**Project 2 Report**

**Part 1. Description and Illustration**

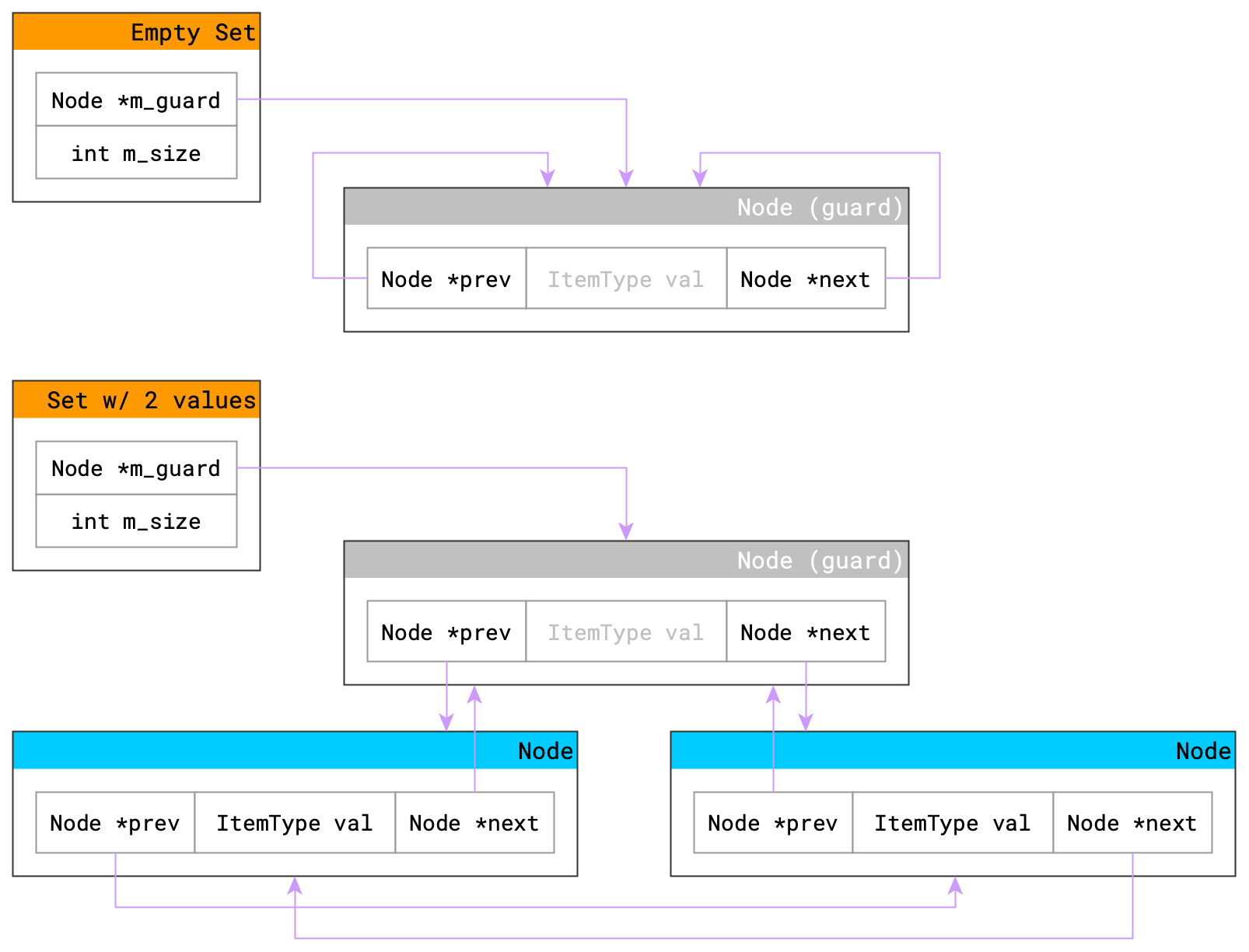
/\* This file has been carefully formatted. In case the formatting is messed up

\* by MS word, consider using this google doc. The content is the same.

<https://docs.google.com/document/d/12NsWZSQdyqiZ7XjzybncnKM8I3EkfLeJZvvDb5U9Y10/edit?usp=sharing>

\*\*/

My doubly-linked list is implemented with a **circular** structure and a dummy node which I call guard node. A sketch illustration of the design is shown below:



Note:

1. The guard node (dummy node)'s value is NEVER visited.

2. The order of the nodes always follows the order of the values in between member function calls.

**Part 2. Pseudocode**

**1) introducing some meta-pseudocode**

Before getting into pseudocode, I want to introduce some meta-pseudocode that would help the function pseudocode be easier to read and understand.

**Linked list iteration**: Several functions in this program involves iterating through the whole list. The process is as follows:

|  |
| --- |
| start from the first node while the current node is not the guard node  do something with this node  move on to the next node |

This iteration process is implemented using a simple for loop as shown below.

|  |
| --- |
| for (Node \*p = m\_guard->next; p != m\_guard; p = p->next) {  // Do something  } |

In the next section, I will simply say iterate through the linked list to represents this process.

**Node removal**: Removing a given node (not the guard node) has the following process:

|  |
| --- |
| link this node's next node's prev pointer to the prev node  link this node's prev node's next pointer to the next node  destruct this node  reduce the size of the list by 1 |

Notice that node removal also reduces the size of the list by 1. This ensures the consistency of the data structure. In the next section, I will simply say remove the node to represent this process.

|  |
| --- |
| node->prev->next = node->next;  node->next->prev = node->prev;  delete node;  m\_size -= 1; |

**Node insertion**: Inserting a given node has the following process:

|  |
| --- |
| link this node's next pointer to the node right after the insertion position  link this node's prev pointer to the node right before the insertion position  link this node's next node's prev pointer to this node  link this node's prev node's next pointer to this node  destruct this node  increase the size of the list by 1 |

Notice that node insertion also increases the size of the list by 1. Again, This ensures the consistency of the data structure. In the next section, I will simply say insert the node at <position> to represent this process.

Assuming the position is right after the pointer p.

|  |
| --- |
| node->prev = p;  node->next = p->next;  node->prev->next = node;  node->next->prev = node;  m\_size += 1; |

With the meta-pseudocode defined, we are ready to move on to the next section.

**2) helper function pseudocode**

To make the functions easier to implement, I wrote several private helper function for the set class. Since I am going to use the helper functions in interface functions, I will introduce the helper functions' pseudocode first.

blue lines means this is a meta-pseudocode defined in part 2 section 1.

**removeNode**: bool removeNode(Node \*node)

Remove the given node. If the given node is the guard node, return false; otherwise, remove and destruct the node, reduce the set's size by 1, and return true.

|  |
| --- |
| if the node is the guard node  return false  remove the node  return true |

**clear**: Set::clear()

Remove all nodes except the guard node in the set. Note that it does not remove the guard node; it only resets the set to the initial status.

|  |
| --- |
| start from the first node while the current node is not the guard node  save the next node's pointer  remove this node  move on to the next node using the saved pointer |

**getNodeWith**: Set::Node \*Set::getNodeWith(const ItemType &val) const

Return the node in the set with the given value. If a node with the given value exists, return the pointer to that node; otherwise, if no node with the value is found, return the guard node instead.

|  |
| --- |
| iterate through the linked list  if the node has the given value  return the node  return the guard node |

**insertNodeAfter**: void insertNodeAfter(const ItemType &val, Node \*pos)

Create a new node with the given value, insert the new node after the positioning node with proper linking, and increase the set's size by 1.

|  |
| --- |
| create new node with value `val`  insert the new node after the given node `pos` |

**insertNodeBefore**: void insertNodeBefore(const ItemType &val, Node \*pos)

Create a new node with the given value, insert the new node before the positioning node with proper linking, and increase the set's size by 1.

Note: the real implementation simply calls insertNodeAfter with adjusted position.

|  |
| --- |
| create new node with value `val`  insert the new node before the given node `pos` |

**getNodeWith**: Set::Node \*Set::getNodeWith(const ItemType &val) const

Return the node in the set with the given value. If a node with the given value exists, return the pointer to that node; otherwise, if no node with the value is found, return the guard node instead.

|  |
| --- |
| iterate through the linked list  if the current node has the given value  return this node  return m\_guard |

**3) function pseudocode**

This section introduces the pseudocode of the public interface functions.

blue lines means this is a meta-pseudocode defined in part 2 section 1.

purple lines means this uses a private function defined in part 2 section 2.

(otherwise) means an implicit else, which is not required because of return or break or continue, etc, but helps to complete the logic.

**Default constructor**: Set::Set()

The default constructor, must initialize all member variables.

|  |
| --- |
| assign m\_guard with a new node  set m\_size to be 0  link guard node's next pointer to itself  link guard node's prev pointer to itself |

**Copy constructor**: Set::Set(const Set &src)

On top of what the default constructor does, the copy constructor also deep-copies all the contents from the source set as well.

|  |
| --- |
| call default constructor  iterate through source set's linked list  insert a new node with value before this set's guard node (which is the  end of the list) [insertNodeBefore] |

**Destructor**: Set::~Set()

Destruct all the normal nodes as well as the guard node.

|  |
| --- |
| remove all the nodes [clear]  remove the guard node |

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

Copy all the nodes from the source set, and insert into this set.

|  |
| --- |
| if the source set is this set  no need to assign, return directly  (otherwise)  remove all nodes [clear]  iterate through source set's linked list  insert a new node with value before this set's guard node (which is the  end of the list) [insertNodeBefore] |

**insert**: bool Set::insert(const ItemType &val)

Insert value into the set if not already present. Return true if the value was actually inserted; otherwise, since the value must be already present, return false.

|  |
| --- |
| iterate through the set  if the current node has the given value  return false since the value is already present  (otherwise)  if the current node's value is greater than the given value  found the position, now break  // At this point, either the position is found, or we are at the end of the list  // which means the given value is larger than any value in the set. Either way..  insert the node before the position |

**erase**: bool Set::erase(const ItemType &val)

Remove the value from the set if present. Return true if the value was removed; otherwise, leave the set unchanged and return false.

|  |
| --- |
| find the node with the given value [getNodeWith]  If found  remove the node and return true  else  return false  [removeNode] |

**contains**: bool Set::contains(const ItemType &val) const

Remove the value from the set if present. Return true if the value was removed; otherwise, leave the set unchanged and return false.

|  |
| --- |
| find the node with the given value [getNodeWith]  return whether it's found (true if found; false otherwise) |

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

If 0 <= i < size(), copy into value the item in the set that is strictly greater than exactly i items in the set and return true. Otherwise, leave value unchanged and return false.

|  |
| --- |
| if the given index is not valid (0 <= i < size())  return false  starting from the first node, move forward i times  give the value of the current node to `val` receiver  return true |

**swap**: void Set::swap(Set &other)

Exchange the contents of this set with the other one.

|  |
| --- |
| swap the pointer to the guard node  (it follows that the linked list are also swapped)  swap the size |

**dump**: void Set::dump() const

Writes the items in the array to cerr.

|  |
| --- |
| iterate through the set  write the value of the current node to cerr |

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

Returns result set that contains one copy of each of the values that appear in s1 or s2 or both. This function does not assume result is empty. Hence it will reset the result at the beginning.

|  |
| --- |
| construct a set for the result (which is not the given result receiver)  for i in 0 to the size of set 1  get the item from set 1  try inserting the item to the result  for i in 0 to the size of set 2  get the item from set 2  try inserting the item to the result  write the set to result receiver |

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

Returns result set that contains one copy of each of the values that appear in s1 but not s2. This function does not assume result is empty. Hence it will reset the result at the beginning.

|  |
| --- |
| construct a set for the result (which is not the given result receiver)  for i in 0 to the size of set 1  get the item from set 1  try inserting the item to the result  for i in 0 to the size of set 2  get the item from set 2  try erasing the item to the result  write the set to result receiver |

**Part 3. Tests**

This tests a wide spectrum of possible error cases. The test\_log function explains which part of the functionality the program is currently testing.

|  |
| --- |
| /\*  \* testSet.cpp  \* Copyright (C) 2019 Jim Zenn <zenn@ucla.edu>  \*  \* Distributed under terms of the MIT license.  \*/  #include "Set.h"  #include <cassert>  #include <iostream>  #include <string>  using namespace std;  void test\_log(const string &msg, const char &sep = ' ') {  cerr << sep << sep << sep << " " << msg << " " << sep << sep << sep << endl;  }  int main() {  string tmp;  cerr << endl;  cerr << "This test assumes that ItemType is set to `std::string`." << endl;  test\_log("TEST 1: Constructor test", '=');  Set s1;  assert(s1.size() == 0);  assert(s1.empty());  test\_log("TEST 1: Passed!", '=');  cerr << endl;  test\_log("TEST 2: Insert Tests", '=');  test\_log("PART 1: duplicate insert test", '-');  Set s2;  assert(s2.insert("1"));  assert(s2.size() == 1);  test\_log("inserting duplicating string", ' ');  assert(! s2.insert("1"));  // The insertion was not successful, the size should not be altered.  assert(s2.size() == 1);  test\_log("passed!", ' ');  test\_log("PART 2: bulk insert test", '-');  Set s3;  // insert 250 items (essentially strings of number 1 to 250)  for (int i = 1; i <= 250; i += 1) {  if (i % 50 == 0)  test\_log("trying inserting " + to\_string(i) + " ...", ' ');  assert(s3.insert(to\_string(i)));  assert(s3.size() == i);  if (i % 50 == 0)  test\_log("checking contains " + to\_string(i) + " ...", ' ');  assert(s3.contains(to\_string(i)));  }  assert(s3.contains(to\_string(250)));  test\_log("passed!", ' ');  test\_log("TEST 2: Passed!", '=');  cerr << endl;  test\_log("TEST 3: Erase Tests", '=');  test\_log("PART 1: size consistancy test", '-');  Set s4;  assert(s4.insert("1")); // size() == 1  assert(! s4.insert("1")); // size() == 1  assert(s4.insert("2")); // size() == 2  assert(s4.insert("3")); // size() == 3  assert(s4.insert("4")); // size() == 4  assert(s4.erase("4")); // size() == 3  test\_log("checking if erase reduces size correctly...", ' ');  assert(s4.size() == 3);  assert(!s4.erase("4"));  // "4" is no longer part of s4, this shouldn't change size  assert(s4.size() == 3);  assert(s4.erase("1")); // size() == 2  assert(s4.size() == 2);  assert(!s4.erase("1"));  // "1" is no longer part of s4, this shouldn't change size  assert(s4.size() == 2);  test\_log("passed!", ' ');  test\_log("PART 2: underloading test", '-');  assert(s4.erase("2")); // size() == 1  assert(s4.erase("3")); // size() == 0  assert(s4.empty()); // the set is empty  assert(s4.size() == 0);  // double check if s4 still contains the items that should've been removed  assert(! s4.contains("1"));  assert(! s4.contains("2"));  // keep trying to erase when the set is already empty  assert(! s4.erase("1"));  assert(! s4.erase("2"));  assert(! s4.erase("3"));  assert(! s4.erase("4"));  // double check if s4 still contains the items that should've been removed  assert(! s4.contains("3"));  assert(! s4.contains("4"));  // make sure the size of the set is still 0, but not something negative  assert(s4.empty());  assert(s4.size() == 0);  test\_log("passed!", ' ');  test\_log("TEST 3: Passed!", '=');  cerr << endl;  test\_log("TEST 4: Get Test", '=');  test\_log("PART 1: false alteration test", '-');  Set s5;  tmp = "don't touch me";  assert(! s5.get(0, tmp));  assert(tmp.compare("don't touch me") == 0);  test\_log("passed!", ' ');  test\_log("PART 2: ordering test", '-');  assert(s5.insert("3")); // 3  assert(s5.insert("8")); // 3 -> 8  assert(s5.insert("2")); // 2 -> 3 -> 8  assert(! s5.insert("3")); // 2 -> 3 -> 8  assert(s5.insert("4")); // 2 -> 3 -> 4 -> 8  assert(s5.size() == 4);  assert(s5.get(0, tmp));  assert(tmp.compare("2") == 0);  assert(s5.get(1, tmp));  assert(tmp.compare("3") == 0);  assert(s5.get(2, tmp));  assert(tmp.compare("4") == 0);  assert(s5.get(3, tmp)); // nothing is at index 4.  assert(tmp.compare("8") == 0);  assert(! s5.get(4, tmp)); // nothing is at index 4.  assert(tmp.compare("8") == 0);  assert(s5.contains("2"));  assert(s5.contains("3"));  assert(s5.contains("4"));  assert(s5.contains("8"));  test\_log("passed!", ' ');  test\_log("PART 2: actual copy test", '-');  // This part is intended to test if the set insert a copy of the value, rather  // than the original value. This could cause problem if the original value is  // altered after insertion. For example, the ordering of the set may be  // broken.  Set s6;  string \*str\_do = new string;  \*str\_do = "do";  string \*str\_love = new string;  \*str\_love = "love";  string \*str\_me = new string;  \*str\_me = "me";  s6.insert(\*str\_do); // "do"  s6.insert(\*str\_love); // "do" -> "love"  s6.insert(\*str\_me); // "do" -> "love" -> "me"  // Now we will try to alter the content of the set from the outside.  \*str\_do = "love";  \*str\_love = "me";  \*str\_me = "do";  // (Elvis Presley liked this test.)  // Now we check if s6 is altered.  string recv;  assert(s6.get(0, recv));  assert(recv.compare("do") == 0);  assert(s6.get(1, recv));  assert(recv.compare("love") == 0);  assert(s6.get(2, recv));  assert(recv.compare("me") == 0);  test\_log("passed!", ' ');  delete str\_do;  delete str\_love;  delete str\_me;  test\_log("TEST 4: Passed!", '=');  cerr << endl;  test\_log("TEST 5: Swap test", '=');  // Reusing s5 and s4 from TEST 3  // s4 is empty, s5 is 2 -> 3 -> 4 -> 8  s5.swap(s4);  assert(s5.size() == 0);  tmp = "nope";  assert(! s5.get(0, tmp));  assert(tmp == "nope");  assert(s4.size() == 4);  assert(s4.contains("2"));  assert(s4.contains("3"));  assert(s4.contains("4"));  assert(s4.contains("8"));  assert(s4.get(0, tmp));  assert(tmp == "2");  assert(s4.get(1, tmp));  assert(tmp == "3");  assert(s4.get(2, tmp));  assert(tmp == "4");  assert(s4.get(3, tmp));  assert(tmp == "8");  assert(! s4.get(4, tmp));  assert(tmp == "8");  test\_log("TEST 5: Passed!", '=');  cerr << endl;  test\_log("TEST 6: Assignment Operator", '=');  Set assigner, assignee;  test\_log("PART 1: memory exchange test", '-');  assigner.insert("1");  assignee = assigner;  // both assignee and assigner should contain "1" now  assert(! assignee.insert("1")); // duplicate "1", can't insert  assert(assigner.erase("1"));  // destruct the original "1", the new "1" shouldn't be affected  assert(! assignee.insert("1")); // duplicate "1", still can't insert  // assignee should still contain that "1"  assert(assignee.contains("1"));  assert(assignee.insert("2"));  assert(assignee.insert("3"));  assert(assignee.insert("4"));  assert(assignee.insert("5"));  assert(assignee.insert("6"));  assert(assignee.insert("7"));  assert(assignee.insert("8"));  assert(assignee.insert("9"));  assert(assigner.insert("1"));  test\_log("passed!", ' ');  test\_log("PART 2: deep copy test", '-');  tmp = "no!";  assert(assignee.get(0, tmp));  assert(tmp.compare("1") == 0);  assert(assignee.get(8, tmp));  assert(tmp.compare("9") == 0);  // should be checking an invalid index  assert(! assignee.get(9, tmp));  // tmp's value shouldn't change  assert(tmp.compare("9") == 0);  // make sure assigner and assignee are not sharing memory  assert(! assigner.contains("3"));  assert(assignee.contains("1"));  assert(assignee.erase("1"));  assert(assigner.contains("1"));  test\_log("passed!", ' ');  test\_log("TEST 6: Passed!", '=');  cerr << endl;  test\_log("TEST 7 Copy constructor", '=');  Set toclone;  toclone.insert("10"); // toclone: 10  toclone.insert("11"); // toclone: 10 -> 11  Set cloned = toclone; // cloned: 10 -> 11  // make sure the copy process is a deep copy  assert(cloned.erase("10")); // cloned: 11  assert(toclone.get(0, tmp)); // toclone[0] should still be 10  assert(tmp.compare("10") == 0);  assert(toclone.size() == 2);  assert(cloned.size() == 1);  assert(cloned.contains("11"));  assert(! cloned.contains("10"));  test\_log("TEST 7: Passed!", '=');  cerr << endl;  test\_log("TEST 8 unite and subtract", '=');  test\_log("PART 1: self-assigning unite test", '-');  // Make sure when the result is one of the operand set, unite still works.  Set implosion1, implosion2;  implosion1.insert("i");  implosion1.insert("m");  implosion1.insert("p");  implosion1.insert("l");  implosion1.insert("o");  implosion1.insert("s");  implosion2.insert("i");  implosion2.insert("o");  implosion2.insert("n");  unite(implosion1, implosion2, implosion1);  string tassu;  assert(implosion1.get(0, tassu));  assert(tassu.compare("i") == 0);  assert(implosion1.get(1, tassu));  assert(tassu.compare("l") == 0);  assert(implosion1.get(2, tassu));  assert(tassu.compare("m") == 0);  assert(implosion1.get(3, tassu));  assert(tassu.compare("n") == 0);  assert(implosion1.get(4, tassu));  assert(tassu.compare("o") == 0);  assert(implosion1.get(5, tassu));  assert(tassu.compare("p") == 0);  assert(implosion1.get(6, tassu));  assert(tassu.compare("s") == 0);  assert(! implosion1.get(7, tassu));  assert(tassu.compare("s") == 0);  test\_log("passed!", ' ');  test\_log("PART 2: self-assigning subtract test", '-');  // Make sure when the result is one of the operand set, subtract still works.  Set explosion1, explosion2;  explosion1.insert("e");  explosion1.insert("x");  explosion1.insert("p");  explosion1.insert("l");  explosion1.insert("o");  explosion1.insert("s");  explosion2.insert("i");  explosion2.insert("o");  explosion2.insert("n");  subtract(explosion1, explosion2, explosion2);  string tasss;  assert(explosion2.get(0, tasss));  assert(tasss.compare("e") == 0);  assert(explosion2.get(1, tasss));  assert(tasss.compare("l") == 0);  assert(explosion2.get(2, tasss));  assert(tasss.compare("p") == 0);  assert(explosion2.get(3, tasss));  assert(tasss.compare("s") == 0);  assert(explosion2.get(4, tasss));  assert(tasss.compare("x") == 0);  assert(! explosion2.get(5, tasss));  assert(tasss.compare("x") == 0);  test\_log("passed!", ' ');  test\_log("PART 3: unite functionality test", '-');  // Make sure unite works as expected  Set su1;  su1.insert("12");  Set su2 = su1;  su2.insert("11");  su2.insert("13");  su2.insert("12");  su2.insert("15");  su1.insert("10");  su1.insert("14");  su1.insert("16");  Set united;  united.insert("no");  united.insert("not");  united.insert("me");  united.insert("10");  united.insert("11");  united.insert("12");  unite(su1, su2, united);  test\_log("Checking if su1 is changed after unite (it shouldn't) ...", ' ');  assert(su1.size() == 4);  string tu = "Hmmm";  assert(su1.get(0, tu));  assert(tu.compare("10") == 0);  assert(su1.get(1, tu));  assert(tu.compare("12") == 0);  assert(su1.get(2, tu));  assert(tu.compare("14") == 0);  assert(su1.get(3, tu));  assert(tu.compare("16") == 0);  test\_log("Checking if su2 is changed after unite (it shouldn't) ...", ' ');  assert(su2.size() == 4);  assert(su2.get(0, tu));  assert(tu.compare("11") == 0);  assert(su2.get(1, tu));  assert(tu.compare("12") == 0);  assert(su2.get(2, tu));  assert(tu.compare("13") == 0);  assert(su2.get(3, tu));  assert(tu.compare("15") == 0);  test\_log("Checking if the result is correct", ' ');  assert(united.size() == 7);  assert(united.get(0, tu));  assert(tu == "10");  assert(united.get(1, tu));  assert(tu == "11");  assert(united.get(2, tu));  assert(tu == "12");  assert(united.get(3, tu));  assert(tu == "13");  assert(united.get(4, tu));  assert(tu == "14");  assert(united.get(5, tu));  assert(tu == "15");  assert(united.get(6, tu));  assert(tu == "16");  assert(! united.get(7, tu));  assert(tu == "16");  test\_log("passed!", ' ');  test\_log("PART 4: subtract functionality test", '-');  // Make sure subtract works as expected  Set ss1;  ss1.insert("12");  Set ss2 = ss1;  ss2.insert("11");  ss2.insert("13");  ss2.insert("12");  ss2.insert("15");  ss1.insert("10");  ss1.insert("14");  ss1.insert("16");  Set subtracted;  subtracted.insert("no");  subtracted.insert("not");  subtracted.insert("me");  subtracted.insert("10");  subtracted.insert("11");  subtracted.insert("12");  subtract(ss1, ss2, subtracted);  test\_log("Checking if ss1 is changed after subtract (it shouldn't) ...", ' ');  assert(ss1.size() == 4);  string ts = "Hmmm";  assert(ss1.get(0, ts));  assert(ts.compare("10") == 0);  assert(ss1.get(1, ts));  assert(ts.compare("12") == 0);  assert(ss1.get(2, ts));  assert(ts.compare("14") == 0);  assert(ss1.get(3, ts));  assert(ts.compare("16") == 0);  test\_log("Checking if ss2 is changed after subtract (it shouldn't) ...", ' ');  assert(ss2.size() == 4);  assert(ss2.get(0, ts));  assert(ts.compare("11") == 0);  assert(ss2.get(1, ts));  assert(ts.compare("12") == 0);  assert(ss2.get(2, ts));  assert(ts.compare("13") == 0);  assert(ss2.get(3, ts));  assert(ts.compare("15") == 0);  test\_log("Checking if the result is correct", ' ');  assert(subtracted.size() == 3);  assert(subtracted.get(0, ts));  assert(ts == "10");  assert(subtracted.get(1, ts));  assert(ts == "14");  assert(subtracted.get(2, ts));  assert(ts == "16");  assert(! subtracted.get(3, ts));  assert(ts == "16");  test\_log("passed!", ' ');  test\_log("TEST 8: Passed!", '=');  cerr << endl;  test\_log("TEST 9: Dump Test");  united.dump();  subtracted.dump();  test\_log("All tests passed!", '\*');  cerr << endl;  } |