Project 2 Report

1.

My doubly linked list for the Set class includes a structure Node. Node contains a variable of type ItemType along with two Node pointers that link the node to the two nodes immediately preceding and succeeding it. My linked list has a head pointer that points to the first node in the list. Each node of the list has next and previous pointers that link it to other nodes. In an empty set, the head pointer points to nullptr. My implementation of the insert function adds new nodes to the beginning of the linked list. Therefore my linked list is ordered from newest nodes to oldest nodes. My doubly linked list does not include a tail pointer, and it is neither circular or uses a dummy node.

2.

**Set()**

Set head pointer to nullptr

Initialize size to 0

**~Set()**

If set isn’t empty

Create a node and iterate through set

Create temp pointer to node to delete

Advance iterator to next node

Delete node pointed to by temp pointer

**Set(const Set& other)**

Set head pointer to nullptr

Initialize size to 0

Create variable of type ItemType

For each node in the other set

Get the value from the node and store in variable

Insert the value into set

**Set& operator=(const Set& rhs)**

If the address of the current set isn’t equal to that of rhs

construct a temp copy of rhs

swap the current set with the copy

return the current set

**bool insert(const ItemType& value)**

If the set doesn’t contain the value

Create new node and initialize with value

Link its next and previous pointers to head and nullptr respectively

If the set isn’t empty, link current head’s previous pointer to new node

Make the new node the head

Increment size and return true

Return false

**bool erase(const ItemType& value)**

If the set contains the value

If set only has one value, delete head and set head to nullptr

Else

Create temp node and traverse list until temp node points to value to erase

If the value to erase is the head

Make the node after it the new head and delete the value

Else

Link the next pointers of nodes before and after the value

If the value isn’t the last node, link the previous pointers

Delete the value

Decrement size and return true

Return false

**bool contains(const ItemType& value) const**

If the set isn’t empty

Traverse through entire list

If a node with matching value is found, return true

Return false

**bool get(int i, ItemType& value) const**

If i satisfies predefined conditions (0 <= i < size)

Create temp node and set to head

Traverse list to the i+1-th node

Set value parameter to that node’s value

Return true

Return false

**void swap(Set& other)**

Create temp variable to store this set’s size

Create temp pointer variable to point to this set’s head

Set this set’s size to the other set’s size

Point this set’s head to the other set’s head

Set the other set’s size to temp size

Point the other set’s head to the temp pointer

**void unite(const Set& s1, const Set& s2, Set& result)**

Create a temp copy of s1

For every value in s2

Get the value from s2

Attempt to insert the value into the temp set using insert

Set result to the temp set

**void subtract(const Set& s1, const Set& s2, Set& result)**

Create a temp set

For every value in s1

Get the value from s1

If s2 doesn’t contain the value, insert the value into the temp set

Set result to the temp set

3.

All tests were performed on a set of strings.

ItemType item = "hello";

Set s;

assert(s.empty()); // check that the set is empty

assert(s.size() == 0); // check that the size of the set is 0

assert(s.erase("hello") == false); // check that we can't erase

assert(s.contains("hi") == false); // check that the set doesn't contain anything

assert(s.insert("Justin"));

assert(!s.empty()); // check that the set isn't empty anymore

assert(s.contains("Justin")); // check that the set contains the inserted value

assert(!s.insert("Justin")); // check that duplicate values can't be inserted

assert(!s.erase("Ryan")); // check that we can't erase a value not in the set

assert(s.insert("Michael"));

assert(s.erase("Michael")); // check erasing from the top of the set

assert(s.insert("Michael"));

assert(s.erase("Justin")); // check erasing from the bottom of the set

assert(s.insert("Justin"));

assert(!s.get(5, item)); // check that we can't get anything when i is out of bounds

assert(item == "hello"); // check that value isn't changed when get fails

assert(s.get(1, item));

assert(item == "Justin" || item == "Michael"); // check that we get one of the values

assert(s.insert("Calvin"));

assert(s.erase("Calvin")); // check erasing from the middle of the set

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

Set copy1(s); // testing copy constructor

assert(copy1.size() == 2); // check that the set has the right number of values

Set copy2 = s; // testing copy constructor

Set blank;

copy1 = blank; // Check that assignment operator works for empty sets

assert(copy1.empty()); // check that copy set is still empty

Set newBlank(blank);

assert(newBlank.empty()); // check that copy constructor works with empty sets

Set node;

node.insert("node");

Set copyNode(node); // checks that copy constructor works with a node

assert(copyNode.contains("node"));

Set swapped;

s.swap(swapped); // tests swapping with an empty set

assert(s.empty()); // check that the number of elements has been swapped

swapped.swap(s); // swaps back

assert(swapped.empty()); // check that the number of elements has been swapped again

Set s1;

s1.insert("a");

s1.insert("b");

s1.insert("c");

s1.insert("d");

Set s2;

s2.insert("b");

s2.insert("c");

s2.insert("e");

s2.insert("f");

Set result;

result.insert("junk"); // checks that unite works correctly when result is originally not empty

result.insert("more junk");

unite(s1, s2, result);

assert(result.size() == 6); // checks that result is the right size

assert(result.contains("a")); // checks that result contains the correct values

assert(result.contains("b"));

assert(result.contains("c"));

assert(result.contains("d"));

assert(result.contains("e"));

assert(result.contains("f"));

unite(s2, s2, result); // checks that unite works with s1 and s2 being the same set

assert(result.size() == 4); // checks that result is the right size

assert(result.contains("b")); // checks that result contains the correct values

assert(result.contains("c"));

assert(result.contains("e"));

assert(result.contains("f"));

unite(s1, s2, s1); // checks that unite works with s1 and result being the same set

assert(s1.size() == 6); // checks that s1 is the right size

assert(s1.contains("a")); // checks that s1 contains the correct values

assert(s1.contains("b"));

assert(s1.contains("c"));

assert(s1.contains("d"));

assert(s1.contains("e"));

assert(s1.contains("f"));

s1.erase("e");

s1.erase("f");

unite(s1, s2, s2); // checks that unite works with s2 and result being the same set

assert(s2.size() == 6); // checks that s2 is the right size

assert(s2.contains("a")); // checks that s2 contains the correct values

assert(s2.contains("b"));

assert(s2.contains("c"));

assert(s2.contains("d"));

assert(s2.contains("e"));

assert(s2.contains("f"));

Set s3;

s3.insert("g");

s3.insert("h");

s3.insert("i");

s3.insert("j");

Set s4;

s4.insert("i");

s4.insert("j");

s4.insert("k");

s4.insert("l");

Set result2;

result2.insert("junk");

subtract(s3, s4, result2);

assert(result2.size() == 2); // check that subtract creates set with right size

assert(result2.contains("g")); // check that result2 contains the correct values

assert(result2.contains("h"));

subtract(s3, s3, result2); // check that subtract works when s1 and s2 are the same set

assert(result2.empty());

subtract(s3, s4, s3); // check that subtract works when s1 and result are the same set

assert(s3.size() == 2);

assert(s3.contains("g"));

assert(s3.contains("h"));

s3.insert("i");

s3.insert("j");

subtract(s3, s4, s4); // check that subtract works when s2 and result are the same set

assert(s4.size() == 2);

assert(s4.contains("g"));

assert(s4.contains("h"));

cout << "Passed all tests." << endl;