Project 2 Report

Marvin Deng

1. I used a circular doubly linked list with a dummy node pointed to by head. Each node has a pointer to the previous and next nodes. The last node of the list points to the dummy node and the dummy node points to the last node. An empty list has the dummy node’s next and previous pointers pointing back to itself. There is no nullptr or tail pointer.

A picture containing text, whiteboard

Description automatically generatedA drawing on a piece of paper

Description automatically generated with medium confidence

1. Pseudocode

**Copy Constructor**

Copy the length of the other set

Create dummy pointer of circular doubly linked list pointing to itself

Create an iterator pointer to start of linked list

Create an iterator pointer to start of other linked list

Repeatedly:

create new Node to add to the list

assign the other value to the new node’s value

Assign the current node’s next pointer to the new node

Assign the new node’s previous pointer to the current node

Assign the new pointer’s next node to head

Assign head’s previous pointer to the new pointer

Shift both iterator pointers forward

**Destructor**

Repeatedly while the list is not empty:

Temporary node stores node that will be deleted

Head’s next pointer points to node after the deleted node

Set the previous pointer of the node after the deleted node to head

Set the next and previous pointer of the deleted node to null (protection)

Delete the node

Decrement the length

Delete the dummy node

**Operator=**

If current set isn’t the same as rhs:

Create a temporary Set temp to store the values of rhs

Swap lhs and temp

Return pointer to current Set

**Insert**

Return false if set already contains value

Create an iterator pointer curr to start of linked list

Initialize space for a new node and set the new node’s value

Repeatedly while curr doesn’t reach head:

Break if value is less than the current value

Shift curr forward

Link curr’s previous node to the new node

Link curr and the new node

Increment length

Return true;

**Contains**

Return false if the value is not in the list

Create dummy pointer curr to start of list

Repeatedly:

If the current value equals value:

Make curr’s previous node point to the node after the deleted node

Make the next node point to the node before the deleted node

Set the deleted node's pointers to null (protection)

Delete curr

Decrement length

Break

Return true

**Get**

Return false if pos less than one or greater than or equal to length

Set iterator pointer p to first node in the list

Set counter to 0

Repeatedly:

If counter equals i:

Set value equal to value of the p

break

Increment counter

Move p forward in the list

Return true

**Swap**

Swap lengths

Swap pointers

**Unite**

Create temporary set temp

copy s1 into temporary set using assignment operator

Repeatedly while i is less than s2’s length:

Get value from s2 and assign it to item

Insert item into temp set

Assign temp to result

**ButNot**

Create temporary set

Repeatedly:

Get value from s1 and assign it to item

Only insert into temp if item is not in s2

Assign temp to result

1. Test Cases

**Test Copy Constructor**

Set ss;

// Test CC for empty sets

Set ss2(ss);

assert(ss.size() == 0 && ss2.size() == 0);

assert(ss.insert("a") && ss.insert("b") && ss.insert("c"));

// Test CC with nonempty set

Set ss3(ss); // intiialize ss with the values of ss2

// ss3 has the values of ss

assert(ss3.contains("a") && ss3.contains("b") && ss3.contains("c") && ss3.size() == 3);

// ss remains unchanged

assert(ss.contains("a") && ss.contains("b") && ss.contains("c") && ss.size() == 3);

// Test ss and ss3 are independent of each other

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

assert(!ss3.contains("d"));

**Test operator=() -** tests operator= (and by extension swap and copy constructor)

Set ss, ss2, empty;

// assign empty set to empty set

ss = ss2;

assert(ss.size() == 0 && ss2.size() == 0);

assert(ss.insert("a") && ss.insert("b") && ss.insert("c"));

// assign non-empty set into empty set

ss2 = ss;

assert(ss.contains("a") && ss.contains("b") && ss.contains("c") && ss.size() == 3);

// ss stays the same

assert(ss2.contains("a") && ss2.contains("b") && ss2.contains("c") && ss2.size() == 3);

// ss2 should be identical to ss

// assign empty set into non-empty set

// ss2: a, b, c

ss2 = empty;

assert(ss2.empty()); // ss2 is now empty

assert(empty.empty()); // empty remains empty

assert(ss2.insert("d") && ss2.insert("e") && ss2.insert("f"));

// assign non-empty set into non-empty set

ss2 = ss; // ss2 now contains a, b, c

// ss stays the same

assert(ss.contains("a") && ss.contains("b") && ss.contains("c") && ss.size() == 3);

// ss2 should be identical to ss

assert(ss2.contains("a") && ss2.contains("b") && ss2.contains("c") && ss2.size() == 3);

**Test empty()**

Set ss;

assert(ss.empty()); // tests empty default constructor

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

assert(!ss.empty()); // shouldn't be ampty after inserting a node

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

assert(ss.empty()); // should be empty after erasing a node

**Test size()** - Tests size, insert, and erase

Set ss;

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

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

assert(ss.size() != 0 && ss.size() == 1); // node abc should have been added, size = 1

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

assert(ss.size() == 2); // another node inserted, size = 2

assert(!ss.insert("abc"));// attempt to insert duplicate

assert(ss.size() != 3 && ss.size() == 2); // duplicate is not added, size = 2

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

assert(ss.size() != 3 && ss.size() == 1); // abc is removed, size = 1

assert(!ss.erase("abc")); // attempts to erase nonexistant element

assert(ss.size() == 1); // size remains as 1

**Test contains()**

void testContains() {

Set ss;

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

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

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

// check contains

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

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

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

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

// erase all nodes in linked list

assert(ss.erase("def") && ss.size() == 2);

assert(ss.erase("abc") && ss.size() == 1);

assert(ss.erase("ghi") && ss.size() == 0);

// check that nodes are erased

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

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

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

assert(ss.empty());

**Test Get()**

Set ss;

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

assert(!ss.insert("abc")); // attempt to insert duplicate

assert(ss.insert("ghi")); // not inserted in lexiographical order, tests inserting in order

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

// check get

ItemType x = "";

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

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

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

// check get() after deleting items

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

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

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

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

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

assert(!ss.get(-6, x)); // checks when pos is negative

assert(!ss.get(1, x)); // check when pos == ss.size()

assert(!ss.get(100, x)); // check when pos > ss.size()

**Test swap()**

Set ss;

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

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

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

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

// Check swap with empty set

Set ss2;

ss2.swap(ss);

// ckeck ss2 swapped values

assert(ss2.contains("abc"));

assert(ss2.contains("def"));

assert(ss2.contains("ghi"));

assert(ss2.size() == 3); // s2 now contains ss values

// check ss swapped values

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

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

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

assert(ss.empty()); // ss now contians the empty set

// Values inserted to ss aren't inserted to ss2

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

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

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

assert(!ss2.contains("yo"));

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

// Check swap between non empty sets

ss2.swap(ss);

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

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

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

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

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

assert(ss2.contains("yo"));

assert(!ss2.contains("abc"));

assert(ss2.size() == 1);

**Test unite()**

Set ss, ss2, ss3, res;

// checks when s1, s2, and res are empty

unite(ss, ss2, res);

assert(ss.empty() && ss2.empty() && res.empty());

// ss: a, b, c

assert(ss.insert("a") && ss.insert("b") && ss.insert("c"));

// ss2: b, c, d, e

assert(ss2.insert("b") && ss2.insert("c") && ss2.insert("d") && ss2.insert("e"));

// ss3: d, e, f

assert(ss3.insert("d") && ss3.insert("e") && ss3.insert("f"));

// Checks when s1, s2, and result aren't empty

unite(ss, ss2, ss3);

assert(ss3.contains("a") && ss3.contains("b"));

assert(ss3.contains("c") && ss3.contains("d") && ss3.contains("e")); // ss3: a, b, c, d, e

assert(ss3.size() == 5);

// Checks when result is empty

unite(ss, ss2, res);

assert(res.contains("a") && res.contains("b"));

assert(res.contains("c") && res.contains("d") && res.contains("e"));

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

// Checks when s2 is empty and result isn't empty

unite(ss, res, ss2);

assert(ss2.contains("a") && ss2.contains("b"));

assert(ss2.contains("c") && ss2.contains("d") && ss2.contains("e"));

assert(ss2.size() == 5);

// Checks when s1, s2, and result are the same set

unite(ss, ss, ss);

assert(ss.contains("a") && ss.contains("b") && ss.contains("c"));

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

// Checks when s1 and result are the same set

unite(ss, ss2, ss);

assert(ss.contains("a") && ss.contains("b"));

assert(ss.contains("c") && ss.contains("d") && ss.contains("e"));

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

}

**Test butNot()**

Set ss, ss2, res;

// Tests when s1, s2, and result are empty

butNot(ss, ss2, res);

assert(ss.empty() && ss2.empty() && res.empty());

// ss: a, b, c, d

assert(ss.insert("a") && ss.insert("b") && ss.insert("c") && ss.insert("d"));

// ss2: b, c, e

assert(ss2.insert("b") && ss2.insert("c") && ss2.insert("e"));

// Test when result is empty

butNot(ss, ss2, res);

assert(res.contains("a") && res.contains("d"));

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

// Test when s1, s2, and result are not empty

// ss: a, b, c, d, res: a, d

butNot(ss, res, ss2);

assert(ss2.contains("b") && ss2.contains("c"));

assert(ss2.size() == 2);

// Checks when s1 and result are the same set

butNot(ss, res, ss);

assert(ss.contains("b") && ss.contains("c"));

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

// Checks when s1, s2, and result are the same set

// ss: b, c

butNot(ss, ss, ss);

assert(!ss.contains("b") && !ss.contains("c"));

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