Artiom Arutiunov

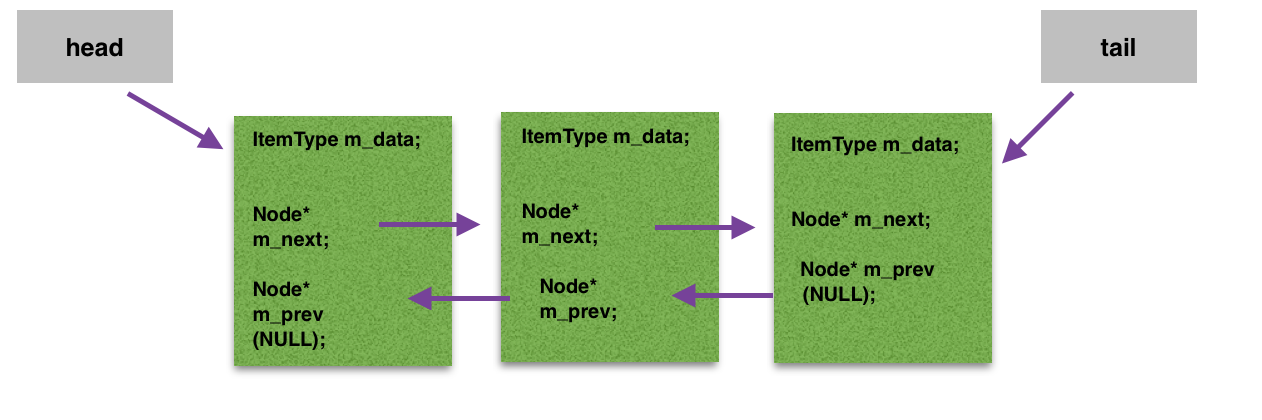
ID: 504597668

Professor Smallberg

4/19/16

Project 2 Report

**Design**

I decided to use a doubly-linked list with a head pointer pointing to the first element in the list and a tail pointer pointing to the last element in the list. The list is not circular; I instead adjusted the head and tail pointers in the cases that I added or deleted a node from the list that was either first or last. The nodes themselves contain three data members: m\_data for the actual data value; a pointer of Node type called m\_next that points to the subsequent node, and a pointer of Node type called m\_prev that points to the node before it. In addition, outside of the struct Node but inside the Set class, I had a pointer of type Node called head and a pointer of type node called tail. The last private member variable of class Set was a variable of type int called numelements that counted the amount of elements the linked list had. Whenever I added a new node, I designed my function to add it to the front of the list (meaning the head pointer now points to it) for convenience. A visual design of my linked list can be found below:

**Pseudocode**

*Destructor:* Set::~Set()

While the head does not point to null:

Temporary node pointer equals next node after head

Delete the head

Set head equal to temp

*Copy Constructor:* Set::Set(const Set& other)

Set number elements equal to other’s

If other is not an empty set

Create a new node

Assign data pointed to head to other’s data pointed to by head

Create temporary node pointer equal to other’s head

While temporary’s next pointer is not null

Call insert function with data from temp’s next pointer

Assign temporary pointer equal to temporary’s next pointer

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

Check that the two sets aren’t the same

Create a temporary set via copy constructor

Call swap function on temporary

Return dereferenced this pointer

*Insert:* bool Set::insert(const ItemType& value)

Check if value is already in the set

Save the old head

If the list isn’t empty

Create a new node via head

Assign value

Assign next pointer by equating it to temp

Assign previous pointer by equating it to head

Increment the amount of elements

Else

Create a new node via three-parameter constructor

Assign nullpointer to head and previous pointers, value from parameter

Increment the amount of elements

*Erase:* bool Set::erase(const ItemType& value)

Check if value is in the list

Create node pointer curr

Identify the element in list with target value using curr

If curr is pointing to element with target value

Check if it’s the first element in the list

Head now points to second element in the list

Second element is now first

Decrement number of elements

Check if it’s the last element in the list

Tail now points to second to last element

Second to last element is now last

Decrement number of elements

Otherwise it’s in the middle

Previous node’s next pointer equal to next’s previous pointer

Next’s previous pointer is equal to curr’s previous pointer

Decrement number of elements

Delete curr

*Contains:* bool Set::contains(const ItemType& value) const

Create a node pointer p

Cycle until we either encounter value or end of the list

If we have found the value

Return true

Else

Return false

*Get:* bool Set::get(int pos, ItemType& value) const

Check that pos parameter is valid

Create node pointer temp equal to head

Cycle through list until we encounter desired position

Assign data from element to value

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

Create set temporary

Iterate until reached size of s1

Create placeholder of ItemType

Call get function on iterator, placeholder

Call insert function on placeholder

Iterate until reached size of s2

Create placeholder of ItemType

Call get function on iterator, placeholder

Call insert function on placeholder

Assign result parameter to temporary using copy constructor

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

Create a set temporary equal to s1 using copy constructor

Iterate until reached size of s2

Create placeholder of ItemType

Call get function on iterator, placeholder

If temporary contains a value equal to placeholder

Erase that node

Assign result to temporary using copy constructor

**Test cases**

All tests were executed with typedef defined as an integer, making it easy to fill up Sets and remove Nodes.

//// TESTING DEFAULT CONSTRUCTOR

Set ss;

ItemType value;

// For an empty set:

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

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

assert(!ss.erase(1)); // nothing to remove

assert(!ss.contains(2)); // no elements in the list

assert(!ss.get(-1, value)); // no elements to get

// TESTING COPY CONSTRUCTOR

Set ss1;

for (int i = 0; i < 21; i++)

{

ss1.insert(i);

}

Set ss2(ss1); // testing copy constructor

assert(ss2.size() == 21);

assert(ss1.contains(15));

assert(ss1.contains(20));

Set ss3;

Set ss4(ss3); // testing copy constructor if initial Set is empty

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

assert(!ss3.contains(10));

// TESTING ASSIGNMENT OPERATOR

Set ss5;

for (int i = 0; i < 15; i++)

{

ss5.insert(i);

}

Set ss6 = ss5; // testing regular assingment operator settings

assert(ss6.size() == 15);

Set ss7;

for (int i = 0; i < 10; i++)

{

ss7.insert(i);

}

ss7 = ss7; // testing if you can assign object to itself

assert (ss7.size() == 10);

Set test3;

for (int i = 15; i < 25; i++)

test3.insert(i);

Set test4;

for (int i = 16; i < 25; i++)

test4.insert(i);

test4 = test3; // testing if old nodes are properly overridden

assert(test4.size() == 10);

assert(test4.contains(15));

// TESTING EMPTY

Set ss8;

assert(ss8.empty());

// TESTING INSERT FUNCTION

Set ss9;

for (int i = 0; i < 15; i++)

ss9.insert(i);

assert(!ss9.insert(13)); // already contains

assert(ss9.insert(25)); // regular insert

assert(ss9.size() == 16);

Set test5;

test5.insert(5); // regular insert as first element

assert(test5.size() == 1);

test5.insert(6); // regular insert as second element

assert(test5.size() == 2);

test5.insert(7); // regular insert as third element

assert(test5.contains(7));

// TESTING ERASE FUNCTION

Set ss11;

for (int i = 0; i < 15; i++)

ss11.insert(i);

for (int i = 0; i < 3; i++)

ss11.erase(i);

assert(ss11.size() == 12); // testing erase's effect on list's size

assert(!ss11.erase(69)); // not conatined

assert(ss11.size() == 12);

ss11.erase(14); // erase from the back

ss11.erase(3); // erase from the front

assert(ss11.size() == 10);

ss11.erase(8); // erase from the middle

assert(ss11.size() == 9);

// TESTING CONTAINS FUNCTION

Set ss12;

ss12.insert(1);

assert(!ss12.contains(0));

assert(ss12.contains(1));

// TESTING GET FUNCTION

Set ss13;

for (int i = 0; i < 30; i++)

ss13.insert(i);

int var(0);

assert(ss13.get(27, var));

assert(var == 28); // testing var's value

assert(!ss13.get(-20, var)); // testing that you can't have negative pos

// TESTING SWAP FUNCTION

Set ss14;

for (int i = 0; i < 10; i++)

ss14.insert(i);

ss14.swap(ss14); // testing swap function on itself

Set ss15;

ss15.insert(1);

ss15.swap(ss14); // testing regular swap function

assert(ss14.size() == 1);

assert(ss14.contains(1));

assert(!ss14.contains(2));

assert(ss15.size() == 10);

assert(ss15.contains(7));

Set ss16; // testing swap function with an empty list

ss16.swap(ss15);

assert(ss15.size() == 0);

// TESTING UNITE FUNCTION

Set ss17;

for (int i = 0; i < 20; i++)

ss17.insert(i);

unite(ss17, ss17,ss17); // testing for unite function called on itself

assert(ss17.size() == 20);

assert(ss17.contains(19));

Set ss18;

Set result18;

for (int i = 0; i < 21; i++) // testing for unite function + empty result set

ss18.insert(i);

unite(ss17, ss18, result18);

assert(result18.size() == 21);

assert(result18.contains(19));

Set ss19;

Set ss20;

Set result19;

for (int i = 0; i < 50; i++) // filling up result before

result19.insert(i);

ss19.insert(2009);

for (int i = 70; i < 95; i++)

ss20.insert(i);

unite(ss19, ss20, result19);

assert(result19.size() == 26 );

assert(! result19.contains(0));

// TESTING SUBTRACT FUNCTION

Set ss21;

for (int i = 0; i < 25; i++)

ss21.insert(i);

subtract(ss21, ss21, ss21); // testing with all 3 parameters as one set

assert(ss21.size() == 0);

assert(! ss21.contains(20));

assert(ss21.empty());

Set ss22;

for (int i = 0; i < 5; i++)

ss22.insert(i);

Set ss23;

for (int i = 0; i < 4; i++)

ss23.insert(i);

Set result22;

subtract(ss22, ss23, result22); // testing normal subtract function functionality

assert(result22.size() == 1);

assert(result22.contains(4));

Set ss24;

for (int i = 0; i < 10; i++)

ss24.insert(i);

Set ss25;

for (int i = 0; i < 6; i++)

ss25.insert(i);

Set result23;

for (int i = 25; i < 35; i++)

result23.insert(i);

subtract(ss24, ss25, result23); // testing subtract function if result list already contains elements

assert(result23.size() == 4);

assert(!result23.contains(30));

assert(result23.contains(8));