Data Structures and Objects
CSIS 3700

Lab 9 — Circular Doubly Linked Lists

Goal

Modify the templated `LinearList` class to use circular doubly linked lists with additional capabilities.

Motivation

The initial version of the `LinearList` class has a sound design. It provides basic list functions and does so reasonably efficiently.

However, there are some capabilities that the class lacks:

- Using negative indexes to count from the "right" end of the list
- Providing rapid access to a "current" node
- Moving the current node and checking on the location of the current node

This exercise will add these capabilities to the existing class.

Details

Begin with our current `LinearList` template. The first step is to change all occurrences of `LinearList` to `CDList`, and change all occurrences of `ListNode` to `CDListNode`. In CLion, you can click on any occurrence and press shift-F6, then type in the new name; this will change all occurrences.

The next step is to add one additional pointer to the `CDListNode` structure and one additional pointer to the `CDList` class. We must also change the count from unsigned to signed, to allow for negative indexing.

```
template <typename ListType>
struct CDListNode {
    ListType
        datum;
    CDListNode<ListType>
        *next,
        *prev;                  // add this
};
```

```
template <typename ListType>
class CDList {
    ⋮
private:
    CDListNode<ListType>
        *head,
        *cur;                         // add this
    int32_t                           // change this from unsigned
        count;
};
```

▹Constructors

There are two constructors — one default constructor that creates an empty list, and one "copy" constructor that makes a (shallow) copy of an existing `CDList` object.

For both, set the current node pointer to `nullptr`.

Next, we need to modify the copy constructor. Initially, the constructor creates a singly linked list. We will need to add two things to the list:

- A next pointer from the last node to the head node
- All of the back pointers from one node to the previous node

Add this code to the copy constructor, immediately after the end of the for loop.

```
        // point last node to first node
        ptr->next = head;

        // set all of the back pointers
        prev = head;
        for (uint32_t i=0;i<l.count;i++) {
            ptr = prev->next;
            ptr->prev = prev;
        }
```

▹Clearing the List

In the `clear()` method, set `cur = nullptr` when you reset `head` and `count`.

▹Searching the List

In the `search()` method, if the key is found, set `cur = tmp` before returning the position.

▹Accessing list elements

With the back pointers, it is now feasible to enable negative indexing. This requires two modifications. First, in the bounds check, 0 should be `-count`. Second, change the for loop to the following:

```
    if (pos < 0)
        for (int32_t i=0;i>-pos;i--)
            tmp = tmp->prev;
    else
        for (int32_t i=0;i<pos;i++)
            tmp = tmp->next;
```

▹Inserting a new node

Inserting a node into the list is almost identical to insertion into a regular linked list, with two exceptions:

- There is a special case to consider before going through the steps
- You need to set up both forward and backward pointers

The special case is when you are inserting into an empty list. Add this code to the end of step 1 of the `insert()` method:

```
        if (count == 0) {
            // make the node point to itself forward and backward
            ptr->next = ptr->prev = ptr;

            // set the head pointer
            head = ptr;

            // update the count
            count++;

            // done
            return;
        }
```

For the general case, we have to update steps 2, 3 and 4 to set up the backward links. Begin by adding a pointer `successor` at the top of the method, after the `ptr` declaration. Note that it has the same type as `ptr`.

At the end of step 2, add the followng line:

```
        // find the node after the new node
        successor = *pred;
```

At the end of step 3, add this line:

```
        // new node points backward to predecessor
        ptr->prev = successor->prev;
```

At the end of step 4, add this line:

```
        // successor points to new node
        successor->prev = ptr;
```

▹Removing a node

Removing a node, like insertion, is almost identical to removal from a regular linked list, with two exceptions:

- There is a special case to consider
- You need to update both forward and backward pointers

The special case is when you are removing the only node in a list. Add this code to the end of step 0 of the `remove()` method:

```
        if (count == 1) {
            // remove the node
            delete head;

            // reset head and cur
            head = cur = nullptr;

            // update the count
            count--;

            // done
            return;
        }
```

For the general case, we have to update steps 2 and 3 to set up the backward links. Begin by adding a pointer `successor` at the top of the method, like you did with the `insert()` method.

At the end of step 2, add the followng lines:

```
        // find the node after the node to be deleted
        successor = ptr->next;

        // if we are removing the current node, set cur to nullptr
        if (cur == ptr)
            cur = nullptr;
```

At the end of step 3, add this line:

```
        // successor points backward to predecessor
        successor->prev = ptr->prev;
```

▹Current Node Features

The preceding section explains how to modify the existing LinearList methods to use the backward links. This section explains the "current" node features. These allow the user to designate one node as "current" with the ability to quickly access that node's data and to move the current node designation to other nodes within the list.

▹*Setting Current to First or Last*

You can designate either the first node or last node as the current node. The `first()` and `last()` methods do this. Both should check the count; if the list is empty, both methods should throw an exception (I use `runtime_error`). Otherwise, set the current node pointer `cur` to either the head of the list or the head node's predecessor (`head->prev`).

For both of these methods, you should return a reference to the datum that the current node is pointing to.

▹*Checking If Current Node is First or Last*

We can check to see if the current node is set to the first or last node in the list with the `isFirst()` and `isLast()` methods. For both, check to see if `cur == nullptr`; if so, return false. Otherwise, check to see if the current node pointer equals the head pointer (for `isFirst()`), or the head node's predecessor (for `isLast()`).

▹*Moving the Current Pointer*

We can move the current pointer to either the next node in the list or the previous node in the list, provided that the current node pointer is pointing to a node and isn't set to `nullptr`.

The `next()` and `prev()` methods should throw an exception (again, use `runtime_error`) if the current node is `nullptr`; otherwise, move the pointer to the next or previous node and return a reference to the datum.

▹*Accessing the Current Datum*

The last part of the current node functionality is accessing the datum pointed to by the current node pointer.

The `current()` method should throw an exception (`runtime_error`) if the current node pointer is set to `nullptr`; otherwise, it should return a reference to the current datum. Returning a reference allows the program to read and write the datum as needed.

Testing

Fix any compilation errors that may arise. Run the program. Fix any failures that may be reported or any crashes that occur. Repeat until all tests pass.

What to turn in

All you need to turn in is your CDlist.h file.