Computer Engineering 12
Term Project: Loony Lists

1 Introduction

Lists are often used in two different ways: as stacks and queues, and as a replacement for arrays. The first use requires efficient operations for adding and removing items at either end of the sequence, whereas the second use requires efficient operations for indexing.

An implementation using a doubly-linked list works well for the stack and queue operations, but indexing requires linear time. An advantage of this implementation over using an array is that the list can grow or shrink as items are added and deleted.

In contrast, an implementation using an array works well for indexing, but does not necessarily work well for the stack and queue operations. However, if we treat the array as a circular queue as we learned in class, we can support the stack and queue operations in constant time. But we are still left with the problem of the array being a fixed size.

One way to overcome the problem of a fixed-size array is to increase the length of your array when it reaches capacity so it dynamically increases to accommodate the number of values. This approach requires allocating a new array, copying the contents from the old array to the new array, and then freeing the old array. All of that data movement takes time, so we do not want to do it too frequently, and it also means that when the array reaches capacity we suddenly pay a large cost. You will use this approach in the last lab project.

In this term project, we will combine a linked-list with an array in order to implement the stack and queue operations as well as indexing efficiently. The implementation is not particularly difficult and is sketched below, but will require you to think outside of class and lab about how to go about it. You have four weeks to complete this project, and will have a checkpoint after two weeks with a TA to make sure you are on the right track.

2 Interface

The interface to your abstract data type must provide the following operations:

- `LIST *createList(void)`
  return a pointer to a new list
- `void destroyList(LIST *lp);`
  deallocate memory associated with the list pointed to by *lp*
- `int numItems(LIST *lp);`
  return the number of items in the list pointed to by *lp*
- `void addFirst(LIST *lp, void *item);`
  add *item* as the first item in the list pointed to by *lp*
- `void addLast(LIST *lp, void *item);`
  add *item* as the last item in the list pointed to by *lp*
- `void *removeFirst(LIST *lp);`
  remove and return the first item in the list pointed to by *lp*; the list must not be empty
- `void *removeLast(LIST *lp);`
  remove and return the last item in the list pointed to by *lp*; the list must not be empty
- `void *getItem(LIST *lp, int index);`
  return the item at position *index* in the list pointed to by *lp*; the index must be within range

3 Implementation

3.1 Overview

A list consists of a linked list of nodes. Each node itself contains an array that will be treated as a circular queue. Recall that in a circular queue, all items are not stored in the first n slots. Rather we keep track of the first slot, f, and the items are stored starting at slot f and continue through slot $(f+n-1) \bmod m$, as shown in Figure 1.

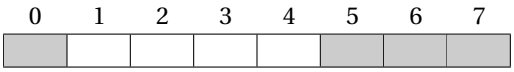

Figure 1: A circular queue of size eight containing four items starting in slot five ($f = 5, n = 4, m = 8$).

To add an item to the front of the list, we perform the following steps:

1. Go to the first node in the linked list.
2. If the array in first node is full, then allocate a new node with an empty array and make it the new first node.
3. Add the item to the front of the array in the first node.

To remove the item at the front on the list, we perform the following steps:

1. Go to the first node in the linked list.
2. If the array in the first node is empty, then deallocate the node and make the next node the new first node.
3. Remove the item from the front of the array in the first node.

Adding and removing items from the rear of the list is similar. To access the ith item in the list, we perform the following steps:

1. Start at the first node in the linked list.
2. If the node contains fewer items than the index, we move to the next node in the list until we find the correct node, adjusting our index along the way.
3. Index the array in the node we found in the previous step.

For example, if we were looking for the item at index ten and the first node contained eight values, we would be looking for the item at index two in the remainder of the list.

3.2 Open Questions

Keeping in mind that we want the operations to be as algorithmically efficient as possible, here are some key questions about your implemention:

- Should the linked list be singly-linked or doubly-linked?
- Do we need only a head pointer or both a head and a tail pointer?
- Should the arrays within the nodes all be the same size, or should nodes increase in size as we add more and more elements?
- Should we start searching for an item at a given index from the first node or the last node in the list? Does it matter?

4 Assignment

Copy the `term.tar` file from `/scratch/coen12` to get started. Call your source file `list.c`. You will present the design of your data structure to a TA by Friday, May 20th. Be prepared to show the definitions of your `list` and `node` structures. You can present your design to any TA, not just the TA assigned to your lab section. Arranging to meet during office hours or by appointment is preferred as students needing help on the lab projects will be given priority during lab time. The final implementation is due June 4th at 5:00 pm and should be submitted through the course website.

5 Grading

Your initial design check will be worth 40% of the term project grade and the final implementation will be worth the remaining 60%. Your implementation will be graded in terms of correctness, clarity of implementation, and commenting and style. Your implementation ***must*** compile and run on the workstations in the lab. The algorithmic complexity of each function ***must*** be documented.