There seems to be much difference in the implementation of the designs of our Linked_List, when compared to the positional model. For example, the idea of having a positions class seemed rather odd to me, but this also helps in terms of efficiency, since all the methods of the positional list ADT run in worst case O(1) time. This means that the positional list ADT is much faster than the Linked_List and that's why it may make sense to use the positional ADT in cases where we would first think to use a Linked_List. .

What personally struck me as odd was that that the positional list would be able to insert elements at O(1) time, since the Linked_List implementation did so at O(n) time. However, this makes sense since accessing elements runs at O(1) time in the positional ADT list, which means that all that needs to further be done is inserting the element, which can easily be done at O(1) time when given the node before or after.

However, there seems to also be much similarity between the positional model and our Linked_List. The positional model in the textbook is actually based around the doubly linked list, which is why it makes sense that the positional list is able to do operations such as L.first(), L.last(), L.add_first(), L.add_last() at O(1). This also helped myself visualize how some of the accessors and mutators worked, although how the position class works still seemed to frustrate me a bit.

Altogether, the positional list ADT seems to have much application in the real world. For example, any time where we would need to create more efficient programs, it seems that using the positional list would be wiser. However for more simple applications, the Linked_List seems easier to use because it doesn't rely on the abstract concept of a positions class.