Project 2 Write Up
CSCI 241
Team BadLuckBrian - Catherine Romadka-Fahl, Yunwoo Noh

Challenges

We encountered multiple problems throughout this project; the major problems were understanding how to access the attributes of the poly val class, writing correct loops for the polynomial addition, calculating the size of the linked list correctly, and printing the list. When we were trying to implement the polynomial addition we continuously received a none type error when we tried to access the attributes of a polyval class object. We tried to store all attributes in variables to clarify and simplify our if statements; however, we continued to receive the same error and eventually, we found our length method was wrong. We noticed that we had unintentionally doubled the size attribute whenever we called length, instead of merely returning the size that other methods had manipulated.

After we resolved the size issue, the none type error disappeared, but p3 was still incorrect. We accidentally wrote the code to compare every element even if it had already been appended to p3, and so we had multiple duplicate terms. After we changed the inner loop from a for loop to a while loop and made sure that we increment i and j, p3 was correct. The other main problem we experienced was printing the list. We specifically had problems when we used the join method, as the elements in list of the node values were of type integer, not string. This created an error when trying to use the join method that was resolved by using the str() method when adding node values to the list.

Justification for Test Cases

1. append correctly to empty list: we wanted to make sure that elements get appended correctly between the sentinel nodes when the list is empty

2. append correctly to non-empty list and change size: make sure the new elements get added correctly after the existing elements. We also wanted to check that it increases the size correctly.

3. insert correctly to non-empty list and change size: insert an element correctly at given index and increase the size correctly

4. insert at invalid index: when an element is attempted to be inserted in invalid index, we wanted to make sure that it returns nothing.

5. insert at out of range index: when an element is attempted to be inserted in invalid index, we wanted to make sure that it returns nothing. This time we tested index out of range.

6. remove an element correctly: we wanted to make sure that it removes an element at given index correctly and changes the size.

7. remove element at invalid index: when an element is attempted to be removed in invalid index, we wanted to make sure that it does not change the list

8. remove at out of range index: when an element is attempted to be removed in invalid index, we wanted to make sure that it does not change the list. This time we tested index out of range

9. get element correctly: we wanted to make sure that it correctly returns the element at given index

10. get element out of range: we wanted to make sure that when it is attempted to get element out of range, it returns none.

11. print list correctly: we wanted to make sure that it prints the list with the correct format

12. print lists with variety of lengths: we wanted to make sure that it prints the list with the correct format regardless of the number of elements.

13. size of empty list: we wanted to make sure that it returns 0 and does not include the sentinel nodes in the size

The Importance of Node Reference Order

The order in which we changed node references is important because changing the wrong references first would result in lost data. For example, if we added a new node between node a and node b and then changed the next reference of node a to the new node, we would lose node b because nothing would be referring to it. Therefore, Python would eventually collect it as garbage. If we changed node b's previous reference to the new node, we would not lose node a and any previous nodes in the list, but we would have to traverse the list starting at the tail in order to navigate to node a. In order to avoid these problems, we assigned the new node's previous reference to node a and its next reference to node b before changing node a's or node b's references. Changing the references in this order ensures that both the part of the list before and after new node always have something referencing them.