The doubly linked list is implemented as two pointers to two nodes, one for the head of the list and one for the tail of the list. The list is not circular and does not have a dummy node. Each node contains a pointer to a previous and a next node, and a struct containing the ItemType and its count. The nodes are not in any particular order.

Psuedocode for Multiset::insert

*search through list for node whose data’s ItemType matches the the given value:*

*if found, increment the node’s data’s count by one*

*otherwise, insert new node containing the given value into the linked list*

Psuedocode for Multiset::erase:

*for each node in the list:*

*if data’s ItemType matches the given value:*

*if count is greater than one:*

*decrement count by one*

*otherwise:*

*remove node from linked list*

*decrement set’s size by one*

*if there is no such node whose data’s ItemType matches the given value:*

*nothing to be done*

Psuedocode for combine:

*if ms1 and result are the same object:*

*add the contents of ms2 into result*

*if ms2 and result are the same object:*

*add the contents of ms1 into result*

*otherwise:*

*erase the contents of result*

*add the contents of ms1 into result*

*add the contents of ms2 into result*

Psuedocode for subtract:

*if ms2 and result are the same object:*

*make a temporary copy of ms2*

*set result equal to ms1*

*for each item in the temporary copy:*

*erase it from result as many times as it occurs in the temporary copy*

*otherwise:*

*set result equal to ms1*

*for each item in ms2:*

*erase it from result as many times as it occurs in ms2*

Tests performed on a multiset of strings

auto equals = [](Multiset a, Multiset b) //Tests if two multisets have the same order and size

{

if (a.size() != b.size())

return false;

if (a.uniqueSize() != b.uniqueSize())

return false;

for (int i = 0; i < a.uniqueSize(); i++)

{

ItemType item1, item2;

int count1 = a.get(i, item1);

int count2 = b.get(i, item2);

if (count1 != count2)

return false;

else if (item1 != item2)

return false;

}

return true;

};

//Empty constructor

Multiset test1;

//Check for insert

assert(test1.insert("a"));

assert(test1.insert("a"));

assert(test1.insert("b"));

assert(test1.insert("c"));

//Check for insert and size working correctly

assert(test1.size() == 4);

//Check for uniqueSize

assert(test1.uniqueSize() == 3);

//Check for copy constructor

Multiset test2(test1);

assert(equals(test1, test2));

Multiset test3;

//Check for assignment operator

{

Multiset temp(test2);

test3 = temp;

}

assert(equals(test1, test2) && equals(test2, test3));

//If assignment opreator and copy constructors don't function properly, these next two should fail because of dangling pointer dereferences or something else

assert(test3.insert(""));

assert(test3.contains(""));

//Make sure erase functions work properly

assert(test3.eraseAll("a") == 2);

assert(test3.erase("b") == 1);

assert(test3.eraseAll("c") == 1);

assert(test3.erase("d") == 0);

assert(test3.erase("") == 1);

assert(test3.empty());

//Any nullptr dereferences should be checked here, as the set is cleared and then inserted into

assert(test3.insert("e"));

assert(test3.size() == 1);

assert(test3.uniqueSize() == 1);

Multiset test\_result;

//Like addition: x + x = y

// y - x = x

//Additionally, verifying that nonempty multisets for the result parameter are properly handled

combine(test1, test1, test\_result);

subtract(test\_result, test1, test\_result);

assert(equals(test1, test\_result));

test\_result = test3;

//Aliasing checks

combine(test1, test3, test3);

subtract(test3, test1, test1);

assert(equals(test1, test\_result));