case? Write pseudo-code for this approach.

ii. Imagine we wanted to be able to rotate nodes in our list, with a call to the method `rotate()` moving the first node to the end of the list, rather than simply swapping the nodes' data. Write pseudocode that accomplishes this without creating any new nodes.

iii. Describe in detail how to *swap* two nodes `A` and `B` (and not just their contents) in a singly linked list given only references to `A` and `B`.