Details on Design Decisions:

- **Constructor:** My node class's constructor takes in a key and value which are both strings and initialises those member variables. It also sets the next and prev pointers to nullptr to avoid pointing to random data. My doubly linked list constructor takes no parameters and assigns the head and tail pointers to nullptr for similar reasons to the node class. My deque constructor calls its parent's (doubly linked list) constructor and also sets the current size to 0. It also takes in a max size and initialises that member variable. The current size represents how many nodes are in the deque at any given time which is important to know when I have exceeded the max size and need to pop.
- **Destructor:** The destructor for the node class sets the next and prev variables to nullptr to avoid dangling pointers. The destructor for the doubly linked list class clears the linked list by repeatedly calling pop_back until the list is empty. The pop_back/push_back function deallocates memory for each Node (which is dynamically allocated on push_front/push_back) and reassigns the head and tail pointers. As a result, by repeatedly calling the pop_back function until the list is empty I know I am deallocating all the memory used. I also know that head and tail are set to nullptr so I don't need to worry about dangling pointers. My deque class inherits the destructor from the linkedlist class and doesn't need its own.
- **Constant Functions:** For the deque class I made print, size and is_empty constant functions. For the doubly linked list class I made front, back, search and is_empty constant. For the node class I made get_next, get_prev, get_key and get_value constant. This is because all of these functions should not change any of my member variables.
- **No Constant parameters or operator overloading used:** All of my function parameters are either strings, pointers or integers and so they are being passed by value meaning it wouldn't make much sense to make them constant. I also had no need to use operator overloading in this project.

Test Cases

Strategy:

A) Brainstorm all possible edge cases that may cause bugs or errors in my program
B) Handwrite solutions to all my possible edge cases and write what my expected output should be
C) Verify through my program that I get the same results as expected through the print function and debug/fix if there are any errors
D) Verify through valgrind that I have no memory leaks

Sample Test Cases:

1. Creating a deque class with max size 3 - Print success
2. Calling pop_back when the deque is empty - Print failure
3. Calling pop_front when the deque is empty - Print failure
4. Calling push_front when the deque is not at max size - Print success and the deque should now hold that URL at the front