Description:

The doubly-linked list I implemented was not circularly linked, and thus had both a head and a tail. It also did not have a dummy node in the beginning, so it accounted for cases where the list was empty and where the manipulation of the list required changes to either the head or tail pointer (i.e. one item in the list or inserted item appears at the top of the list). The list nodes contained the value they held, the address of the next node, and the address of the previous node. I used the insert function to order the list from least to greatest, alphabetical or numerical order depending on the ItemType.

Pseudocode:

* Constructor
  + Sets size of list to 0
  + Initializes head and tail pointer to nullptr
* Insert
  + If list is empty
    - Create new node with value
    - Connect the appropriate nodes (nullptr in this case)
    - Set head and tail to node
    - Increment list size
  + If value belongs at the front of the list
    - Create new node with value
    - Connect the appropriate nodes
    - Set head to node
    - Increment list size
  + Traverse the list
    - Exit if item is in the list
    - Break if next item is greater than value
      * Create new node with value
      * Connect the appropriate nodes
      * Increment list size
  + // Below executes if item belongs at end of list
  + Create new node with value
  + Connect the appropriate nodes
  + Set tail to node
  + Increment list size
* Erase
  + Check if this set contains the value to be erased
    - If it doesn’t, exit and return that this function failed
  + Find the node that contains the value
  + Point the previous node to the next node and next to the previous
  + Delete the node containing the value
* Get
  + Exit if first parameter is not valid
  + Traverse the sorted list until the appropriate node
  + Copy the item into value
* Swap
  + Swap the size of list of this with other
  + Swap the head of this with other
  + Swap the tail of this with other
* Copy Constructor
  + Set the size of list to other’s size of list
  + Traverse through each node in other
    - Make a new node in this set
    - Set its value to the value of the respective node in other
    - Connect this node to the previous and next node
* Assignment Operator
  + If object isn’t being set to itself
    - Copy the right hand side to a temp variable
    - Swap the left hand side with that temp variable
  + Return itself
* Destructor
  + If there are no items in the list
    - Exit
  + Else
    - Traverse through each node
      * Delete this node
      * Go to next node
* Unite
  + Set result to s1
  + Create temporary variable to store get values
  + For loop iterating through s2
    - Copy value to temporary variable using get
    - Insert temporary variable to result (insert function does not insert duplicate values)
* Difference
  + If s1 and s2 are equal
    - Create empty set
    - Set result to empty set
    - Return to exit function
  + Set result to s1
  + Create temporary variable to store get values
  + For loop iterating through s2
    - Copy value to temporary variable using get
    - If result contains variable
      * Erase variable from result
    - Else
      * Insert variable to result

Test Cases:

Set ss;

assert(ss.size() == 0); // test size on empty set

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

assert(!ss.erase("roti")); // test a failure to remove on an empty set

assert(ss.insert("roti")); // test insertion into empty set

assert(ss.size() == 1); // test if size is 1 after inserting 1 item

assert(!ss.empty()); // test if empty works on a set of size 1

assert(ss.erase("roti")); // test if erase works with only 1 element

assert(ss.size() == 0); // test size on empty set

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

assert(!ss.erase("roti")); // test a failure to remove on an empty set

assert(ss.insert("roti")); // test reinserting 1 item

assert(ss.insert("pita")); // test adding in second item

assert(ss.size() == 2); // test that size is working correctly with 2 items

assert(ss.contains("pita")); // make sure that contains works properly

ItemType x = "laobing";

assert(ss.get(0, x) && x == "pita"); // make sure get works when getting object 0

assert(ss.get(1, x) && x == "roti"); // make sure get works when getting object 1

assert(!ss.insert("roti")); // make sure it doesn’t allow repeats

assert(ss.insert("naan")); // add third element to set

Set ss2(ss);

assert(ss2.get(0, x) && x == "naan"); // make sure get works when getting object 0

assert(ss2.get(1, x) && x == "pita"); // make sure get works when getting object 1

assert(ss2.get(2, x) && x == "roti"); // make sure get works when getting object 2

assert(ss2.insert("bread")); // make sure insertion works

assert(ss2.erase("naan")); // check if removing works

assert(!ss.contains("bread")); // make sure ss2 and ss are not connected

assert(ss.contains("naan")); // make sure ss2 and ss are not connected

Set ss3;

ss3 = ss; // make sure operator= override works

assert(ss3.get(0, x) && x == "naan"); // make sure get works when getting object 0

assert(ss3.get(1, x) && x == "pita"); // make sure get works when getting object 1

assert(ss3.get(2, x) && x == "roti"); // make sure get works when getting object 2

assert(ss3.erase("naan")); // check if removing works

assert(ss.contains("naan")); // make sure ss3 and ss are not connected to the same linked list

unite(ss, ss2, ss); // check if unite works, including if ss1 unites into ss1

// checking if unite works correctly

assert(ss.get(0, x) && x == "naan"); // make sure get works when getting object 0

assert(ss.get(1, x) && x == "pita"); // make sure get works when getting object 1

assert(ss.get(2, x) && x == "roti"); // make sure get works when getting object 2

assert(ss.get(3, x) && x == "bread"); // make sure get works when getting object 2

difference(ss, ss2, ss); //check if difference works

// checking if difference works correctly

assert(ss.get(0, x) && x == "naan"); // make sure get works when getting object 0

assert(ss.get(1, x) && x == "bread"); // make sure get works when getting object 2

ss.swap(ss2); // check if swap works properly

assert(ss.get(0, x) && x == "pita"); // checking that ss now has what ss2 had

assert(ss.get(1, x) && x == "roti"); // checking that ss now has what ss2 had

assert(ss2.get(0, x) && x == "naan"); // checking that ss2 now has what ss had

assert(ss2.get(1, x) && x == "bread"); // checking that ss2 now has what ss had