Wesleyan University, Fall 2021, COMP 211
Homework 8: Linked lists
Due by 11:59pm on Novemember 16, 2021

---

1. Written problems (5 points)

Problem 1. *In Problem 2 you will implement a function to perform a linearity check on a linked-list: if the linked-list is linear, the function returns true, otherwise the function returns false.*

In this problem, you will analyze the cost of the linearity check you implemented. If the number of nodes in the linked list is `n`*, give the cost of your linearity check algorithm in terms of* `n` *and explain how you computed that cost. You must include your code for Problem 2 in your write-up as well as in* `hw8.c`*. This will greatly facilitate our checking of your cost computation.*

2. Programming problems (15 points)

Problem 2. *In this problem you will implement a linearity test to check that a linked-list is linear (that is, traversing* `next` *fields results no loops and eventually gets to a* `NULL` *node). Such a test is important for checking the correctness of functions: any operations performed on a linked-list should maintain the linearity of the linked-list.*

Your function will take the head of a linked-list as input, and return true if the linked-list is linear and return false if the linked-list is non-linear. By non-linear, we mean that as you loop through the linked-list, eventually one of the nodes visited is a node that has already been visited before. The linearity function has been specified for you in `hw8.h`*. I have already specified a linked-list node type for you (*`struct qnode`*) in* `hw8.h`*.*

Hint 1: You can compare two variables of type pointer to check for equality. If the values of the pointers (i.e. addresses contained in them) are the same, then the pointers really do contain the same memory address and the `==` *operator will return true.*

Hint 2: A straightforward algorithm to check linearity compares each node to all of the nodes preceding it. There is also a less straightforward but more efficient algorithm in which two pointers both traverse the list, one pointer traversing the list twice as fast as the other pointer. If either pointer ever reaches the end of the list then the list is linear. If the two pointers ever end up on the same node, then the list is not linear.

Problem 3. *Use a linked-list to implement a queue (an ordinary queue, not a priority queue). In a bit more detail, you will implement the following:*

- `qnode` *structure: an individual element in the queue. Note that this* `struct` *has already been implemented for you.*
- `queue` *structure: a structure with a* `qnode` *field that represents the linked-list that represents the queue, along with any other fields you need.*
- `create` *function: initialize a new queue.*

```
$ ./driver
(0) Exit
(1) Create queue
(2) Enqueue character
(3) Dequeue character
(4) Print queue
Enter choice: 1
Queue contents:
(0) Exit
(1) Create queue
(2) Enqueue character
(3) Dequeue character
(4) Print queue
Enter choice: 2
Enter character: a
Queue contents:
Q: a
(0) Exit
(1) Create queue
(2) Enqueue character
(3) Dequeue character
(4) Print queue
Enter choice: 2
Enter character: b
Queue contents:
Q: a b
(0) Exit
(1) Create queue
(2) Enqueue character
(3) Dequeue character
(4) Print queue
Enter choice: 2
Enter character: c
Queue contents:
Q: a b c
(0) Exit
(1) Create queue
(2) Enqueue character
(3) Dequeue character
(4) Print queue
Enter choice: 2
Enter character: 3
Queue contents:
Q: a b c 3
(0) Exit
(1) Create queue
(2) Enqueue character
(3) Dequeue character
(4) Print queue
Enter choice: 3
Queue contents:
Q: b c 3
(0) Exit
(1) Create queue
(2) Enqueue character
(3) Dequeue character
(4) Print queue
Enter choice: 3
Queue contents:
Q: c 3
(0) Exit
(1) Create queue
(2) Enqueue character
(3) Dequeue character
(4) Print queue
Enter choice: 2
Enter character: g
Queue contents:
Q: c 3 g
```

FIGURE 1. A sample session using the linked-list-backed queue driver program.

- `is_empty` *function: check whether the queue is empty.*
- `enqueue` *function: add an element to the back of queue. Your function should take constant time.*
- `dequeue` *function: remove an element from the front of the queue. Your function should take constant time.*
- `as_array` *function: fill an array with the contents of the queue.*
- `size` *function: return the size of the queue. For full credit, your function should be constant time, although a linear-time solution will receive some credit.*
- `print` *function: print all of the elements in the queue.*

The functions are specified in `hw8.h`*, and you will implement them in* `hw8.c`*. Using the function you wrote in Problem 2, add asserts to your code to check for linearity before your* `enqueue` *and* `dequeue` *functions return. The header file* `hw8.h` *has an empty definition of a* `queue` *structure; it is up to you to fill in the details. Do not modify any other part of* `hw8.h`*.*

3. Code distribution

As usual, the code distribution contains a driver program (`make driver`) and a tests program (`make tests`). The driver program is used to build a queue using your implementation. Make sure to test your `is_linear` function on lists that are linear and on lists that are not! Figure 1 shows a sample session using the driver.

4. Submission

Submit your written work as `hw8.pdf` and your programming problem as `hw8.h` and `hw8.c` to the Google Drive directory I have created for you named `comp211-f21-USERNAME/hw8/`. You should replace `USERNAME` with your Wesleyan username.

Do not forget that your written work must be submitted as a PDF! And make sure that at the top of each file you have put your name! Do not, however, change the names of the files.