Computer Engineering 12
Project 4: An Amazing Sort of Assignment

Due: Sunday, May 16th at 11:59 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.)

As if that wasn't enough, Professor Loony also wants you to implement a set using a hash table, which of course you already did. (Professor Loony seems about as mixed up as his hash values.) However, he promises that it will be easier than your previous project because you won't have to worry about probing. He wants you to implement ***hashing with chaining***, in each slot in the hash table is actually a linked list, allowing each slot to have infinite capacity and eliminating the need for probing.

2 A Generic List ADT

2.1 Interface

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

- `LIST *createList(int (*compare)());`
  return a pointer to a new list using *compare* as its comparison function, which may be `NULL`
- `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 *getFirst(LIST *lp);`
  return, but do not remove, the first item in the list pointed to by *lp*; the list must not be empty

- `void *getLast(LIST *lp);`
  return, but do not remove, the last item in the list pointed to by *lp*; the list must not be empty
- `void removeItem(LIST *lp, void *item);`
  if *item* is present in the list pointed to by *lp* then remove it; the comparison function must not be `NULL`
- `void *findItem(LIST *lp, void *item);`
  if *item* is present in the list pointed to by *lp* then return the matching item, otherwise return `NULL`; the comparison function must not be `NULL`
- `void *getItems(LIST *lp);`
  allocate and return an array of items in the list pointed to by *lp*

2.2 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 `destroyList`, `removeItem`, `findItem`, and `getItems` 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.

2.2.1 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.

2.2.2 Radix Sorting

To demonstrate the versatility of a deque, Professor Loony has also provided you with a sorting algorithm called ***radix sort***. His program reads in a sequence of non-negative integers from the standard input and then writes them in sorted order on the standard output. Radix sorting uses queues as its main data structures.

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 ten 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.

3 A Generic Set ADT

Professor Loony also thinks your hashing knowledge is somewhat limited, so he wants you to implement a hash table with ***chaining*** used to resolve collisions. Rather than each slot in the hash table holding an element, it holds a pointer to a list. The lists themselves hold the elements. For example, to add an element to our set, we simply hash the element to its appropriate list and then use our list ADT functions to check if the value is already present in the list, and if not then add the item. The list ADT does all the work for us. Our hash table is only responsible for picking the appropriate list on which to operate.

If the client specifies that at most n keys will be inserted, you should create $m = \lceil n/\alpha \rceil$ lists, where α is a constant with a value of 20. If we assume uniform hashing, then each list will contain $n/m = \alpha$ elements and the expected search time will be $(\alpha + 1)/2$.

4 Submission

Download the `project4.tar` file from the course website to get started. Call your source files `list.c` and `set.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 both of your abstract data types ***must*** be documented.