**Description:**

An empty set:

m\_head: m\_tail:

m\_head and m\_tail point to dummy nodes marking the start and the end of the set. The above rectangle represents the location that saves m\_value. The middle one represents the m\_next pointer. For the starting dummy node, m\_next points to the ending dummy node and for the ending dummy node, m\_next is a null pointer. The rectangle below these two represents the m\_previous pointer. For the starting dummy node, m\_previous is a null pointer, and for the ending dummy node, m\_previous points to the starting dummy node.

A typical set:

m\_head Node1 Node2 m\_tail

This picture shows the circular structure of this doubly-linked list. Only values in nodes are considered in operations. While the function insert is executed, a new node with intended value will be placed directly after the starting dummy node.

**Pseudocode:**

**unite**:

Declare a variable of Itemtype, value, to be the call-by-reference parameter to the get function

Declare a int variable, n, and initialize it to the size of the original result set

Declare a int variable, eraseTime, to record times that the erase function is called and initialize it to 0

Declare a int variable, pos, to keep track of all nodes one by one in the result set and initialize it to 0

If pos is less than the nodes remaining in the set

Repeatedly:

Call the get function to get the value of certain node in the result set

if both s1 and s2 do not contain that value, erase that value from the result set, increment eraseTime and reset pos to 0 since the size of result set changes

if s1 or s2 contains that value, increment pos to check the next node in the result set

Declare a variable of Itemtype, value\_s1, to be the call-by-reference parameter to the get function

Declare a int variable, pos\_s1, to keep track of all nodes one by one in s1 set and initialize it to 0

Repeatedly:

Call the get function to get the value of certain node in s1 set

If the result set does not contain that value, insert that value into the result set

Increment pos\_s1 until pos\_s1 equals the size of s1 set

Declare a variable of Itemtype, value\_s2, to be the call-by-reference parameter to the get function

Declare a int variable, pos\_s2, to keep track of all nodes one by one in s2 set and initialize it to 0

Repeatedly:

Call the get function to get the value of certain node in s2 set

If the result set does not contain that value, insert that value into the result set

Increment pos\_s2 until pos\_s2 equals the size of s2 set

**Subtract:**

Declare a variable of Itemtype, value, to be the call-by-reference parameter to the get function

Declare a int variable, n, and initialize it to the size of the original result set

Declare a int variable, eraseTime, to record times that the erase function is called and initialize it to 0

Declare a int variable, pos, to keep track of all nodes one by one in the result set and initialize it to 0

If pos is less than the nodes remaining in the set

Repeatedly:

Call the get function to get the value of certain node in the result set

if both s1 and s2 do not contain that value, erase that value from the result set, increment eraseTime and reset pos to 0 since the size of result set changes

if s1 or s2 contains that value, increment pos to check the next node in the result set

Declare a variable of Itemtype, value\_s1, to be the call-by-reference parameter to the get function

Declare a int variable, pos\_s1, to keep track of all nodes one by one in s1 set and initialize it to 0

Repeatedly:

Call the get function to get the value of certain node in s1 set

If both the result set and s2 set do not contain that value, insert that value into the result set

Increment pos\_s1 until pos\_s1 equals the size of s1 set

**Swap:**

Declare two pointers, tempHead and tempTail, and initialize them to m\_head and m\_tail

Let m\_head and m\_tail point to the dummy-starting node and the dummy-ending node of other set.

Let other.m\_head and other m\_tail point to the dummy-starting node and the dummy-ending node of this set.

Exchange the size of these two sets

**Insert:**

Declare a new node pointer and dynamically allocate a node

If haven’t contained that value, assign the intended value to m\_value of node

Make the m\_previous of the node after the dummy-starting node point to this new node

Make the m\_next of the new node point to the node originally after the dummy-starting node

Make the m\_next of the dummy-starting node point to this new node

Make the m\_previous of the new node point to the dummy-starting node

Adjust the size

Return true

If have contained that value, return false

**Erase:**

Declare and initialize a pointer to the node, which contains our objective value

If the m\_next of that node is nullptr, meaning it’s the dummy-ending node, return false

Otherwise, make the m\_next of the node before our objective node point to the node after our objective node

Make the m\_previous of the node after our objective node point to the node before our objective node

Adjust the size

Delete the objective node from our doubly-linked list

Return true

**Test Cases:**

#include "Set.h"

#include <iostream>

#include <cassert>

using namespace std;

#define CHECKTYPE(f, t) { auto p = (t)(f); (void)p; }

void thisFunctionWillNeverBeCalled()

{

Set();

(Set(Set()));

CHECKTYPE(&Set::operator=, Set& (Set::\*)(const ItemType&));

CHECKTYPE(&Set::empty, bool (Set::\*)() const);

CHECKTYPE(&Set::size, int (Set::\*)() const);

CHECKTYPE(&Set::insert, bool (Set::\*)(const ItemType&));

CHECKTYPE(&Set::erase, bool (Set::\*)(const ItemType&));

CHECKTYPE(&Set::contains, bool (Set::\*)(const ItemType&) const);

CHECKTYPE(&Set::get, bool (Set::\*)(int, ItemType&) const);

CHECKTYPE(&Set::swap, void (Set::\*)(Set&));

CHECKTYPE(unite, void (\*)(const Set&, const Set&, Set&));

CHECKTYPE(subtract, void (\*)(const Set&, const Set&, Set&));

}

void test()

{

}

int main()

{

Set ss;

assert(ss.insert("roti"));

assert(ss.insert("pita"));

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

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

ItemType x = "focaccia";

assert(ss.get(0, x) && (x == "roti" || x == "pita"));

Set new\_s1;

Set new\_s2;

Set new\_result;

unite(new\_s1, new\_s2, new\_result);

assert(new\_s1.size() == 0 && new\_s2.size() == 0 && new\_result.size() == 0);

new\_s1.insert("ucla");

new\_s1.insert("usc");

new\_s1.insert("ucb");

new\_s1.insert("ucsd");

assert(new\_s1.size() == 4);

//unite function

unite(new\_s1, new\_s2, new\_result);

assert(new\_result.size() == 4 && new\_result.contains("ucla") && new\_result.contains("usc") && new\_result.contains("ucb") && new\_result.contains("ucsd"));

new\_result.erase("ucla");

new\_result.erase("usc");

assert(new\_result.size() == 2);

new\_result.insert("ucsb");

//copy constructor

Set new\_copy1(new\_result);

assert(new\_copy1.size() == 3);

//assignment operator

Set new\_copy2 = new\_result;

//subtract function

subtract(new\_s1, new\_copy1, new\_result);

assert(new\_result.size() == 2 && new\_result.contains("ucla") && new\_result.contains("usc"));

unite(new\_copy2, new\_s1, new\_s2);

assert(new\_s2.size() == 5);

subtract(new\_copy2, new\_s2, new\_result);

assert(new\_result.empty());

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

}