# Writeup for Project 2

## Part 1: Explanation of Algorithm

List is a commonly used data structure. However, it has many drawbacks. For example, all the elements inside a list must be stored physically next to each other in the memory of the computer. This requirement largely affects the performance of utilizing the list structure. For example, appending an element to the list may change the physical location of the whole list in the memory thus resulting in insufficiency. This Project Constructs a new type of data structure called linked list to improve the performance of the normal list structure.

Elements in the linked list structure are called the node objects. Each node object has three attributes: next, previous and value. The next attribute points to the next element in the linked list and the previous attribute points to the previous object in the linked list. For example, element B has next attribute C and previous attribute A. B._next is C and B._previous is A. These two attributes connect all the nodes together to form the linked list structure thus are of great significance. The value attribute stores the value stored in the node object. When a node object is created, it would initially have next and previous attribute point to none. And these attributes will be finalized by the methods operating on the linked list. Note that the node class is a private class in the linked list so that users cannot access any single node object.

When being created, each linked list object would have 3 attributes: the header node, the trailer node and size. The header node marks the start of the linked list and the trailer node marks the end of the linked list. Both nodes would have value None and would only act as sentinel nodes that do not contain any information. Size records the number of nodes in the linked list object except the header and trailer nodes. When a linked list object is initially created, the header node would have its next attribute point to the trailer node and the trailer node would have its previous attribute pointing to the header node. In other words, header._next is trailer and trailer._previous is header. The size of the linked list would be 0. Note that all the attribute in the linked list class is private, meaning that users cannot access the attributes inside the linked list class. The only way to interact with a linked list is through its methods.

A linked list class has 9 methods in total. The following paragraph would describe the algorithms behind each method and analyze the overall performance of each method.

The __len__( ) method is a simple method that returns the size of the linked list to the user. This method has constant time performance as size is stored as an attribute in the linked list.

The append_element( ) method appends an element to the tail position of the linked list which is the position previous to the trailer node. This is the only method to append an element to the linked list. When this method is called. Create a new node object with the value given by the user. First, make the next attribute of the new node point to the trailer node. Then change the next attribute of the node previous to the trailer node which is trailer._previous from the trailer node to the new node. Also, link the previous attribute of the new node to the node previous to the trailer node which is trailer._previous. Finally, point the previous attribute of the trailer node to the new node. Add 1 to the size of the linked list. Now the new node is connected between the trailer node and the node originally previous to the trailer node. This means that the new element is successfully appended to the structure. This method has a constant time performance O(1). Each time an element is appended, we only access the trailer node and the last node in the original structure. As the trailer node is stored as an attribute and the last node in the original structure can be accessed through trailor._previous, the performance of the method is constant time.

The insert_element_at( ) function inserts an element of a specific value to the desired index provided by the user. If the index provided by the user is smaller than 0 or larger or equal to the size of the linked list, the method would terminate and raise an IndexError. If a valid index n is given, first create a new node object with the value provided by the user. Then evaluate whether this index is in the first half of the linked list or the second half of the linked list. If the index is in the first half, iterate through n elements from the header node to locate the node at the position previous to the desired index. Call that node current. Make the next attribute of the new node new_node._next point to the next node of current -- current._next. Make the previous attribute of the next node of current -- current._next point to the new node. Then change the next attribute of the current node to the new node. Finally, make the previous attribute of the new node point to the current node. Now the new node becomes the next element of the current node and the index of the new node is n. If the index n is in the second half, the algorithm is similar, iterate through (size-n) element to locate the node at the location after the desired index n. Call that node current. Link the new node between the current node and the previous node current._previous. Now the new node is linked in the structure and the index of the new node is n. This method would have linear time performance O(n). To insert a node at an arbitrary index, the method would have to walk through at most half of the elements in the list. For example, if we try to insert an element 7 to index 3 of the list [ 1, 2, 3, 4, 5, 6 ]. The method would first go

through the second half of the linked list, locating element 4 in the list. Then link 7 between element 3 and element 4. In the case where the linked list is a singly linked stricture where a node only has the next attribute but not the previous attribute. To insert an element at an arbitrary index may need to iterate through every element in the list. In this doubly linked list structure, the previous attribute enables us to locate an element using the previous attribute from the trailer node, resulting in double efficiency comparing to a single linked list.

The remove_element_at( ) function remove the element at a given index. In order to remove an element, that element should be unlinked from the linked list. If the index is larger or equal to the size of the linked list or smaller than 0, the method would terminate and raise an index error. If a valid index is given, first evaluate whether the index n is in the first or the second half of the linked list. If it is in the first half, iterate through n elements from the header node to locate the node at index n-1. Call that element current. Make the previous attribute of the second element next to current -- current._next._next._previous point to current and make the next attribute of current point to the second element next to current. So, the element at index n next to current is unlinked from the list. If the index is in the second half of the linked list, iterate through (size – n – 1) element to locate the element at index n+1. Call that element current. Make the next attribute of the second element previous to current -- current._previous._previous._next point to current and make the previous attribute of current point to the second element previous to current. Now the element at index n previous to current is unlinked from the list. This method has a linear time performance as it has to walk through at most half of the elements in the worst case where the user wants to remove an element in the middle of the linked list. For example, the list is [ 1, 2, 3, 4, 5 ] and the user wants to remove the element at index 2. The method would first locate element 4 and link 2 to 4. So, 3 is unlinked from the list. In this case, the method goes through almost half of the list. Comparing to a singly linked list where each node only has the next attribute, this method is more efficient. The remove_element_at function for a single linked list would have to iterate through almost all the elements in a list if the user wants to remove an element at the tail position. However, in this doubly linked list structure, we can locate an element from the trailer node or the header node using the previous or the next attribute, achieving double efficiency comparing to a single-linked list structure.

The get_element_at( ) method returns the value stored in a node at a given index to the user. If the index is larger than the size of the linked list or is smaller than 0, raise an index error. If a valid index is given, first evaluate whether the index n is in the first or the second half of the linked list. If the index is in the first half, walkthrough n+1 element from the header node to locate the element at the desired index, return the value stored

in that element to the user. If the index is in the second half of the linked list, walkthrough (size-n) elements from the trailer node to locate the element at the desired index. Return the value stored in that element. This method has a linear time performance. To get the value at an arbitrary index, the method would walk through at most half of the elements in the worst case where the user wants the element in the middle of the list. Comparing to a single linked list, this structure also has higher efficiency. In a single linked list, the get_element_at method would have to walk through all the elements in a list if the user wants to get the last element in the list. In this double linked list structure, we are able to locate an element form both the header and the trailer node using the next and previous attribute. This limits the number of elements the get_element_at method need to walk through to half the size of the linked list. Although the overall performance is still linear time, it is more efficient than a single linked list.

The rotate_left( ) function rotates the list left so that each node moves one position earlier than it was and the node originally at the head position move to the tail position. If the size of the list is 0 or 1, the function would do nothing and return. The list remains unchanged. If the size of the list is greater than 1, record the first node at the head position as current. Make the next attribute of the header node header._next point to the second node in the list. And make the previous attribute of the second node point to the header node. Now the header node is linked to the second node in the original list and current is unlinked. The next step is to put current to the last position in the list. Make the next attribute of current current._next point to the trailer node. Make the next attribute of the last node in the original list trailer._previous._next point to current. Make the previous attribute of current point to the last node in the original list trailer._previous. Finally, make the previous attribute of trailer trailer._previous point to current. Now current is linked between the trailer node and the original last node in the list. So current is at the tail position and every other node moves one position to the header node. This method has a constant time performance. Each time the method is called, we only have to locate header node, trailer node and the nodes at index 0, 1 and tail position. As this is a doubly-linked list, we can use the next attribute to find the nodes at index 0,1 from the header node and locate the node at the tail position using trailer._previous. Any other elements in the list would not be accessed throughout the whole process. A single linked list, on the other hand, would have a linear time performance as to access the last node, the method has to go through every element in the linked list. This doubly linked list largely improves the overall performance, making the rotate_left method much more efficient.

The __str__ method( ) returns the value stored in the linked list as a string. When the method is called, mark current the element next to the header node. Define string variable as the single bracket “ [ “. Set variable number_of_element to 1. While current

is not the trailer node, append a blank space and the value stored in current as a string to the string variable. Add 1 to variable number_of_element. If number_of_element is greater than 1 meaning there is more than 1 element in the list, also append a " , " to the string variable to separate the values. Use current = current._next to iterate through the next element. As current is initialized to be the node next to the header node, and the string would append only when current is not the trailer node, the value of the header and trailer node would never be appended to the string variable. So only the useful information stored in the linked list would be returned as the string variable. This method has a linear time performance as the method goes through every element in the list and record its value. This performance is the same for a singly linked list structure.

The __iter__ method( ) helps to initialize the iteration of the linked list structure by setting the variable iterator to be header._next. This is an initializer that initializes a variable thus has a linear time performance.

The __next__ function( ) helps the build-in structure in python to iterate through every value in a linked list. The iteration would stop if it iterates through every element in the linked list so if the variable iterator is the trailer node, call StopIteration. When python iterates through every element in a linked list, it will first call the __iter__ method and then call the __next__ function until a StopIteration is called. In the __next__ method, mark the value stored in the variable iterator as to_return. Move the iterator function to the next element in the list by iterator = iterator._next. Return to_return. The __next__ function itself has a constant time performance as each time It is called it moves to the next element in the list and returns a value. When python iterates through every element in a list. It would be a constant time performance overall as it would call the __iter__ function n times to iterate through every element of a list of size n.

Part 2: Analysis of Testing

In the testing of the program, each method would be tested with its expected error when the size of the linked list is 0, 1, and 3. These tests reflect the reliability of the program in various situations.

The basic method of linked list is append_element, __str__, and __len__ as the testing of other methods would rely on those methods. These methods are tested first.

Create a linked list object a. Test the __len__and __str__ method by printing the length and the string representation of a. As a is empty, it should have length 0 and the __str__ function should return "[ ]". Test the get_element_at function by trying to get an

element at an invalid index. This should raise an exception of IndexError. Results are as expected, meaning these 3 methods are functional when the size of the list is 0. Try to append an element when the size of the list is 0. Append number 2 to the linked list, print “Element appended!” if the function was successfully called, print an error message if there exist AttributeError inside the append method. “Element appended” is printed, meaning there is no Attribute Error in the function. Now print the string representation of the list and the length of the list. The string is now [ 2 ] and the length is 1, showing that the __str__ and the __len__ operates correctly when the length of the list is 1. And it also shows that the append_element method successfully appends 2 to the linked list and increases the size of the list by 1. Now test the get_element_at method with a valid index by printing the element at index 0. 2 is printed. Also, test the function with an invalid index by trying to print an element at index 3. The testing raises an index error and prints an error message, showing that the get_element_at function is reliable when the length is 1. Similarly, test the append function when size is 1, append 3 to the list and then append 4 to the list, print an error message if there is an AttributeError in the function. “Element appended!” is printed, meaning there is no AttributeError in the function. Test the __str__ and __len__ function by printing the length and the string representation of the list. Length is 3 and the string is now [ 2, 3, 4 ]. This result shows that the __str__ , __len__ and append_element function works in all situations and elements are successfully appended. Now test the get_element_at function. As this function uses different approaches to get an element from the first half and the second half of the linked list, two indexes should be tested. Test the method for getting an element in the second half of the list by printing the last element in the list, raise an IndexError if the index is out of range. 4 is printed and no IndexError is raised. Test the method for getting an element in the first half of the linked list by printing the first element in the list. 2 is printed. Finally, print element at index 4 which is not a valid index. An IndexError Exception is raised, and an error message is printed. This result shows that the get_element_at method is reliable under all circumstances.

After these four methods are tested, we can test more methods base on these results. First test the remove_element_at function. This function is index-based and includes code like current._next._next that may raise an AttributeError. So in each case there are two exceptions. The first case is when the size of the list is 3. Try to remove an element at an invalid index, raise an exception if there is an IndexError or AttributeError, print the list after the method is called. An error message is printed, and the list remains unchanged. Call the remove_element_at function with index 3 and 0 so that index in the first and second half of the linked list is included. Print the string representation and length of the list. The list is now [ 3 ] and the size is 1. Try to remove at an invalid index again, an error message is printed and the list is unchanged. Now remove element at index 0, the new list is [ ] and the size is 0. Try to remove again, an Index Error

message is printed. These tests show that the remove_element_at methods function well for all lengths and all indexes.

Similarly, test the insert_element_at function. First insert when the size is 0. All indexes should be invalid. An IndexError exception is raised. The list is unchanged. Now append an element 2 so that the size is 1. Insert element 3 at index 2 which is an invalid index, an IndexError exception is raised. Insert element 3 at index 0 which is a valid index, the insertion is successful and the list is now [ 3, 2 ] and the length is 2. Append an element 4 so the size of the list is 3. The list is now [ 3, 2, 4 ]. Try to insert element 3 at index 4 which is invalid, an error is raised, and the list is unchanged. Now test inserting in the second half of the list by inserting 7 to index 2. The list is now [ 3, 2, 7, 4 ] and the size is 4. Try to insert element 9 to index 0 which is the first half of the linked list. The list is now [ 9, 3, 2, 7, 4 ] and the length is 5. These tests show that the insert_element_at method works for all indexes and inserting in both the first half and the second half of the linked list is functional.

Now test the iterator. The iterator would iterate through every element in the linked list. The print method would print the value. For the list of length 5, all the elements are printed in order: 9, 3, 2, 7, 4. Now use the remove_element_at method to change the length of the list to 1 by keep removing the last element in the list. The list is now [ 9 ]. Test the iterator again. The only element 9 is printed. Now remove again so that the size of the list is 0. Test the iterator. Nothing is printed. These results show the iterator works for lists of all lengths and gets the value correctly.

Finally, test the rotate_left function. When the size of the list is 0, call the rotate_left function, raise an exception if there exist AttributeError in the function. Print the list after the method is called. [ ] is printed and no error was raised. Append element 2 to the list so that the length is 1. Call the function again. The list is [ 2 ] and the length of the list is unchanged. Append elements 3 and 4 to the list so that the length is 3. Call the method again. The list is now [ 3, 4, 2 ] which is correctly rotated. These results show that the rotate_left method successfully rotates the list for any size and does not change the length of the list.

Part 3: Analysis of Josephus

To solve the Josephus problem, define the Josephus( ) function. The Josephus( ) function takes in a linked list as a parameter. It rotates the list left and delete the first element in the list until the size of the list is 1. This method is equivalent to make the linked list into a circle. Mark a location in the circle. Rotate two positions each time and

terminate the element on the marked location until one is left. The function of Josephus has a linear time performance. The while loop keeps operating until there is only 1 element in the list. As each time 1 element is removed from the list. The while loop would run n-1 times for a list of length n. Inside the while loop, two methods are called, the rotate_left( ) method and the remove_element_at(0) method. As analyzed above, the rotate_left( ) function has a constant time performance. Although the remove_element_at( ) function has a linear time performance for an arbitrary index, the performance of remove_element_at(0) has a constant time performance as it only has to link the header node to the second node in the list. So, the overall performance of the Josephus function is linear time performance as it deletes every element except the last survivor. Also note that in the main function, a for loop is used to append every integer from 1 to n to the list. This requires a linear time performance. So the main program overall is a linear time performance which consists of a for loop and the Josephus function.

Part 4: Comparison to the textbook

The example in the textbook focuses on a positional list which records the relative position of each node object. Although this implementation has the benefit of more efficient interaction with the list, it also has some drawbacks.

First, compare the method for our doubly linked list and the positional linked list.

The append_element and len_method have similar performance in both structure as both structures can easily access the size of the linked list and last element in the list. The append_element method in the textbook can be defined as insert_element_between the trailer node and its predecessor. Thus the performances of both structures are O(1).

The insert_element_at, remove_element_at and get_element_at methods are different in these two structures. As discussed in the performance analysis, the performance for our double linked list is O(n) as the method would have to iterate through at most half of the elements in a linked list in the worst situation. However, in the linked list in the textbook, all three methods have a constant time O(1) performance as each node is stored so that user has direct access to each node in the list. The insert_between method in the textbook takes in the predecessor and the successor as input for the method. So the new node is directly linked between these two node objects. The delete method also directly link the predecessor to the successor so that the performance is also constant time. Note that the insert_element_between function in the textbook takes in the predecessor and successor as input. So if the user does not know exactly the

successor and predecessor node object, he or she would also have to go through each element to find the exact position which would also result in linear time performance.

The rotate_left function would be similar in both structures. In our linked list structure, it has a constant time performance. In the positional list structure, it can also record the first node, unlink it from the list and append it to the tail position of the list. This would also be a constant-time performance. So the performances of these two methods are similar.

The __str__ function would also be similar for both structures. The positional linked list structure would also have to iterate through each element and append the string representation of the elements one by one. This would also result in linear time O(n) which is the same as our linked list structure.

The __next__ and __iter__ functions are also similar in both cases, __next__ acts as an initializer for the __iter__ function and the __iter__ function goes to the next element in the list and return the value.

Overall, the textbook has potentially more efficient methods. But to achieve its full potential, the user has to record each node objected created which also complexify the implementation of the structure. If the user does not store the node object created, he or she also has to go through elements in the list to find a specific location. The linked list method we use, on the other hand, is more user-friendly as it is index-based, although the performance of list interaction may have linear time performance.

To put all into a nutshell, if the data stored is of large size but not much change would be made except at both sides of the list, our list implementation would provide a better user experience. If the data stored is going to be manipulated constantly, the implementation of the textbook provides a better option.