The `SinglyLinkedList<E>` ADT

Data Structures & Algorithms at Dwight-Englewood School

overview

The `SinglyLinkedList` abstract data type (ADT) is a classic data type, one of the most important in programming history. It is an implementation of the general *linked-list* concept, whereby a collection of *nodes* forms a linear sequence. In the *singly*-linked list, each node stores some data as well as a link to another node, the next one in the list or linear sequence. Linked lists are important and effective because of their highly efficient insertion and deletion behavior, and they are widely used to implement other important data structures, such as stacks, queues, and other ADTs.

In our implementation of this ADT, we will utilize Java's *generic* framework so that our list can contain *any* reference datatype.

The `SinglyLinkedList` class will support the following methods:

i. `size()`: returns the number of elements in the list

ii. `isEmpty()`: returns `true` if the list is empty, and `false` otherwise

iii. `first()`: returns (but does not remove) the first element in the list

iv. `last()`: returns (but does not remove) the last element in the list

v. `addFirst(e)`: adds a new element (`e`) to the front of the list

vi. `addLast(e)`: adds a new element (`e`) to the end of the list

vii. `removeFirst()`: removes and returns the first element in the list

We will also take advantage of Java's nested class support to implement the `Node` class inside of the `SinglyLinkedList` class. We do it this way for several reasons: 1) different types of linked lists might require different types of nodes, and 2) there is no need for users to know anything about nodes per se.

implementation

The `SinglyLinkedList` class will consist the following components:

A. instance fields:

   i. `head`: a node reference to the start of the list
   ii. `tail`: a node reference to the end of the list
   iii. `length`: an `int` representing the size of the list (number of elements)

B. the nested class `Node<E>`, which will itself consist of:

   i. `element`: the actual data contained within the node, of datatype `E`
   ii. `next`: a reference to the *next* node in the list
   iii. a constructor that creates a new node from some data and another node reference
   iv. getters the two fields
   v. a setter for the `next` field

C. methods: the methods specified above, in addition to a `toString` method.

implementating the class

You will receive the Java file `SinglyLinkedList.java`, which contains the class definition specified above. Most of the methods are not yet implemented: you will do this. However, the following elements are already implemented and should not be modified:

- the nested `Node<E>` class
- the instance fields for the `SinglyLinkedList` class are declared and initialized
- the lone, default constructor for the `SinglyLinkedList` is complete.

Each of the remaining methods contains some pseudo-code that you will follow to implement them, as well as a stub solution that allows the class to compile.

You are also getting a class called `SLLClient.java`, which is a file that you may not modify unless directed to do so. You will use it to test out your class. This client program will compile and run when you get it, but until you add the method implementations to the `SinglyLinkedList` class, it won't really do anything.

closeups on individual methods

The constructor:

```
public SinglyLinkedList(){}
```

This method is empty because the fields are already assigned (all 3 actually take on their default values and thus do not need explicit assignment).

The `first()` and `last()` accessor methods:

These two methods simply *access* the first or the last element in the list, which they return, *unless* the list is empty, in which case they return `null`.

The `addFirst(e)` method:

The algorithm here is

1. create a new node containing `e` and pointing to `head` and then point `head` at it. (This is accomplished in one line as shown.)
2. if the length is zero, set the `tail` to the `head`
3. increase the length by one

The `addLast(e)` method:

The algorithm here is

1. create a new node containing `e` and pointing to `null`, since this node will be at the end of the list.
2. if the list is empty, set the `head` to the new node, otherwise set `tail`'s next to the new node
3. set the `tail` to the new node
4. increase length by one

The `removeFirst()` method:

The algorithm here is

1. if the list is empty return `null`
2. create a variable to store the element from `head`'s node
3. advance `head` to its next node
4. decrease length by one
5. if the length is now zero, set `tail` to `null`
6. return the stored element

other methods to implement

Here are some additional methods that we will implement for our class. In some cases an algorithm is provided.

- `String toString()`: returns a `String` representation of the list
- `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).

algorithms for selected 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:
- `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:
- `equals(Object other)`: This method is an override of the method 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.)
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 other.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
```