Junyu Teoh
CSCD 300 Data Structures
HW2 Progress Report

1. **void addLast(Object data)**

Step 1) We stat off with an object cur with type Node, and assign this.head to cur.

Step 2) Now we have to increment to find the end of the Circular Linked List. I used a for loop with int i = 0, and end at i < this.size. However, we can also use i !=this.head, as circular linked lists will loop around after you hit the end.

Step 3) Since we know we are at the end, we can now add the new data. A new node nn is created. Nn's data is the data parameter of the function, nn's previous is cur, as cur is at the "end" of the loop, and nn's next is cur.next, which loops back to the head.

Step 4) In order to link Node nn to the linked list, we change cur.next.prev (head, in this case) to nn, and cur.next to nn. This basically slots nn between cur and head, which is the end of the loop.

Step 5) Increment size by 1, as we've added 1 new node.

2. **CDoublyLinkedList subListOfSmallerValues(Comparable data)**

Step 1) A new CDoublyLinkedList is created with the name toReturn. Currently it only has a dummy head, with no nodes.

Step 2) A node is created with the name cur. This.head is assigned to cur.

Step 3) In order to move through the linked list, we use a for loop. Here I used a int i = 0, and end at i < this.size, incrementing i by 1 each time.

Step 4) Everytime the loop runs, we move cur forward with cur = cur.next. Then, we check if data.compareTo(cur.data) > 0 is true, as we want to know when cur.data is smaller than the comparable data that was passed in. If it is, we add a new node to toReturn with the cur's current data with toReturn.addLast(cur.data).

Step 5) Return CDoublyLinkedList toReturn.

3. **boolean removeStartingAtBack(Object dataToRemove)**

Step 1) A new Node named cur is created, this.head is assigned to cur. A Boolean named toReturn is also created with false as the default value.

Step 2) In order to find the end of the Circular Linked List, we use a for loop. We start with i = 0, and increment by 1 till i < this.size is true.

Step 3) Each time the loop runs, we step forward with cur=cur.next. Then, we check if cur.data is equal to dataToRemove **AND** if toReturn is false.

Step 4) If the above requirements are satisfied, we then remove the node by doing cur.next.prev = cur.prev, cur.prev.next=cur.next, reduce the size with this.size-- and change toReturn to true, to prevent this loop from removing possibly another node with the same data. Like the previous method, we essentially cut the node from inbetween the previous and next node.

Step 5) return toReturn. Notice that if nothing is removed, toReturn stays false, which is what we want. It only returns true when something is removed.

4. int lastIndexOf(Object o)

Step 1) Node named cur is created, this.head is assigned to it.

Step 2) An int named index is created with -1 as the value. This will be returned later.

Step 3) In order to check the whole linked list, a for loop is created. As usual, we start with int i = 0, increment i by 1 until i < this.size

Step 4) We move forward in the linked list with cur = cur.next.

Step 5) if cur.data is equals to o (the object passed in), we assign i to index. Notice that the counter i will be at whichever index cur is currently at.

Step 6) return index.

5. boolean retainAll(CDoublyLinkedList other)

Step 1) Check if the linkedlist that's passed in is empty with other.size == 0. If true, throw nullpointerexception.

Step 2) Otherwise, create a Boolean named toDelete. We assume that the current node we're looking at in THIS has no duplicates in other. Another Boolean named anyDeleted is created with false. This is to make sure we return the right Boolean flag.

Step 3) 2 nodes are created. thisCur = this.head.next and otherCur = other.head.next. These are "pointers" to the "current" nodes in their respective linkedlists.

Step 4) In order to check through all the nodes in THIS, a for loop is created. It will run until thisCur!=head and we increment with thisCur.next, that was thisCur will step forward with each run.

Step 5) Now we have to check the other linked list. Another for loop is created, that will run until otherCur!=head, and increment with otherCur = otherCur.next.

Step 6) Now we check if the data matches with thisCur.data == otherCur.data. If they match, we change toDelete to false, which would prevent the node from getting cut out. Basically, if we find ANY node in other that matches the current cur node, we don't delete it.

Step 7) If it doesn't match anything from other linkedlist, we delete the node from THIS list with thisCur.next.prev = thisCur.prev, thisCur.prev.next = this.Cur.next and finally decrement the size by 1 with this.size --. We also change anyDeleted to true, so we can return the right result later.

Step 8) After we delete it, we reset toDelete to true. Otherwise, the previous if statement will still be true, and it will end up deleting everything. Also, this helps cement the idea that we assume all nodes in THIS will be removed, UNTIL we find a match in other.

Step 9) Return toDelete.

6. void insertionSort()

Step 1) We create a Node named lastSorted, it gets this.head.next. We then make another comparable named FUD (first unsorted data).

Step 2) We imagine that the linked list is split into two. One is the "Sorted" part (a linked list a non-dummy node is considered sorted), and the other being "Unsorted".

Step 3) We point a node to the first unsorted data (FUD) and last sorted data (LSD).

Step 4) We compare if LSD is bigger than FUD. If it is, we move lastSorted node downwards in the sorted region until LSD is smaller than FUD

Step 5) Swap LSD with FUD. We have to modify the .prev and .next elements of neighbouring nodes.