Project 1 – Design Description

Tony Shen

For this project, I utilized a doubly linked list, creating a class called node to hold the node's data, as well as pointers to both the previous and next nodes. This was defined in the header file dequedriver.h included in the submission, with the three elements all set to public. Also, this file contains the initialization of the head node, which is set to NULL to indicate an empty linked list. It is important to note that this is a global variable, which allows any function to change any properties of the head node if needed.

All function definitions are included in the header file, with the addition of the function CreateNewNode. This function allows for the easy creation of a new node, saving program runtime by avoiding repeated code in our enqueue_front and enqueue_back functions. We create the new node in dynamic memory, storing this data in the heap so that it is not cleared unless we explicitly instruct it to. Node->data is set to the value i (passed in), and the next and prev pointers are set to NULL, leaving it up to the function calling CreateNewNode to set these to whatever value is appropriate (eg. front or back).

I included a header file deque_empty.h with the class deque_empty. This is used to return a failure message when the function detects that the linked list is empty. The message returned can be accessed later in the main function through a try/catch statement blocks, used in dequeue_front, dequeue_back, front, and back.

Moving to the dequedriver.cpp, we can now go through each individual command and the associated function. Enqueue_front first creates a new node using CreateNewNode, then checks if the list is empty. If this is true, the new node will simply be the first in the linked list. Otherwise, the previous address of the head node must be changed from NULL to NewNode, and NewNode->next must be changed to head to fit this in the linked list. Head is then set to NewNode, allowing for future nodes to be inserted at the front of the queue. Enqueue_back involves a very similar implementation, with the only difference being a while loop used to traverse to the tail node (using temp, to avoid changing head), then inserting the node there. Unlike enqueue_front, head does not need to be updated here, only the pointers involved in connecting the current tail node to the new node.

The dequeue_front and dequeue_back functions also work similarly to each other. First, the exception of an empty list is handled, which is important in this case as removing from an empty list will crash the program. For dequeue_back, we then traverse to the tail node, defining the node tail to represent this. We then split into two conditions, checking if there is only one node left. In the case where there is not true, we must redefine head and either head->prev (dequeue_front) or tail->next. In the case where there is only one node left, we only need to set head to NULL, as next and prev are already handled.

The clear function utilizes two nodes current and next to clear the list. Next is used to traverse through all the non-empty nodes, and current is responsible for deleting all these nodes. First, we check if the list is empty, as we can simply output "success". We then use a while loop to traverse to the last node using current. We set next to the address pointed to by current->next, delete current, then set current to next so that we can keep progressing through the linked list. Once the while loop finishes processing, we need to take care of the head node, since this will always be left over. We make sure to set both head->next and head to NULL, then we delete head, outputting “success”.

The front and back functions also require exception handling, throwing the same error if the list is empty. Similar to the above functions, back requires us to first traverse to the tail node. We then check to see if the input parameter i is equal to the temp node that is set to either the front or back node. The strings “success” or “failure” are returned accordingly, following the project requirements.

The empty function is very simple. If head is equal to NULL, we know that the list is empty. This can be a result of either a new list or one that has had all its nodes deleted. We output “success” for an empty list, and “failure” otherwise.

The size function returns the number of nodes in the linked list. We first check if head is NULL, as we must manually return 0. If head is not equal to NULL, we simply step through the linked list and increment a counter.

The final function print first ensures that the list is not empty, as nothing should be printed if this is the case. We define two nodes: backward and forward, initializing forward to head. We then use forward to traverse through the node, outputting forward->data each time. Before we set forward to the next node, we check if we have reached the end of the list. If we have, we set backward to forward so that we have a reference for the tail node. Once we finish printing in the forward direction, we print in reverse using the backward node, setting backward to be backward->prev to get back to the head (at which backward will equal NULL).

Our main function handles input, using cin to take the first input as the command, and i as the second input, if applicable. Each command is handled with the conditional statement. Some just run the respective function (eg. enqueue_front), others must handle an exception through a try/catch block (eg. dequeue_front), and some must print the function (eg. front), as the P1 Description file requires a value be returned in these cases. If the input is not recognized, we output “ERROR”. The while loop ensures that the code stays running at all times, ready to input a command and act on it.

Testing for this program was done initially through console input. I either inputted commands line by line and observed results and errors, or by copy-pasting in a large block of commands (my own test cases as well as the ones provided), relying on the while loop to process each command sequentially. I then performed the same tests on the Ubuntu server, creating text files and using them as inputs to run the program. These text files included the same input files provided as well as my own test cases. Specifically, I tested the error class (empty list), as well as the case where there was only one element.