/////////////////////////////////////

// DESCRIPTION OF DOUBLY-LINKED LIST

/////////////////////////////////////

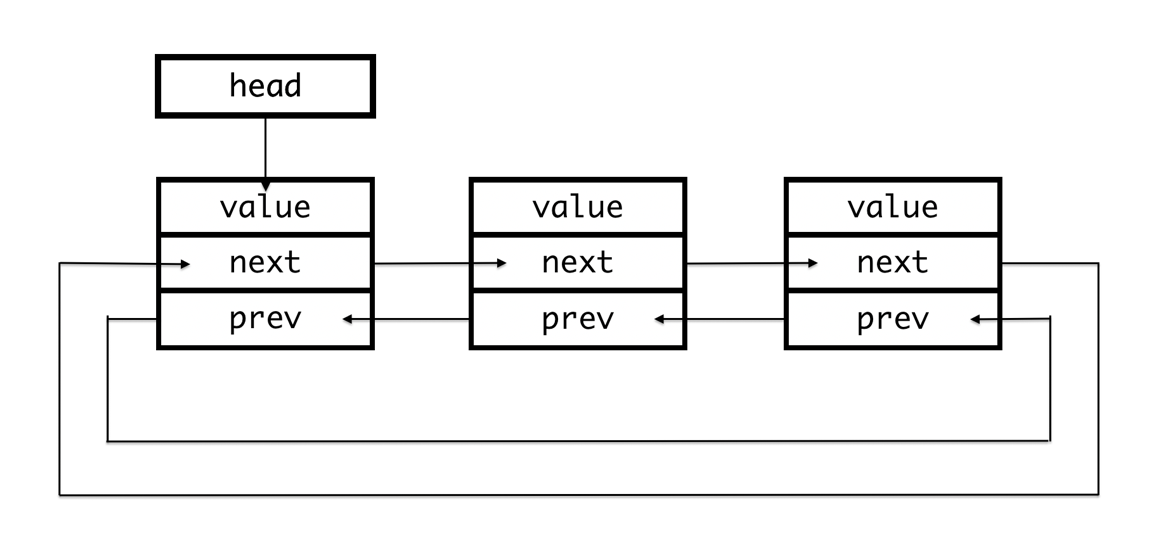
I used a circular linked list with a dummy node. My head pointer points to the dummy node (which contains a garbage value that will never be accessed), and whenever I loop through the list, I can always find the first node at head->next. I know the list is finished when the pointer equals head (i.e. the pointer points to the dummy node). Within the list, each node holds three items: a "value" member variable that holds a value of ItemType, a "next" pointer that points to the next node in the list, and a "prev" pointer that points to the previous node in the list. If the list is empty, the next and prev pointers will point to the same node (the dummy node). The values of the nodes, meanwhile, are sorted from smallest value to greatest value.

/////////////////////////////////////

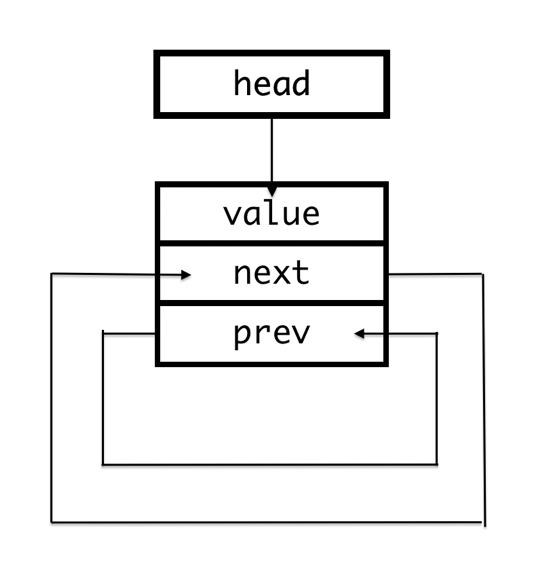
// ILLUSTRATION OF DOUBLY-LINKED LIST

/////////////////////////////////////

multiple items



one item



//////////////

// PSEUDOCODE

//////////////

---------insert()---------

// set a temporary pointer ptr to head->next

// increment ptr in order to loop through list; at the end, ptr points to node that should go after inserted node

// if a duplicate is found, return false

// if value of a node > parameter value, break

// store value of the previous node in pointer variable before

// create a new node newNode with value set as parameter value

// link up newNode's pointers

// set newNode's next pointer to ptr

// set newNode's prev pointer to before

// adjust prev and next pointers for previous and following nodes

---------erase()---------

// set a temporary pointer p to head->next

// traverse list until either end is reached or node value is found

// if p equals head, return false since value is not present

// create temporary pointers to hold value of previous/following nodes

// set previous node's next pointer to following node

// set following node's prev pointer to previous node

// free memory and return

---------unite()---------

// create a temporary new set

// add values from s1 into new set

// add values from s2 if they do not already exist

// swap the new set with result and return the result

---------subtract()----------

// create a new set to avoid aliasing issues

// loop through each value in s1

// if the value does not appear in s2

// add to new set

// swap the new set with result

//////////////

// TEST CASES

//////////////

/\* Testing initial set conditions \*/

Set set;

assert(set.size() == 0 && set.empty());

/\* Testing insert() function. \*/

set.insert("first");

assert(set.size() == 1); // empty list insertion

set.insert("second");

assert(set.size() == 2); // one element in list insertion

set.insert("third");

set.insert("abracadabra");

set.insert("third"); // error: duplicate.

set.insert("abracadabraa");

assert(set.size() == 5); // regular case

/\* Testing contains() function. \*/

assert(set.contains("first") && set.contains("second"));

assert(!set.contains("not there")); // error: should not exist

/\* Testing erase() function. \*/

set.erase("first");

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

set.erase("abracadabraa");

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

set.erase("not there"); // error: does not exist

assert(set.size() == 3); // unchanged size

set.erase("second");

set.erase("third");

set.erase("abracadabra");

set.erase("empty list"); // error: erasing from empty list

/\* Insert fresh values for the set. \*/

set.insert("a");

set.insert("b");

set.insert("d");

set.insert("c");

/\* Testing get() function. \*/

ItemType value;

set.get(1, value);

assert(value == "b");

set.get(2, value);

assert(value == "c");

set.get(0, value);

assert(value == "a"); // first item

set.get(set.size(), value);

assert(value == "a"); // unchanged: size() is too big

set.get(set.size() - 1, value);

assert(value == "d");

set.erase("d");

set.erase("c");

set.erase("b");

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

set.get(1, value);

assert(value == "d"); // unchanged: 1 is too big

set.erase("a"); // empty set

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

set.get(0, value);

assert(value == "d"); // unchanged: empty set

/\* Insert fresh values for set. \*/

set.insert("aloha");

set.insert("hello");

set.insert("bonjour");

set.insert("ciao");

set.insert("nihao");

set.insert("hola");

assert(set.size() == 6);

/\* Create a new set. \*/

Set newSet;

newSet.insert("crazy");

newSet.insert("fool");

newSet.insert("friend");

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

/\* Testing swap() function. \*/

set.swap(newSet);

assert(newSet.size() == 6 && set.size() == 3);

assert(newSet.contains("ciao") && newSet.contains("aloha"));

assert(set.contains("friend") && set.contains("fool") && set.contains("crazy"));

/\* Testing copy constructor. \*/

Set copySet = newSet;

assert(copySet.size() == 6);

assert(copySet.contains("ciao") && copySet.contains("aloha"));

/\* Testing assignment operator. \*/

newSet.insert("some");

newSet.insert("more");

newSet.insert("values");

assert(newSet.size() == 9);

copySet = newSet;

assert(copySet.size() == 9);

assert(copySet.contains("some") && copySet.contains("more") && copySet.contains("values"));

assert(newSet.size() == 9); // original set remains unchanged

/\* Testing unite() function. \*/

Set s1;

Set s2;

Set result;

unite(s1, s2, result); // unite empty sets

assert(result.size() == 0); // should be empty still

/\* Insert new values. \*/

s1.insert("a");

s1.insert("b");

s1.insert("c");

s1.insert("d");

s1.insert("e");

s2.insert("a");

s2.insert("b");

s2.insert("c");

s2.insert("f");

s2.insert("g");

unite(s1, s2, result);

assert(result.size() == 7); // regular case

unite(s1, s1, result);

assert(result.size() == s1.size()); // same array, same size

unite(s2, s2, result); // same array, same size

assert(result.size() == s2.size());

unite(result, result, result); // alias

assert(result.size() == s2.size()); // shouldn't change

/\* Testing subtract() function. \*/

subtract(s1, s2, result); // regular case

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

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

s1 = result;

s2 = result;

subtract(s1, s2, result); // same values

assert(result.size() == 0); // empty set

s1.erase("d");

s1.erase("e");

s2.erase("d");

s2.erase("e");

subtract(s1, s2, result); // empty sets

assert(result.size() == 0); // empty set

s1.insert("a");

s1.insert("b");

s1.insert("c");

s1.insert("d");

s2.insert("a");

s2.insert("b");

subtract(s1, s2, result); // regular case

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

subtract(s1, result, result); // aliasing

assert(result.size() == 2); // should still work

subtract(result, result, result); // all the same

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