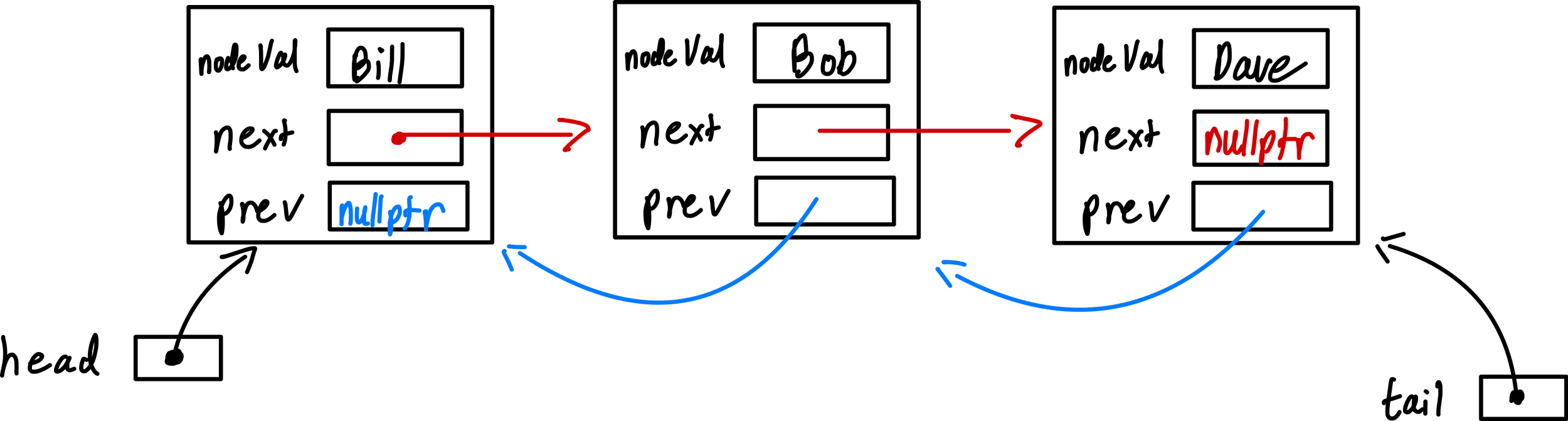
### Description of Design

My Set is a doubly-linked list that is not circular and does not have a dummy node. 

In my list’s nodes there are 3 data types:

*nodeVal* - holds a value of ItemType specified in header file “Set.h”

*Next* - pointer that holds the address to the next node in the list

*prev* - pointer that holds the address to the previous node in the list

In every object Set there is a “head” pointer that points to the first node in the linked list, there is a “tail” pointer that points to the last node in the linked list, and a “m\_size” that holds the number of nodes in the linked list. When adding new nodes to the list, the values of the new node are defined first, and then the list is adjusted to the new node insertion.

### Pseudocode

**Insert:**

if list does not contain value

if list is empty, add to empty

if want to add item to the front of the list, but not empty, add to front

if item is greater than all items in list, add to end

if item is to be inserted somewhere in the middle of Set, add to set

else , do nothing

**Erase:**

if value we delete is first in list, delete

otherwise, loop through list and stop right above desired node to delete(if found)

adjust pointers of previous and next nodes and delete desired node

**Get:**

if valid position

loop through nodes until temporary pointer points to desired value

Copy desired value into value parameter

**Swap**:

Use temp variables to swap head and tail pointers of 2 sets, and size values of 2 sets

**Unite:**

erase all elements in result set

insert all elements of set 1 into result set

insert elements of set 2 that ARE NOT contained in set 1

**butNo**t:

Create temporary set

loop through s1 and get a value

if s2 does not contain that value, insert into temporary Set

return Set

### Test Cases

***These cases were performed on a set of strings (ItemType was a type alias for std::string at compilation time) The highlighted comments are to explain the reasoning behind my tests.***

Set a;

Set b; // Check copy constructor

Set c;

c = b; // Check assignment operator

assert(a.size() == 0); // Check that initial size is zero

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

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

assert(a.empty() && b.empty() && c.empty()); // Check that EMPTY works

a.insert("1"); a.insert("2"); a.insert("3"); a.insert("3");

assert(a.size() == 3); // Check that insert only works if value is not already present

b.insert("1"); b.insert("2"); b.insert("3"); b.insert("4");

assert(b.size() == 4);

c.insert("rag");

Set d(c); // Check the copy constructor with only one node

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

Set e(b); // Check that the copy contructor works with multiple nodes

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

assert(c.size() == 1); // Check that INSERT increases the size

assert(!c.empty());

unite(a, b, c);

assert(c.contains("1") && c.contains("2") && c.contains("3") && c.contains("4"));

// Check that UNITE works properly

unite(a, a, c);

assert(c.contains("1") && c.contains("2") && c.contains("3") && c.size() == 3);

// Check aliasing case

unite(a, b, a);

assert(c.contains("1") && c.contains("2") && c.contains("3") && c.size() == 3);

// Check alternate aliasing case

unite(b, a, b);

assert(b.contains("1") && b.contains("2") && b.contains("3") && b.contains("4"));

unite(a, a, a);

assert(c.contains("1") && c.contains("2") && c.contains("3") && c.size() == 3);

// Check last aliasing case

assert(!a.erase("5")); // Check that erasing something that is not present returns false

assert(a.erase("1")); // Check that erasing something that is present returns true

assert(a.erase("2"));

assert(a.erase("3"));

assert(c.erase("1")); // Check that erasing something that is present returns true

assert(c.erase("2"));

assert(c.erase("3"));

b.erase("1");

b.erase("2");

b.erase("3"); //setting b back to empty for further testing

b.erase("4");

assert(a.size() == 0); // Check that erase actually decreased size

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

a.insert("Wow");

a.insert("So"); //insert 3 elements to a

a.insert("Cool");

b.insert("Swag");

a.swap(b); // Check that swap function cancels swap function out (by spec)

b.swap(a);

assert(a.contains("Wow") && a.contains("So") && a.contains("Cool"));

// Check that contain functions properly

assert(b.contains("Swag"));

assert(!c.contains("Swag"));

Set g;

Set h;

Set i;

g.insert("1"); g.insert("2"); g.insert("3"); g.insert("4");

//check butNot function if result is empty

h.insert("1"); h.insert("2"); h.insert("3");

butNot(g, h, i);

assert(i.contains("4") && i.size() == 1);

Set p;

p.insert("5"); p.insert("1"); p.insert("3");

//checks butNot function if result Set has values inside it prev

butNot(g, h, p);

assert(p.contains("4") && p.size() == 1);

butNot(g, h, g);

assert(g.contains("4") && g.size() == 1);

//checks butNot for aliasing if s1 = result

butNot(h, h, h);

assert(h.size() == 0); //check if butNot asks for same sets, then the result is an empty set

ItemType item;

a.get(0, item);

assert(item == "Cool");

// Check that get functions for the first element (note: maybe use numbered strings next time)

a.get(1, item);

assert(item == "So");

//check get is accessing sorted values (sorted alphabetically with <> operators)

a.get(2, item);

assert(item == "Wow");

a.swap(b); //Check that swap function swaps b into a correctly

assert(a.size() == 1 && a.contains("Swag") && b.size() == 3 && b.contains("Wow") && b.contains("So") && b.contains("Cool"));

b.get(0, item);

assert(item == "Cool"); // Check that swap worked for b

b.get(1, item);

assert(item == "So");

b.get(2, item);

assert(item == "Wow");

Set empty;

Set nempty(a);

assert(empty.empty()); //check if sets are empty/copy constructor is at least copying something

assert(!nempty.empty());

empty.swap(nempty); // Check swap if the set is empty

assert(!empty.empty());

assert(nempty.empty());

cout << "Passed enough tests for me to turn it in. PLEASE WORK" << endl << endl;

//YAY