5. Calling push_back when the deque is not at max size - Print success and the deque should now hold that URL at the back
6. Calling push_front when the deque is at max size - Print success and delete the URL that was originally at the end of the deque
7. Calling pop_back when the deque is non empty - Print success and the last URL should be removed
8. Calling pop_front when the deque is non empty - Print success and the first URL should be removed
9. Calling clear on an empty deque - Print success and deque should stay empty
10. Calling clear on non empty deque- Print success and deque should now be empty

Performance Considerations

- **Find**: In my implementation find has an asymptotic tight bound of theta(n) where n is the number of elements in the deque. My find function does a linear search through all the nodes starting from the head. It sets a temp node to the head and iterates while that temp node is not null or when the temp nodes key is equal to the key we are looking for. Each iteration it sets the temp node to the next node. After the loop finishes the temp node is returned. In the worst case the key does not exist in the deque and the while loop does n iterations. Checking the while loop condition is constant time and returning the temp node is also constant time so overall the tight bound running time of find is theta(n). This all also means that the upper bound is O(n) since the tight bound is also an upper bound.
- **Clear:** In my implementation clear has an asymptotic tight bound of theta(n) where n is the number of elements in the deque. Clear continuously calls the pop_back function while the deque is not empty. Each time pop_back is called the size is reduced by 1 so the number of times clear calls pop_back is n times. The pop_back function is done in constant time since it only consists of constant time operations. As a result the overall tight bound running time of clear is theta(n). This also means that the upper bound is O(n) since the tight bound also represents an upper bound.
- **Print:** In my implementation print has an asymptotic tight bound of theta(n) where n is the number of elements in the deque. My print function sets a temp node to the end of the list and traverses backwards using a while loop until it reaches nullptr (the prev node of the head). As a result the number of iterations the while loop does is n. Each iteration prints the current key and value of the temp node which takes constant time. Therefore the overall running time of print is theta(n). This also means the upper bound is O(n) since the tight bound also represents an upper bound.
- **Others:** All of my other functions only use constant time operations and therefore run in O(1). This is done by manipulating pointers to prevent needing to traverse through the entire linked list a lot.