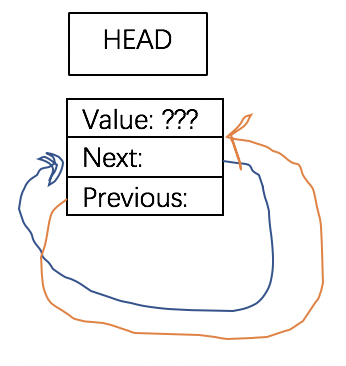
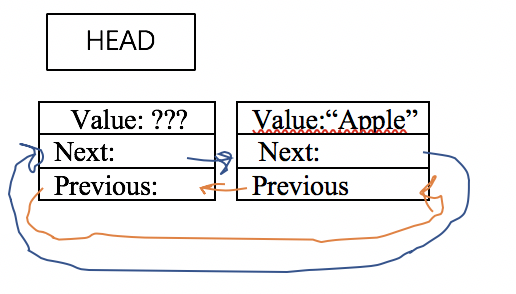
My list is a doubly linked list made up of nodes. Each node has three features: a pointer to the previous node, a pointer to the next node, and a value of type ItemType (which can be specified depending on the tester/user). The list is circular meaning that the head will point to the tail and the tail will point to the head. The list does have a dummy node at the head. The dummy node does not hold any value. The nodes in my list are inserted in alphabetical order. This makes it so that a lot of functions, like get(int pos, ItemType& value), are easier to implement. If the nodes are in alphabetical order, when I call get, I just have to iterate through the list and stop at position pos to get the correct value instead of searching otherwise.

Here’s a diagram of my doubly linked list:

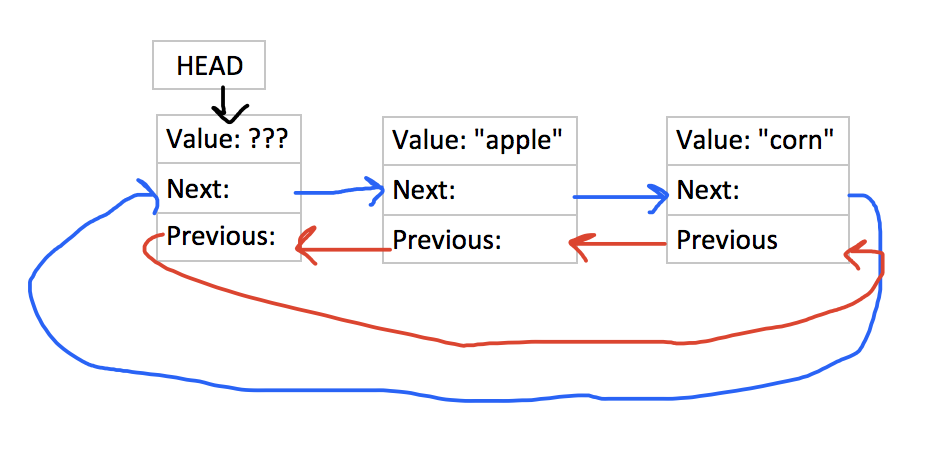
When it’s an empty list, the list will have a dummy node



When it’s a list with one item (let’s say that the item has the value, “apple”



When it’s a list with multiple items (let’s say the items are “apple” and “corn”):



Here’s some pseudocode for non-trivial algorithms:

**Unite:**

…

Repeatedly:

Get a value from s1

If this value doesn’t already exist in result, add the value to result

…

…

Repeatedly:

Get a value from s2

If this value doesn’t already exist in result, add the value to result

**Subtract:**

…

Repeatedly:

Get a value from s2

Erase this value

…

…

Repeatedly:

Get a value that’s remaining in result

If this value is also in s2, erase the value

**Swap:**

Swap head pointers

Swap sizes

**Get:**

Iterate through the list until you reach the position desired

Store the desired value into value

**Contains:**

…

Repeatedly:

Get a value

Does the value match the one desired?

If so, you’re done.

If not, continue process

Here’s a list of test cases that would thoroughly test the program:

**Constructors / Using size and contains:**

// default constructor

Set ss;

// For an empty set:

assert(ss.size() == 0); // test size

assert(ss.empty()); // test empty

assert(!ss.erase("roti")); // nothing to remove

This would be the base case to test the default constructor when there is nothing in the list. This would also test the empty and size functions. Empty() should return true and size() should return zero. It would check to see that the erase() function returns false because there’s nothing to erase.

Set ss; *// ItemType is std::string*

ss.insert("aaa");

ss.insert("ccc");

ss.insert("bbb");

Set ss2(ss);

assert(ss2.size() == 3);

assert(ss2.contains("aaa") && ss2.contains("bbb") && ss2.contains("ccc"));

This would test the copy constructor. If the copy constructor worked properly, then ss2 should contain the same values and size as ss.

**Assignment Operator Overloading:**

Set ss; *// ItemType is std::string*

ss.insert("aaa");

ss.insert("ccc");

ss.insert("bbb");

Set ss2;

ss2 = ss;

assert(ss2.size() == 3);

assert(ss2.contains("aaa") && ss2.contains("bbb") &&ss2.contains("ccc"));

This would test assignment operator overloading. If the assignment operator worked properly, then ss2 should contain the same values and size as ss.

Set ss; *// ItemType is std::string*

ss.insert("aaa");

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

ItemType x = "apple";

assert(ss.get(0, x) && x=="aaa");

assert(ss.erase("aaa"));

assert(!ss.erase("bab"));

}

**Adding / Erasing / Contains / Get items:**

This would the next case to test how the list deals with one item in the list. This would test to see that insert() works correctly and that the size of the list is 1. This can also test the get() function because it should just return the one item in the list. This can also test the erase() function. Erase should be able to erase “aaa” because it’s in the set but it shouldn’t be able to erase “bab” because “bab” is not in the set.

Now after that, we can test multiple items.

Set ss;

ss.insert("aaa");

ss.insert("ccc");

ss.insert("bbb");

ss.insert("aaa");

assert(ss.size() == 3);

ItemType x = "apple";

assert(ss.get(0, x) && x=="aaa");

assert(ss.get(1, x) && x=="bbb");

assert(ss.get(2, x) && x=="ccc");

assert(ss.erase("aaa"));

assert(!ss.contains("aaa"));

assert(ss.size() == 2);

assert(!ss.erase("bab"));

assert(ss.erase("bbb"));

assert(!ss.contains("bbb"));

assert(ss.contains("ccc"));

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

This would test how the list deals with multiple items. We can test that insert() works properly and that it inserts items in alphabetical order with get(). The list should be “aaa”, “bbb”, “ccc”. If insert() works properly, then get() should return “aaa” for the smallest value, “bbb” for the value that’s greater than one value, and “ccc” for the value that’s greater than two values. We can also test erase() by removing one item at a time and checking the size of the list after each erase(). We can also test contains() by making sure that the value erased is no longer in the list.

Set ss; *// ItemType is std::string*

ss.insert("aaa");

ss.insert("ccc");

ss.insert("bbb");

ss.insert("aaa");

assert(ss.size() == 3);

ItemType x = "apple";

assert(ss.get(0, x) && x=="aaa");

assert(ss.get(1, x) && x=="bbb");

assert(ss.get(2, x) && x=="ccc");

assert(ss.erase("aaa"));

assert(ss.size() == 2);

assert(!ss.erase("bab"));

assert(ss.erase("bbb"));

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

This is the same as above except for the highlighted one. This would test that insert() doesn’t insert duplicate values.

Set ss;

ss.insert("aaa");

ss.insert("ccc");

ss.insert("bbb");

ss.insert("aaa");

assert(ss.size() == 3);

ItemType x = "apple";

assert(ss.get(0, x) && x=="aaa");

assert(ss.get(1, x) && x=="bbb");

assert(ss.get(2, x) && x=="ccc");

assert(!ss.get(6, x) && x=="ccc");

assert(ss.erase("aaa"));

assert(!ss.contains("aaa"));

assert(ss.size() == 2);

assert(!ss.erase("bab"));

assert(ss.erase("bbb"));

assert(!ss.contains("bbb"));

assert(ss.contains("ccc"));

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

This is the same as above except for the highlighted one. This is to check that get() doesn’t work with some crazy, out of bounds number.

Set ss; *// ItemType is std::string*

ss.insert("aaa");

ss.insert("ccc");

ss.insert("bbb");

ss.insert("aaa");

ss.insert("bolonge");

ss.insert("bell");

ss.insert("carrot");

ss.insert("apple");

assert(ss.size() == 7);

ItemType x = "apple";

assert(ss.get(0, x) && x=="aaa");

assert(ss.get(1, x) && x=="apple");

assert(ss.get(2, x) && x=="bbb");

assert(ss.get(3, x) && x=="bell");

assert(ss.erase("aaa"));

assert(!ss.contains("aaa"));

assert(ss.size() == 6);

assert(!ss.erase("bab"));

assert(ss.erase("bbb"));

assert(!ss.contains("bbb"));

assert(ss.contains("ccc"));

assert(ss.size() == 5);

This is the same as above except with added values like “bell” and “bolonge”. This is just to test to see if insert() still inserts values that starts with the same letter in alphabetical order.

**Swap:**

Set ss; *// ItemType is std::string*

ss.insert("aaa");

ss.insert("ccc");

ss.insert("bbb");

ss.insert("aaa");

ss.insert("bolonge");

ss.insert("bell");

ss.insert("carrot");

ss.insert("apple");

assert(ss.size() == 7);

Set ss2;

ss2.insert("bbb");

ss2.insert("aaa");

ss.swap(ss2);

assert(ss.size() == 2);

assert(ss2.size() == 7);

assert(ss.contains("aaa") && ss2.contains("bell"));

Swap is tested here to make sure that ss and ss2 actually swapped its values. We do so by checking the size of each. After the size is correct, we check the contents of each by calling contains(). I just picked a random value from both sets and checked it.

Set ss; *// ItemType is std::string*

ss.insert("aaa");

ss.insert("ccc");

ss.insert("bbb");

ss.insert("aaa");

ss.insert("bolonge");

ss.insert("bell");

ss.insert("carrot");

ss.insert("apple");

assert(ss.size() == 7);

Set ss2;

ss2.insert("bill");

ss2.insert("nick");

ss.swap(ss2);

assert(ss.size() == 2);

assert(ss2.size() == 7);

assert(ss.contains("bill") && ss.contains("nick") &&

ss2.contains("bell"));

This is the same as above except with different values in ss2. In the last test case, ss2 contained values that also appeared in ss.

**Unite:**

Set ss; *// ItemType is std::string*

ss.insert("aaa");

ss.insert("ccc");

ss.insert("bbb");

ss.insert("aaa");

ss.insert("bolonge");

ss.insert("bell");

ss.insert("carrot");

ss.insert("apple");

assert(ss.size() == 7);

Set ss2;

ss2.insert("bill");

ss2.insert("nick");

Set ss3;

unite(ss, ss2, ss3);

assert(ss3.size() == 9);

assert(ss3.contains("bill") && ss3.contains("nick") &&

ss3.contains("aaa") && ss3.contains("bell"));

This tests the function unite(). We joined together ss and ss2 in ss3. After uniting the two sets, ss3’s size should be 9 and ss3 should contain values from ss and ss2. I just picked four random values that should be found in ss3. Ideally, we’d check to see that ss3 contains all of them.

Set ss; *// ItemType is std::string*

Set ss2;

Set ss3;

unite(ss, ss2, ss3);

assert(ss3.size() == 0);

If both ss and ss2 are empty, then ss3 should be empty as well.

**Subtract:**

Set ss; *// ItemType is std::string*

ss.insert("aaa");

ss.insert("ccc");

ss.insert("bbb");

ss.insert("aaa");

ss.insert("bolonge");

ss.insert("bell");

ss.insert("carrot");

ss.insert("apple");

assert(ss.size() == 7);

Set ss2;

ss2.insert("bill");

ss2.insert("nick");

Set ss3;

unite(ss, ss2, ss3);

subtract(ss, ss2, ss3);

assert(ss3.size() == 7);

assert(!ss3.contains("bill") && !ss3.contains("nick") &&

ss3.contains("aaa") && ss3.contains("bell"));

If subtract() worked, then ss3 should be just what ss is. ss3 should have 7 items and it shouldn’t contain values from ss2 such as “bill” or “nick”

Set ss; *// ItemType is std::string*

assert(ss.size() == 0);

Set ss2;

Set ss3;

subtract(ss, ss2, ss3);

assert(ss3.size() == 0);

This is just to test subtract of empty sets. You should get 0 for the size.

Set ss; *// ItemType is std::string*

ss.insert("aaa");

ss.insert("ccc");

ss.insert("bbb");

Set ss2;

Set ss3;

unite(ss, ss2, ss3);

subtract(ss, ss2, ss3);

assert(ss3.size() == 3);

assert(ss3.contains("aaa") && ss3.contains("bbb"));

This is just to test subtracting away an empty set. The united set should be untouched, hence its size should still be 3 and it should contain all the values of ss.