Computer Engineering 12
Project 4: An Amazing Sort of Assignment

Due: Sunday, May 19th at 5:00 pm

1 Introduction

Unfortunately, your data structures instructor has fallen ill. Apparently, he made *himself* sick by all his bad jokes. For the remainder of the term, Professor Gosheim Loony will be your instructor. Good luck . . . you'll need it!

Professor Loony is working on an application to help a robot find its way out of a maze. He has written the maze program itself, but still needs to implement a stack. Professor Loony has already decided to implement the stack using a doubly-ended queue abstract data type, also known as a deque (pronounced *deck*). A deque allows adding items to or removing items from both the front and rear of a list of items. Professor Loony has written the interface, but did not have time to finish the implementation, so he has given you that task. (Clearly, Professor Loony is not playing with a full deque.) He has decided that you will implement the deque using a circular, doubly-linked list with a sentinel or *dummy* node. (Professor Loony assures you that his use of the term *dummy* node is in no way a reflection of his assessment of your academic ability.)

2 Interface

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

- `DEQUE *createDeque(void);`
  return a pointer to a new deque
- `void destroyDeque(DEQUE *dp);`
  deallocate memory associated with the deque pointed to by *dp*
- `int numItems(DEQUE *dp);`
  return the number of items in the deque pointed to by *dp*
- `void addFirst(DEQUE *dp, int x);`
  add *x* as the first item in the deque pointed to by *dp*
- `void addLast(DEQUE *dp, int x);`
  add *x* as the last item in the deque pointed to by *dp*
- `int removeFirst(DEQUE *dp);`
  remove and return the first item in the deque pointed to by *dp*; the deque must not be empty
- `int removeLast(DEQUE *dp);`
  remove and return the last item in the deque pointed to by *dp*; the deque must not be empty
- `int getFirst(DEQUE *dp);`
  return, but do not remove, the first item in the deque pointed to by *dp*; the deque must not be empty
- `int getLast(DEQUE *dp);`
  return, but do not remove, the last item in the deque pointed to by *dp*; the deque must not be empty

3 Implementation

As required by Professor Loony, you will use a circular, doubly-linked list with a sentinel or *dummy* node. The sentinel node is always the first node in the list, but does not itself hold data. All operations except `destroyDeque` are required to run in $O(1)$ time.

For help on the project, you seek out the assistance of the teaching assistant, Mr. Noah Tall. Noah tells you that the implementation is actually quite simple. You need to maintain a head pointer and, either explicitly or implicitly, a tail pointer. According to Noah, if you do it right, you will find that no special cases exist.

3.1 The Maze Game

Professor Loony has provided you with his maze game, which will generate and then solve a maze. To both generate and solve the maze, a stack is used to keep track of the path so that if the robot reaches a dead end, it can backtrack and explore alternative paths.

This application is actually a great example of why we would want to use a linked list instead of an array. The maximum stack size is equal to the longest path in the maze. In the worst case, the maze is simply one very long path, and the maximum stack size would equal the number of cells in the maze. Clearly, this possibility is extremely remote, and the maximum stack size in practice will be much smaller. Therefore, a linked list, in which we only allocate as much memory as we need, as opposed to an array, in which we allocate a fixed amount for the worst case, is a much better choice.

3.2 Radix Sorting

To demonstrate the versatility of a deque, Professor Loony also requires that you implement a sorting algorithm. (Yes, Professor Loony hasn't covered sorting yet, but what do you expect from him at this point!) The sorting algorithm is called *radix sort*. You are required to read in a sequence of non-negative integers from the standard input and then write them in sorted order on the standard output. Radix sorting uses queues as its main data structures, and works as follows:

1. Take the least significant digit of each number.
2. Place each number based on its least significant digit into one of a series of queues. For example, all numbers whose least significant digit is zero, will be placed in the "zero" queue.
3. Once all numbers have been placed into their queues, copy the numbers in the order they appear back into the original list.
4. Repeat the process using the next most significant digit until all digits have been processed.

You are allowed to use only a single array in your program: an array of queues that is indexed by digit. Each queue is in actuality a deque, as is the list of numbers. The number of iterations required is equal to the number of digits in the largest value:

$$\lceil \log(m+1) / \log(r) \rceil$$

where m is the maximum value and r is the radix. The larger the radix, the fewer iterations required, but then more queues are needed. For simplicity, use a radix of $r = 10$.

Again, this application is a great example of why we would use a linked list, rather than an array. The worst-case size for any one queue is the number of integers in the original list. Therefore, we would need to allocate r arrays of size n, where n is the size of the input. However, the total number of entries in all queues in any iteration will clearly be only n. Using a linked list, where we allocate space only for the entries we use, is clearly a better choice for this application.

4 Submission

Create a directory called `project4` to hold your solution. Call the source file for the deque implementation `deque.c` and your source file for the radix sort application `radix.c`. Submit a `tar` file containing the `project4` directory using the online submission system.

5 Grading

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 in your deque abstract data type *must* be documented.