Linked List Implementation:

My doubly-linked list is constructed in a through a chained group of nodes. I store a head pointer, which is a pointer to the first node in the linked list. By following the head pointer, I get to the first node object. Each node object has a value, a previous node pointer, and a next node pointer. This means that if I am at a particular node, I know where in memory I can find the next and previous node. From this, the only pointer I must store in my linked list is the head pointer. The head node has nullptr as its previous, since there is nothing before, and the last node has nullptr as its next since it is the last one. I chose not to save a tail pointer, which would point to the last node in the list, since I am only traversing the list from top to bottom, so I don’t need the extra functionality of traversing bottom to top. The nodes in my list store values of type ItemType (which was set to string for this problem). For simplicity, I chose to add each new element to the front of the list so that I had to rearrange the fewest pointers, so the order of items in the list reflects the order of their addition to the list.

nullptr

Head pointer

Node 2

Node 1

Node 3

nullptr

Pseudocode:

~Set()

1. Start from the head pointer and iterate through all nodes
2. Use delete to delete dynamically allocated nodes

Set(const Set &src)

1. Start from src’s head pointer and iterate through all the nodes
2. Use the insert function to insert the value of each node in src to this set

Set &operator=(const Set &src)

1. Start from the head pointer and iterate through all nodes
2. Use delete to delete dynamically allocated nodes
3. Start from src’s head pointer and iterate through all the nodes
4. Use the insert function to insert the value of each node in src to this set

int size() cons

1. Start from the head pointer and iterate through all nodes
2. Count the number of nodes

bool insert(const ItemType& value)

1. If the list already has the value, return false
2. Create a new dynamically allocated node
3. Set the current head node’s previous pointer to the new node
4. Set the head pointer to the new node

bool erase(const ItemType& value)

1. Start from the head pointer and iterate through all nodes
2. If there is only 1 item in the list
   1. Delete the item
   2. Set head to the nullptr
3. If trying to delete the top value
   1. Set next’s previous value to nullptr
   2. Delete the item
4. If trying to delete middle value
   1. Set next’s previous to the item’s previous
   2. Set previous’s next to the item’s next
   3. Delete the item
5. If trying to delete last value
   1. Set previous’s next to nullptr
   2. Delete the item

bool contains(const ItemType& value) const

1. Start from the head pointer and iterate through all nodes
2. Return true if the value of a node in the list matches the value passed in

bool get(int pos, ItemType& value) const

1. Start from the head pointer and iterate through all nodes
2. Create a nested loop by creating another pointer to iterate through all the nodes
3. Count the number of values a particular node is greater than
4. Change value to be the node that is greater than pos nodes

void swap(Set& other)

1. Swap the two set’s head pointers

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

1. If s1, s2, result all same
   1. Do nothing
2. If s1 and result are same
   1. Start from the head pointer of s2 and iterate through all nodes
   2. Add all nodes from s2
3. If s2 and result are same
   1. Start from the head pointer of s1 and iterate through all nodes
   2. Add all nodes from s1
4. Else
   1. Start from the head pointer of result and iterate through all nodes
   2. Delete all nodes from result
   3. Start from the head pointer of s1 and iterate through all nodes
   4. Add all nodes from s1
   5. Start from the head pointer of s2 and iterate through all nodes
   6. Add all nodes from s2 that have not already been added from s1

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

1. If s1 and s2 are same
   1. Start from the head pointer of result and iterate through all nodes
   2. Delete all nodes from result
2. If s1 and result are same
   1. Start from the head pointer of result and iterate through all nodes
   2. Delete all nodes from result that exist is s2
3. If s2 and result are same
   1. Start from the head pointer of s1 and iterate through all nodes
   2. Delete all nodes from result that exist in s2
   3. Start from the head pointer of result and iterate through all nodes
   4. Delete all nodes from result that don’t exist in s1
4. Else
   1. Start from the head pointer of result and iterate through all nodes
   2. Delete all nodes from result
   3. Start from the head pointer of s1 and iterate through all nodes
   4. Insert all nodes from s1 to result if they are not in s2

Test cases:

Set s;

// testing empty function

assert(s.empty());

// testing inserting

s.insert("abc");

s.insert("arepa");

s.insert("dsf");

// try to insert something in list that is already in list

assert(!s.insert("dsf"));

// testing size function

assert(s.size() == 3);

// testing get function with various values for pos

string value;

s.get(0, value);

assert(value == "abc");

s.get(1, value);

assert(value == "arepa");

s.get(2, value);

assert(value == "dsf");

// testing contains function

assert(!s.contains("123"));

assert(s.contains("arepa"));

// testing erase function

s.erase("dsf");

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

assert(!s.erase("i am not in the set"));

// testing copy constructor

Set b(s);

assert(!b.empty());

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

// testing assignment operator

Set c;

c = b;

assert(!c.empty());

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

// testing swap function

Set d;

d.insert("I was originally in d");

d.swap(c);

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

assert(d.contains("arepa"));

assert(c.contains("I was originally in d"));

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

Set s1;

s1.insert("2");

s1.insert("8");

s1.insert("3");

s1.insert("9");

s1.insert("5");

Set s2;

s2.insert("6");

s2.insert("3");

s2.insert("8");

s2.insert("5");

s2.insert("10");

Set result;

// testing unite function

unite(s1, s2, result);

assert(result.contains("9"));

assert(result.contains("3"));

assert(result.contains("6"));

assert(result.contains("5"));

assert(result.contains("10"));

assert(result.contains("2"));

assert(result.contains("8"));

assert(result.size() == 7);

// testing special cases of unite

Set s3(s1);

unite(s3, s2, s3);

assert(s3.size() == 7);

Set s4(s2);

unite(s1, s4, s4);

assert(s4.size() == 7);

Set s5(s1);

unite(s1, s1, s1);

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

// testing special cases of subtract

subtract(s1, s2, result);

assert(result.contains("9"));

assert(result.contains("2"));

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

subtract(s1, s1, result);

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

Set s6(s1);

subtract(s6, s2, s6);

assert(s6.size() == 2);

Set s7(s2);

subtract(s1, s7, s7);

assert(s7.size() == 2);