**Linked List Implementation**

For my doubly linked list implementation I used a head pointer and a tail pointer. I didn’t use any dummy nodes and instead I just accounted for edge cases when inserting and erasing the first item in the array. Each link in the list contained a m\_next pointer and a m\_prev pointer. The tail pointer had a nullptr for m\_next and the head pointer had a nullptr for m\_prev.

**Pseudocode**

Copy constructed Set Declaration

If the size is 0

Set head and tail to nullptrs

return

Create new node Head

Initialize it with nullptrs and insert its value

While current item not pointing to nullptr

Move current item pointer to next item

Create new node

Assign prev pointer of new node to prev item

Assign next pointer of new node nullptr

Insert value of new node with current items value

Assign previous items next pointer to the new item

Previous item now has address of new item

Tail is equal to address of last item inserted

Assignment Operator Declaration

If the address passed in is not equal to this set

Create copy constructor initialized with address passed in

Swap temp Set with the values of this Set

Return \*this

Destructor Declaration

If the size is 0

Delete head and tail pointers

Return

While current item is not a nullptr

Assign address of next item to temp variable

Delete current item

Current item is assigned the address of the next item saved in the temp variable

Insert declaration

If the value is contained within the list

Return false

If the size is 0

Create new node

Assign head and tail pointers to new node

Make pointers within node nullptrs

Insert value in to the node

Increment size

Return true

Traverse through string and break where currentitem is alphabetically greater than value passed in

If the previous pointer within the node is a nullptr

Create new node

Insert value

Make prev pointer of new item a nullptr

Make next ptr of new item the current item

Set the head to the new item

Increment size

Return true

Create new node and insert its value

Assign appropriate next and prev pointers to node

Assign appropriate next and prev pointers to the previous item and the next item

If the next pointer is a null ptr

Assign it to the tail pointer

Increment size

Return true

Erase Declaration

If the list doesn’t contain the word

Return false

If the size is 1

Delete the item

Set head and tail pointers to nullptr

reduce size

Return true

Traverse to value which contains to target value

If its the last item

Delete the node

Set the previous node’s next pointer to null

Set previous node to tail

Reduce size

Return true

If its the first item

Delete the current Item

Set next Node’s previous pointer to null

Set the Next node to the head pointer

Reduce Size

Return true

Previous item’s next pointer to the next item

Next Item’s previous pointer to the previous item

Delete the current item

Reduce size

Return true

Swap Declaration

Swap Sizes

Swap head pointers

Swap Tail pointers

Unite declaration

Declare new temp set

Loop through values of the first set

Insert all values in to the temp set

Loop through values of the second set

Insert all values that are not duplicates in to the temp set

Swap temp set with the result set

butNot declaration

Declare new temp Set

Loop through Values of the first set

if the second set doesn’t contains the same value

Insert the value in to the temp set

Swap the temp set with the result set

**Test Cases**

The tests were performed o strings

Set X;

assert(X.size() == 0); //check that size 0

assert(X.empty()); //check that it returns as empty;

assert(!X.erase("Laser")); //nothing to erase

assert(X.insert("ades")); //inserts in to a empty Set

assert(!X.insert("ades")); //doesn't insert duplicates

assert(X.insert("spade")); //inserts without issue alphabetically

assert(X.insert("zade")); //inserts without issue alphabetically

assert(X.insert("jade")); //inserts without issue alphabetically

assert(X.size() == 4); //check that size is 4

X.dump(); //print out list forward/backwards to ensure pointers intact after insert

assert(!X.empty()); //check that the list is not empty

assert(X.contains("ades") && X.contains("spade")); //check that contains is working

assert(X.erase("jade")); //check that erase works with pointers intact

X.dump(); //print out list forward/backwards to ensure pointers intact after erase

assert(X.erase("spade")); //check that erase works with pointers intact

assert(!X.erase("spade")); //erase doesn't work after item been removed

assert(X.erase("zade")); //check that erase works with pointers intact

assert(X.erase("ades")); //check that erase works with pointers intact

assert(X.size() == 0); //check that size 0

assert(X.empty()); //check that it returns as empty;

ItemType Test = "Test";

Set Y;

assert(Y.insert("Avi")); //inserts without issue alphabetically

assert(Y.insert("Larsen")); //inserts without issue alphabetically

assert(Y.insert("Joey")); //inserts without issue alphabetically

assert(Y.insert("Rohit")); //inserts without issue alphabetically

assert(Y.get(2, Test)); //get works when indexing within linked list

assert(Test == "Larsen"); //Saves correct value in to value passed in

assert(!Y.get(4, Test)); //doesn't work when indexing out of list

assert(Test == "Larsen"); //value passed in remains unchanged

Set Z;

assert(Z.insert("Brady")); //inserts without issue alphabetically

assert(Z.insert("Rohit")); //inserts without issue alphabetically

assert(Z.insert("Arian")); //inserts without issue alphabetically

assert(Z.insert("Jon")); //inserts without issue alphabetically

assert(Z.get(2, Test)); //get works when indexing within linked list

assert(Test == "Jon"); //Saves correct value in to value passed in

assert(!Z.get(100, Test)); //doesn't work when indexing out of list

assert(Test == "Jon"); //value passed in remains unchanged

Y.swap(Z); //swapping works and values switch including size

assert(Y.contains("Arian") && Y.contains("Brady") && Y.size() == 4);

assert(Z.contains("Avi") && Z.contains("Joey") && Z.size() == 4);

Set XX;

Set YY;

unite(Y, Z, XX); //unite works

Set ZZ(XX); //can initialize a set w a copy constructor

ZZ.dump(); // ensure pointers intact after unite

butNot(XX, Y, YY); //save an identical list to Z in YY

YY.dump(); //ensure pointers are intact

ZZ = YY; //check assignment operator

ZZ.dump(); //contains corrrect values (YY)

butNot(XX, Y, XX); //works while refering to the same set

XX.dump(); //check contains values in XX and not in Y

unite(XX, Y, XX); //works in same set

XX.dump(); //prints out the Z and Y together

butNot(XX, XX, XX); //emptys out the Set

assert(XX.empty()); //check that set is now empty

unite(ZZ, ZZ, ZZ); //checks that string remains the same

ZZ.dump(); //check that ZZ remains the same