Project 2 Write Up:

1. My doubly-linked list is a standard one with a head pointer. I do not use a dummy node and it is not circular. When the list is empty, I have the head pointer simply as a null pointer, meaning there is nothing in the list. When there is only one element, the head points to the new node, and the one element previous and next are both null pointer. As I add elements I make sure that the first elements previous pointer is the null pointer as well as the last elements next pointer. Each element in the list has a value, pointer to the next element as well as the previous. When ever a new node is inserted, I make sure to put in ascending order so I do not have to sort it later.
2. Psedocode

* Subtract:
  + initialize temp set equal to s1
  + initialize temp2 set equal to s1
  + Repeatedly:
    - Get each value from temp
    - Repeatedly:
      * Get each value from s2
      * If value from temp is equal to value from s2
        + Erase value from temp1 // use erase function
  + Result is set equal to temp 1
* Unite:
  + Initialize a set temp equal to s2
  + Repeatedly:
    - Get each value from s1
    - Attempt to insert into temp // insert only inserts if it can
  + Set result equal to temp
* Insert
  + Repeatedly
    - If value of current node = value
      * Return false
  + If( head is null ptr or value < head value)
    - Make new node
    - New previous assigned null ptr
    - Reassign pointer values
    - increment number of items
    - return true
  + else
    - initialize node pointer to head;
    - repeatedly loop through linked list:
      * if(value is greater than current node value, less than next node and not at the last element)
        + make a new node
        + reassign pointer values
        + assign value
        + increment items
        + return true;
      * if value is greater than last node
        + make new node
        + assign pointer variables
        + assign value
        + increment items
        + return true
* erase
  + if head is null ptr
    - return false
  + if head value = value
    - make a node pointer to kill
    - head is assigned to kill me next
    - kill me previous set to null ptr
    - delete kill me
    - decrement items
    - return true
  + make nod pointer equal to head
  + repeteadly
    - find when the current node’s next value is the value passed in
    - make a node pointer to kill
    - adjust the previous and next pointers surrounding the node to kill
    - delete node
    - decrement items
    - return true;
* swap
  + make a temporary node pointer pointing to the other head
  + other head is assigned to head
  + head is assigned equal to temp
  + initialize a int equal to other.m\_items
  + other.M\_items is given m\_items value;
  + m\_items is given temp value;
* copy constructor
  + make a temporary set;
  + repeatedly:
    - get other value and each index
    - insert into a temporary set
  + m\_items is set equal to other items
  + call swap(temp);
* assignment operator
  + check to see if the same class is being passed in
  + (code identical to copy constructor)
  + Return \*this
* Destructor

Repeatedly:

* + - Delete each node of the linked list
    - Advance to the next node

Test Cases:

Set s1; //default constructor

Assert(s1.size ==0); //make sure there are no elements

Assert(s1.empty == true);

//insert items (not in order so we can check with get to see if they get organized);

Assert(s1.insert(“a”));

Assert(s1.insert(“c”));

Assert(s1.insert(“b”));

Assert(s1.insert(“f”));

Assert(s1.insert(“e”));

//size

Assert(s1.size()==4);

// insert fail

Assert(s1.insert(“a”)==false);

Assert(s1.insert(“b”)==false);

//get

ItemType value;

Assert(s1.get(0, value);

Assert(value== “a”);

Assert(s1.get(3, value);

Assert(value== “e”);

Set s1; //default constructor

Assert(s1.size ==0); //make sure there are no elements

Assert(s1.empty == true);

//insert items (not in order so we can check with get to see if they get organized);

Assert(s1.insert(“a”));

Assert(s1.insert(“c”));

Assert(s1.insert(“b”));

Assert(s1.insert(“f”));

Assert(s1.insert(“e”));

//size

Assert(s1.size()==5);

// insert fail

Assert(s1.insert(“a”)==false);

Assert(s1.insert(“b”)==false);

//get

ItemType value;

Assert(s1.get(0, value);

Assert(value== “a”);

Assert(s1.get(3, value);

Assert(value== “e”);

//erase and contains

Assert(s1.erase(“f”);

Assert(s1.contains(“f”) == false);

Assert(s1.contains(“a”));

Assert(s1.contains(“b”));

Assert(s1.contains(“c”));

//test unite and subtract and copy

Set s2;

Assert(s2.insert(“a”));

Assert(s2.insert(“b”));

Assert(s2.insert(“c”));

Assert(s2.insert(“d”));

Assert(s2.insert(“e”));

Set s3;

Assert(s3.insert(“a”));

Assert(s3.insert(“b”));

Assert(s3.insert(“f”));

Assert(s3.insert(“g”));

Assert(s3.insert(“h”));

Set s4 = s1; // give s4 values //copy constructor

Assert(s4.contains(“a”));

Assert(s4.contains(“b”));

Assert(s4.contains(“c”));

Assert(s4.size()==4); // make sure the size was copied as well

Set s5;

S5 =s1;

Assert(s4.contains(“a”));

Assert(s4.contains(“b”));

Assert(s4.contains(“c”));

Assert(s4.size()==4); // make sure the size was copied as well

//test unite

unite(s2,s3,s4); //unite

Assert(s4.contains(“a”));

Assert(s4.contains(“b”));

Assert(s4.contains(“c”));

Assert(s4.contains(“d”));

Assert(s4.contains(“e”));

Assert(s4.contains(“f”));

Assert(s4.contains(“g”));

Assert(s4.contains(“h”));

//test subtract

Subtract(s2,s3,s4); //subtract

Assert(s4.contains(“c”));

Assert(s4.contains(“d”));

Assert(s4.contains(“e”));