Vishal Yathish

UID: 506166275

COM SCI 32

8 December 2023

Project #5 Deque-Garage Report

**Project Implementation:**

The first class I implemented was DequeNode. All of the methods here were relatively straightforward, with only some basic getters and setters for the element of the node, *value*, the pointer to the previous node in the linked list, *previous*, and the pointer to the next node*, next*. The constructor definition in the header file set default values of *next* and *previous* as nullptr, in case the node is the first in a linked list.

The second class I implemented was Garage; I skipped Deque, since this was clearly the bulk of the project. When writing my methods for Garage, I called functions from the Deque class based on their descriptions in the Project spec and their definitions in the “Deque.h” file. Here were some of the most interesting functions from this class. The *tossIn()* function inserts an object into the garage; adds an element to the front of the mDeque linked list. If the mDeque has reached its set capacity, then it deletes the last element of the list -- the one used least -- to make space for the new object. It returns true if this deletion has to occur, and false otherwise. *use()* tries to find an object in the linked list. If it can be found, it is brought to the front of the linked list, and the function returns true. If it cannot be found, the function returns false. *find()*, as is evident, checks if object can be found in the linked list. It iterates through all indices of the list, using the get function to extract the element of each node, and checks if the element is equal to the given string parameter. If the parameter can be found in the list, returns true. If not, it returns false.

Last, I implemented Deque, which was the most tricky in my opinion. At one point, I was having so many errors with the pointers, and values being referenced that weren’t allocated in memory, that I ended up scrapping my entire implementation of the class and starting over. The first method that I found challenging was *addToFront()*, which adds an input Type value as a node to the front of the list. If the list is empty, then it simply sets the head and tail to be that node. If the list is not empty, it assigns the new front node as the previous of the old front node, and the old front to be the new node's next value. Head is reassigned to the new front. Originally, I was having issues, because I forgot to walk the head value back to the new front, and to set previous node pointers. I also intermittently forgot to have nullptr as my breaking condition in my while loops, saying that it should stop at tail, which resulted in some errors. *addToRear()* functions similarly, except adding a new node to the rear rather than the front, and walking the tail pointer. *deleteFront()* deletes the front node from the linked list. If the list is empty, then this method should do nothing. If the list has only one element, then it should render the list empty (can use the *makeEmpty()* function from before). Otherwise, it should walk the head pointer by one node, set the current head's previous to nullptr, and delete the old front node. I ended up writing many versions of this method before settling on one that I found to make the most logical sense and work the best. *deleteRear()* works very similarly. My *deleteItem()* function traverses the linked list, looking for a particular element. If it is in the list, the function deletes this node – only the first occurrence of the given element –, and returns true. If not, it returns false. When implementing this method, I made sure to take care of some special cases. If the list is empty, then this function should simply return false, since the element data cannot be found in the list. If the given element is at the front or the back of the linked list, then it can simply call the *deleteFront()* and *deleteRear()* functions defined previously. Otherwise, traverses the entire linked list to find the given element. If it can be found, it connects the two nodes on either side of the current node, then deletes the current node pointer. It is important to note that we would not be able to walk the pointer if we want to continue searching for other occurences of the element data. In that case, we would have to save the current pointer to a temporary variable, walk the current node and delete the temp. *printItems()* returns a string representation of all the objects in the linked list, using the Type->string conversion method provided to us. It iterates through all nodes in the linked list, and adds the string representation of each element to our output string, with a separator comma and space for easy reading. I formatted my output this way (instead of using arrows as typically seen with linked lists) because the conext for this assignment was that this was the list of items in a garage. I thought this format made the most sense in that circumstance.

**Project Testing:**

After ultimately re-writing my Deque class, I was able to pass all of the sample test cases that Howard provided. A common bug that came up was Thread 1: EXC\_BAD\_ACCESS (code=1, address=0x0), which occured (quite often) when there were issues with the getter functions in the DequeNode class, *getNext(), getPrevious(),* and *getElement()*. This could be because I was erroneously accessing a nullptr, my walking my pointer to the next of a tail node. It also arose as a result of not properly assigning previous pointers to my nodes; thus when it tried to access the *getPrevious()* method, it would hit the default nullptr.

Another issue was when my test cases were looking for the *front()* or *rear()* of an empty Deque, it would not properly provide default values.

Deque< int > dInt;

assert(dent.front() == 0);

assert(dent.rear() == 0);

The reason for this was that I wasn’t properly using the default no-argument constructor in these functions. With a small modification, I was able to resolve this error.

I also tested my code in g32 to check for memory leaks, and it passed all tests with no issues.