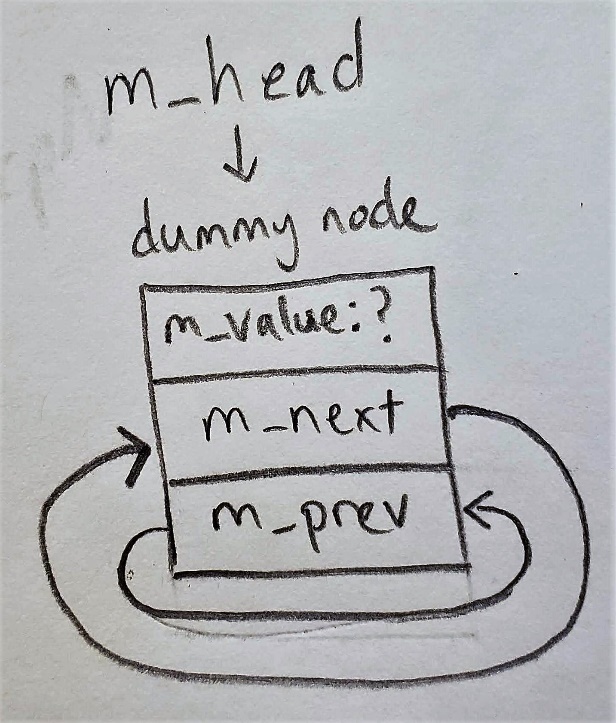
Oo, Thant Zin (Andy)

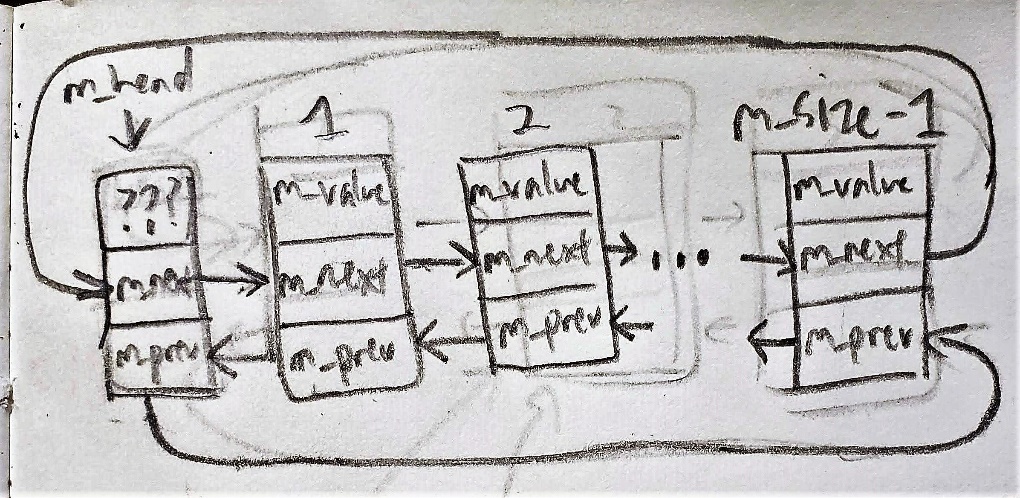
CS 32, Section 2D

# Project 2 Report

## 1. Doubly-linked list implementation

 My doubly-linked list implementation for Set used a circularly linked list with the head pointing to a dummy sentinel node. The dummy node’s m\_value is never assigned, as nowhere in my code is it accessed. If the list is not empty, the dummy node’s m\_next pointer points to the first interesting element in the linked list; m\_prev points to the last element. The reason I chose this design implementation was to avoid having to deal with nullptrs in special cases such as when the list is empty, or has one element.

The picture to the right represents an empty Set. M\_head points to the dummy node. Both the dummy node’s m\_next and m\_prev pointers point to itself.

The picture underneath the empty Set representation represents a Set with m\_size number of interesting elements. Notice how the last element’s (labelled m\_size) m\_next pointer circles back around and points to the dummy node. Similarly, the dummy node’s m\_prev pointer circles around and points to the last element. (Note: the numbers 1, 2 and m\_size are merely labels to show the positions of the first, second and last node respectively. They don’t actually exist in my implementation)

## 2. Pseudocode for non-trivial functions

**Member functions:**

bool Set::insert(const ItemType& value)

check if Set is not empty, if so:

check all elements currently in the Set to see if value already exists, if so:

return false

create new node to be inserted at the front of the Set

connect new node to the node previously at the front, such that the old front succeeds the new

connect the new node to the head, such that head precedes it

update the previous front node so that it connects backwards to the new node

update the head, so that it connects forwards to the new node

increment m\_size to reflect the new number of elements

return true

bool Set::erase(const ItemType& value)

create a traversal pointer to check if value already exists in the Set

the traversal pointer will either stop when it has traversed the entire Set or when it has reached a node containing the value.

if the pointer is pointing to m\_head (ie. Traversed the entire list):

return false

otherwise (meaning it has found a node containing value)

update the node which precedes the node to be deleted, such that it precedes the node after the node to be removed

update the node which succeeds the node to be deleted, such that it succeeds the node which precedes the node to be removed

delete the node containing value

decrement m\_size to reflect the new number of elements

return true

bool Set::get(int i, ItemType& value) const

check if i is within bounds (for a Set of n elements, i can only be the set of integers from 0 to n-1, per the spec), if so:

create a counter to keep track of the number of items greater than the value of the current node

repeatedly, for each element in the Set:

(re)set counter to 0

compare each element’s value with all other values in the Set

update the counter as needed (when current value > another value being compared)

after all values in Set are compared, check if counter == i, if so:

set value to the current node’s value

return true

return false

void Set::swap(Set& other)

create a temporary variable to swap this Set and the other Set’s values

swap values

create a temporary pointer to swap the memory addresses of this Set and the other Set’s m\_head pointers

swap pointers

Set::Set()

Initialize an empty set (set m\_size to 0)

Initialize a dummy node (per my implementation design)

Set::Set(const Set& other)

Copy other Set’s m\_size

Initialize a dummy node

Repeatedly, for the number of elements in other Set:

Create a copy of the other Set’s node

Copy other’s value into the copied node

Connect the copied node forwards to the previous first item

Connect the copied node backwards to dummy node (because node is new first item)

Connect the previous first item backwards to the copied node

Connect dummy node forwards to the copied node

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

check to make sure the same Set isn’t being assigned to itself, if so:

create temporary Set with the copy constructor (copying other)

swap this Set with temporary Set

return reference to Set (for multiple assignment ie. a = b = c)

Set::~Set()

Repeatedly, for the number of elements in Set:

Store pointer to current node in a temporary pointer

Access next node with traversal pointer

Delete current node via the temporary pointer

Traversal stops when traversal pointer points to the dummy node (ie. Set is empty)

Delete the dummy head

**Non-member functions:**

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

repeatedly, for the number of elements in result:

get the value in result > than the ith element (i ranging from 0 to n-1, given a Set with n elements), and store it in a temporary variable

check if neither set contains the value, if so erase it

check if erase was successful, if so:

decrement the number of iterations and the current iterator\*\*

loop through both sets and insert all values; insert should automatically check if value is present

\*\* Because if the item that is greater than i elements is erased, then the item greater than i + 1 elements is now the item greater than i elements. If i is not decremented then the new ith element will not be checked.

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

repeatedly, for the number of elements in result:

get the value in result > than the ith element (i ranging from 0 to n-1, given a Set with n elements), and store it in a temporary variable

check if s1 contains the value in result, if not erase it

check if erase was successful, if so:

decrement the number of iterations and the current iterator\*\*

repeatedly, for the number of elements in s1:

get the value in s1 > than the ith element (i ranging from 0 to n-1, given a Set with n elements), and store it in a temporary variable

check if s2 contains the value in result, if not erase it

check if erase was successful, if so:

decrement the number of iterations and the current iterator\*\*

if s2 doesn’t contain the value, insert it into the result Set

## 3. Test Cases

These tests were performed on a set of strings (i.e., ItemType was a type alias for std::string).

Set a; //testing default constructor

assert(a.size() == 0); //testing size function on empty set

assert(a.empty()); //testing empty function on empty set

assert(a.insert("a")); //testing insert function

assert(a.size() == 1); //testing size function on non empty set

assert(!a.empty()); //testing empty function on non empty set

assert(!a.erase("b")); //testing erase to ensure values that don’t match b don’t get erased

assert(!a.insert("a")); //testing insert not adding an already present value

assert(a.erase("a")); //testing erase to make sure present values are erased

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

assert(a.contains("a")); //testing contains function

assert(!a.contains("b")); //testing contains function for nonexistent value

a.insert("b");

assert(a.contains("b")); //making sure contains works for b that was added

Set b(a); //testing copy constructor

assert(b.contains("a") && b.contains("b") && b.size() == 2); //making sure a was copied into b

Set c;

Set d(c); //testing copy constructor for empty set

assert(!d.size()); //making sure d is empty

assert(d.empty());

string ab = "";

assert(a.get(1, ab)); //testing get function

assert(ab == "b"); //b is greater than one value

assert(a.get(0, ab) && ab == "a"); //a is smallest value

assert(!d.get(0, ab)); //testing get return value (should return false for empty set)

assert(!a.get(2,ab)); //testing get return value for out of bounds case (2 > m\_size – 1, should return false)

Set e;

swap(e, d); //testing swap with two empty sets

swap(a, d); //testing swap function with empty set and non empty set

assert(a.size() == 0 && d.size() == 2 && d.contains("a") && d.contains("b")); //making sure values were swapped

e = c = d; //testing assignment operator

assert(c.size() == 2 && c.contains("b")); //ensuring values are consistent between c and d

assert(e.size() == 2 && e.contains("b")); //ensuring values are consistent between e and c

a.insert("f");

a.insert("as");

unite(a, c, d); //testing unite function with two non empty sets

assert(d.size() == 4 && d.contains("as")); //ensuring result set is accurate

subtract(d, a, c); //testing subtract function with two non empty sets

assert(c.size() == 2 && c.contains("a") && c.contains("b")); //ensuring that the two values leftover are consistent with the requirements for subtract

Set f;

a = b = f; //check assignment operator with empty sets

assert(a.empty() && b.empty());//check to make sure they are empty

unite(a, b, f); //testing unite function with empty sets

assert(f.empty());

subtract(a, b, f); //testing subtract function with empty sets

assert(f.empty());

a.insert(“a”);

b.insert(“b”);

unite(a, b, a); //testing unite with aliasing, ie when result set == s1 or s2

assert(a.size() == 2 && a.contains(“b”); //making sure the result set was correctly modified

subtract(b, a, f); //testing subtract when set2 is bigger than set1 and has all of set1’s values

assert(f.empty());

a = f;

f.insert(“a”);

f.insert(“b”);

a.insert(“c”);

a.insert(“d”);

a.insert(“e”);

subtract(a, f, f); //Check if subtract function works if result and arguments are the same

assert(f.size() == 3 && f.contains(“c”)); //since a and f didn’t have any values in common, f

assert(f.contains(“d”) && f.contains(“e”)); //should contain everything a has