**Description:**

My doubly-linked list implementation has a dummy node, and is circular (next pointer of the tail points to the head (dummy node) and prev pointer of head points to the tail). Each Node of the list contains a value of ItemType, next and prev pointers that link it to the list. Nodes are sorted from smallest to largest according to the value they store.

**Pseudocode:**

Set::Set(int capacity): m\_size(0), m\_capacity(capacity)

{

Initialize head as a new Node

Set next and prev pointer (all point to head itself)

Set value as Empty (e.g. “” for std::string type )  
}

Set::~Set()

{

Traverse the list (break while loop when next point to head)

{ Define a temporary Node P that store next Node after head

Link the Node after P to head as next

Delete P  
}

Delete head

}

Set::Set(const Set& other) : m\_size(other.m\_size), m\_capacity(other.m\_capacity)

{

Initialize head of “this” set

Traverse “other” set

Insert each value stored in the Nodes to “this” set

m\_size = other.m\_size again

}

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

{

If this and rhs are different Set

Define a temporary set tmp which copies rhs

Swap this and temp (now this has all nodes of temp, in other words, rhs)  
return \*this

}

bool Set::insert(const ItemType& value)

{

If set is full or value already exist

Return false

Traverse the list

Stop when next node contains value greater than the value we insert or when we move to the tail

Create a new Node that store the value we insert

link the next of the point we stopped as the next of new Node

link the point we stopped as the prev of new Node

increment m\_size

return true

}

bool Set::erase(const ItemType& value)

{

If the list doesn’t contain the value

Return false

Else

{

Create a temporary Node p that store the node contains the value we want to erase

Link the node before p and next to p together (cut p off the list)

Delete p

Decrease m\_size

Return true  
}

}

void Set::swap(Set& other)

{

Create temporary head tempHead that store this->head

Swap this->head and other->head

Create temporary size tempSize that store this->m\_size

Swap this->m\_size and other->m\_size

Create temporary capacity tempCapacity that store this->capacity

Swap this->capacity and other->capacity  
}

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

{  
 create a new Set tmp that copies s1

Create a variable of ItemType called value

Repeatedly (stop when we cannot get new value from s2)

{

If tmp doesn’t contain the new value we get

Insert the new value to tmp

Move to next node of the list of s2  
 }

Copy tmp to result

}

void butNot(const Set& s1, const Set& s2, Set& result)

{

Create an empty Set tmp that contains nothing except the dummy node

Create a variable of ItemType called value

Repeatedly (stop when we cannot get new value from s1)

{

If s2 doesn’t contain the new value we get

Insert the new value to tmp

Move to next node of the list of s1

}

Copy tmp to result

}

**Test cases:**

// 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

// copy constructor

Set s1;

// for an empty set:

Set s2 (s1);

assert(s2.size() == 0); // size = 0

assert(!s2.erase("h")); // nothing to remove

// for a list that is not empty

Set s3;

s3.insert("a");

s3.insert("b");

s3.insert("c");

Set s4 = s3;

assert(s4.size() == 3); // size is matched

assert(s4.contains("a") && s4.contains("b") && s4.contains("c")); //items are matched

// assignment operator

Set s5;

s5.insert("a");

// for the same set

Set s6(s5);

s5 = s6;

assert(s5.size() == 1);

assert(s5.contains("a"));

// for an empty set

Set s7;

s5 = s7;

assert(s5.size() == 0);

// for a normal set

Set s8;

s8.insert("H");

s8.insert("rr");

s5 = s8;

assert(s5.size() == 2);

assert(s5.contains("H") && s5.contains("rr"));

// empty

Set s9;

assert(s9.empty());

assert(!s5.empty());

//insert

//empty list

Set i1;

i1.insert("b");

assert(i1.size() == 1 && i1.contains("b"));

// new item should be added at the top

i1.insert("a");

assert(i1.size() == 2);

//new item should be added at the bottom

i1.insert("c");

assert(i1.size() == 3);

// new item already in the list

Set i2;

i2.insert("b");

assert(!i2.insert("b") && i2.size() == 1);

// a normal insert

i1.insert("bb");

assert(i1.size() == 4);

// if the new item is the same as the first item in the list

assert(!i1.insert("a"));

// if the new item is the same as the last item in the list

assert(!i1.insert("c"));

//erase

Set e1;

// an empty list

assert(!e1.erase("a")); //nothing to erase

// erase the first item and the last item

e1 = i1;

e1.erase("a");

assert(e1.size() == 3);

e1.erase("c");

assert(e1.size() == 2);

// erase the only item in the list

Set e2;

e2.insert("a");

e2.erase("a");

assert(e2.size() == 0);

// a normal erase

e2 = i1;

e2.erase("bb");

assert(e2.size() == 3);

// if there is no such value in the list

assert(!e2.erase("aa"));

//contains

Set c1 = i1; // a->b->bb->c

// test the first item

assert(c1.contains("a"));

// test the last item

assert(c1.contains("c"));

// test an item in the middle

assert(c1.contains("b"));

// test en empty list

Set c2;

assert(!c2.contains("a"));

// test a list with only one item

c2.insert("a");

assert(c2.contains("a") && !c2.contains("b"));

// get

Set g1 = i1;

ItemType temp = "h”;

// pos is out of range

assert (!g1.get(-1, temp) && !g1.get(4, temp) && temp == "h");

// get the first, last, and middle item

g1.get(0, temp);

assert(temp == "a");

g1.get(2, temp);

assert (temp == "b");

g1.get(3, temp);

assert(temp == "c");

// swap

Set sw1;

sw1.insert("a");

sw1.insert("b");

sw1.insert("c");

Set sw2;

sw2.insert("d");

sw2.insert("e");

sw1.swap(sw2);

// test if the sizes are swapped

assert (sw1.size() == 2 && sw2.size() == 3);

// test if the items are swapped

temp = "hhh";

sw1.get(0, temp);

assert(temp == "d");

sw1.get(1, temp);

assert(temp == "e");

temp = "hhh";

sw1.get(2, temp);

assert (temp == "hhh");

sw2.get(0, temp);

assert(temp == "a");

sw2.get(1, temp);

assert(temp == "b");

sw2.get(2, temp);

assert(temp == "c");

temp = "hhh";

sw1.get(3, temp);

assert(temp == "hhh");

// unite

// two sets are both empty

Set u1,u2;

Set uRes;

unite(u1, u2, uRes);

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

//u1 is empty

u2.insert("a");

unite(u1, u2, uRes);

assert(uRes.size() == 1 && uRes.contains("a"));

//u2 is empty

u2.erase("a");

u1.insert("A");

unite(u1, u2, uRes);

assert(uRes.size() == 1 && uRes.contains("A"));

//whether uRes can make itself empty

assert(!uRes.contains("a"));

//whether it works in normal cases

u1.erase("A");

u1.insert("a");

u1.insert("b");

u1.insert("c");

u2.insert("b");

u2.insert("c");

u2.insert("d");

unite(u1, u2, uRes);

assert(uRes.size() == 4 && uRes.contains("a") && uRes.contains("b") && uRes.contains("c") && uRes.contains("d"));

// result is u1

unite(u1, u2, u1);

assert(u1.size() == 4 && u1.contains("a") && u1.contains("b") && u1.contains("c") && u1.contains("d"));

// result is u2

u1.erase("d");

unite(u1, u2, u2);

assert(u2.size() == 4 && u2.contains("a") && u2.contains("b") && u2.contains("c") && u2.contains("d"));

//butNot

Set b1, b2, bRes;

// two sets are both empty

butNot(b1, b2, bRes);

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

// b1 is empty

b2.insert("a");

butNot(b1, b2, bRes);

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

// b2 is empty

b2.erase("a");

b1.insert("A");

butNot(b1, b2, bRes);

assert(bRes.size() == 1 && bRes.contains("A"));

// b1 and b2 contain same items

b2.insert("A");

butNot(b1, b2, bRes);

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

// whether it works in normal case

b1.insert("B"); // b1: A->B

b2.erase("A");

b2.insert("B");

b2.insert("C"); // b2: B->C

butNot(b1, b2, bRes);

assert(bRes.size() ==1 && bRes.contains("A"));

// result is b1

butNot(b1, b2, b1);

assert(b1.size() == 1 && b1.contains("A"));

//result is b2

b1.insert("B");

butNot(b1, b2, b2);

assert(b2.size() == 1 && b2.contains("A"));