Lab 3: Doubly-linked Lists

Week of October 15, 2018

Objective

To sort a doubly-linked list using quick sort.

Exercises

The file `Lab3.java` contains a nearly-complete program to insertion sort a doubly-linked list, except it needs TWO methods added to the `List` class (details below).

Note that the file contains THREE classes. It contains:

- A `List` class, which is a doubly-linked list with a `start` and an `end` pointer),
- A `DNode` class, which is a node for a doubly-linked list with an integer item and a forward and a back pointer (it is a `private` class inside the `List` class with `public` instance members), and
- The `Lab3` class containing the `main()` method and the code that tests the sorting method.

Add the two required `List` methods (and any helper methods needed for the two methods) to `Lab3.java` without changing any of the existing code.

The two methods you will write:

1. Method `middleNode()`, which returns a pointer to the middle node in the calling `List`. Note: If the list contains an even number of nodes, then there are two "middle" nodes and you can return a pointer to either one of them.

   For example, if the calling list is `start` → 43 ↔ -19 ↔ 100 ↔ 17 ↔ -93 ← `end`, then `middleNode()` should return a pointer to the node containing 100 (it has an equal number of nodes on either side of it, so it is the middle node). For another example, if the calling list is `start` → 43 ↔ -19 ↔ 100 ↔ 17 ↔ -93 ↔ -13 ← `end`, then `middleNode()` should return a pointer either to the node containing 100 or the node containing 17 (it doesn't matter which one is returned).

   Note: This method is used by method `choosePivot()`, which finds a good pivot using the median-of-three technique. Method `choosePivot()` has already been written for you — you only have to write `middleNode()`.

   Restriction: You can only access each node in the calling list at most ONCE. (If `curr` is a node, then executing `curr.item` or `curr.forward` or `curr.back` accesses the node `curr`.) So you cannot count the nodes and then go to the middle node. Instead, you need to move through the list from BOTH ends at the same time, moving one node closer to the middle from each direction at each step.

   Assumption: You may assume that the list contains at least three nodes.

   Challenge: Recursively find the middle node. You are not required to write this code recursively, but if you find the non-recursive code easy, you should try recursion.

2. Method `partition()`, which is passed one parameter `pivot`, which is a pointer to the node containing the pivot (and this node is not part of any list).

   This method must partition the calling `List` ("`this`") into the smalls (nodes containing items less than the pivot) and the bigs (nodes containing items no smaller than the pivot).

   When the method returns, the bigs should be in the calling `List` ("`this`") and the smalls should be in a new `List`, which should be returned by `partition()`. (The node pointed at by parameter `pivot` should not be in either list.)

   For example, if the calling list is `start` → 43 ↔ -19 ↔ 100 ↔ 17 ↔ -93 ↔ -13 ← `end` and `pivot` points to a node containing 42, then when `partition()` returns, the calling list should contain only the nodes with values 43 and 100 (the order of the two nodes is unimportant) and the new list that `partition()` returns should contain only the nodes with values -19, 17, -93 and -13 (again, the order of the four nodes is unimportant).

   Restriction: You cannot change the value stored in a node and you cannot create new nodes (that is, you cannot type `new DNode(...)`). You can only unlink and link nodes that already exist.

   Assumption: the calling `List` contains at least two nodes.

Note: You can change the constants defined at the beginning of the `Lab3` class, which control the testing that the code does, but don't change any of the already-written code.