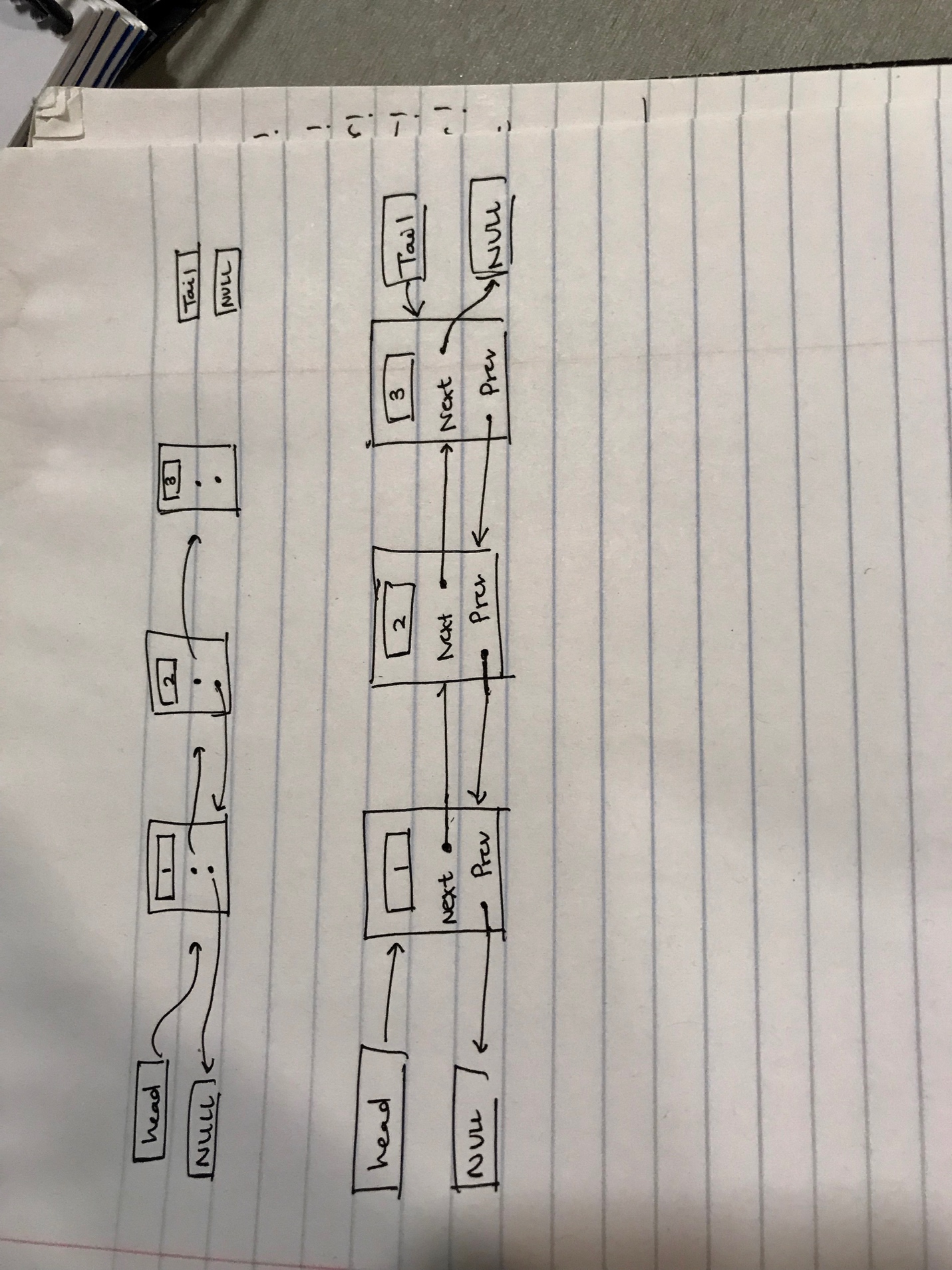
1. My doubly linked list’s implementation:

Member variables include m\_size (keeps track of the number of elements in the list), a pointer to head (top of list), a pointer to tail (bottom of list).

Each node in the list has m\_item (which stores the value), next (which points to the next node), and prev (which points to the previous node).

Nodes are arranged in ascending order (this is accounted for in the insert function). This decision was made because it makes it easier to implement the get function.

1. Pseudocode/implementations:

Constructor{

Since we insert as we go, we don't need to define a capacity

Initialize size to 0, head and tail both pointer to nullptr in an empty linked list

}

Copy constructor{

Re-assign traversal index to the temporary pointer (that stored next node)

Now we need to traverse the linked list to make copies

Index pointers for previous item, new item, head, tail

All initialized to null

Traversal pointer starts off as the other set’s head pointer

While the traversal pointer is not null (loop over valid values)

Allocate a new item in the newItem index

If the new item is other’s head: assign head to that item

Assign the value of the traversal ptr’s item to the new item

new item’s prev = the previous index pointer

if previous index is not null: previous index’s next = newItem index

only when a valid previous exists

Update previous index to the newItem index (latest allocated value)

Outside of loop, tail = last value in newItem index (last allocated value)

Tail’s next is null

Assignment operator{

Check to make sure the other is not the same as the variable we are assigning to

Delete current data using code identical to the destructor

Do the exact same thing as the copy constructor

}

Destructor{

Traverse linked list and delete each item as we go

To do this:

Assign temporary ptr to current node’s next

Delete current node

}

empty{ if m\_size is 0 then true, otherwise false }

size{ returns m\_size }

insert{

Want to insert in ascending order to make get function easier

Case 1: if linked list is empty (i.e head is null)

Allocate a new item and assign value

Head will point to the new item

New item’s previous is null

New item’s next is null

Tail will point to new item

Increment size and return

Case 2: linked list is not empty

The pointer used to traverse the linked list is the checking/check ptr

Traverse the linked list until the value we check exceeds the one we want to add

If we find the value that we want to add in the linked list, return false

Otherwise, break out of the loop

Case 2.1: to add to the end of the list (i.e the checking ptr points to null)

Allocate new item and assign its value to inserted value

Check pointer will now point to tail’s next

Old Tail’s next = new item

New item’s prev = old tail

New item’s next = null

Tail = new item

Increment size and return true

Case 2.2: to add to the front of the list (i.e. checking ptr points to head)

Allocate new item and assign its value

New item’s next = head

Head’s prev = new item

New item’s prev = null

Head = new item

Increment size

Case 2.3: add to the middle of the list

Checking pointer points at the element AFTER where the new one will go

Allocate new item and assign its value

New item’s prev = Checking pointer’s prev

Checking pointer’s prev = new item

New item’s next = checking pointer

New item’s prev’s next = new item (check ptr’s prev’s next = new item)

}

erase {

If linked list is empty, return false

Traverse the linked list using the standard while loop

If we don’t find the value we want to delete, return false

If we do, break out of the while loop

If we broke out of the loop, check pointer points at the item we want to delete

Now we need to check four conditions:

If the element is the first in the list (head == check ptr):

Head = check ptr’s next

If the element is the last in the list (tail == check ptr):

Tail = check ptr’s prev

If the element has a valid prev (not null):

Check ptr’s prev’s next = check ptr’s next

If the element has a valid next (not null):

Check ptr’s next’s prev = check ptr’s prev

Now delete the check ptr’s object

Decrement m\_size and return true

}

contains{

Traverse the linked list using the while loop

Check each value to see if it matches the one we are searching for

If we ever get a match, return true

Return false outside of the loop (after it terminates)

}

get{

Check if input is valid (must be > 0, < size)

Traverse the linked list with a counter

(increments counter every time we move forward a node)

When that counter = pos (argument of get function), we will break.

After breaking, we will point at the item that we need to return

Value = the value that the traversal pointer points to

}

swap{

Swap head pointers:

Store head in a temporary pointer

Head = other’s head

Other’s head = temporary pointer

Swap tail pointers:

Store tail in a temporary pointer

Tail = other’s tail

Other’s tail = temporary pointer

Swap size

Store size in temp variable

Size = other’s size

Other’s size = temporary pointer

}

unite{

copy s1 into result using assignment operator

Loop over s2 (i goes from 0 to s2.size – 1), insert s2 into result

We will get values from s2 using the get function

Get (i, variable holding our value)

Insert value into result

}

subtract {

identical to unite

Except instead of inserting value, we erase it

}

My dump function wrote out all of the elements in the linked list

1. A function to test my implementation with item type int

**void** testInt(){

*//ItemType int*

Set test\_set1; *//c'tor*

*//Tests insert*

test\_set1.insert(12);

test\_set1.insert(150);

test\_set1.insert(200);

*//Tests erase*

test\_set1.erase(200);

*//Tests size, contains*

assert(test\_set1.size() == 2);

assert(test\_set1.contains(200) == **false**);

assert(test\_set1.contains(11) == **false**);

assert(test\_set1.contains(12) == **true**);

ItemType x;

*//Tests get*

assert(test\_set1.get(2,x) == **false**);

assert(test\_set1.get(0,x) == **true** && x == 12);

cerr << "set 1: ";

test\_set1.dump();

cerr << endl;

Set test\_set2;

test\_set2.insert(14);

test\_set2.insert(15);

test\_set2.insert(12);

assert(test\_set2.size() == 3);

assert(test\_set2.contains(12) == **true**);

assert(test\_set2.contains(13) == **false**);

assert(test\_set2.contains(14) == **true**);

assert(test\_set2.insert(12) == **false**);

ItemType y;

assert(test\_set2.get(-1, y) == **false**);

cerr << "set 2: ";

test\_set2.dump();

cerr << endl;

*//Tests swap*

test\_set1.swap(test\_set2);

cerr << "new set 1: ";

test\_set1.dump();

cerr << endl;

cerr << "new set 2: ";

test\_set2.dump();

cerr << endl;

Set test\_set\_result;

test\_set\_result.insert(1124);

cerr << "result before unite: " << endl;

test\_set\_result.dump();

cerr << endl;

*//Tests unite*

unite(test\_set1, test\_set2, test\_set\_result);

cout << "result after unite: " << endl;

test\_set\_result.dump();

cerr << endl;

*//Tests copy constructor*

Set test\_set\_result2 = test\_set1;

cerr << "result2 before subtract: " << endl;

test\_set\_result2.dump();

cerr << endl;

subtract(test\_set1, test\_set2, test\_set\_result2); *//subtract is not working...*

cout << "result2 before subtract: " << endl;

test\_set\_result2.dump();

cerr << endl;

*// Tests assignment operator*

Set test\_operator;

test\_operator = test\_set\_result2;

cerr << "test assignment operator: " << endl;

test\_operator.dump();

cerr << endl;

cout << "Passed all tests" << endl;

}

A function to test my implementation with type std::string.

**void** testString(){

*//ItemType string*

Set test\_set1; *//c'tor*

*//Tests insert*

test\_set1.insert("swim");

test\_set1.insert("run");

test\_set1.insert("jump");

*//Tests erase*

test\_set1.erase("jump");

*//Tests size, contains*

assert(test\_set1.size() == 2);

assert(test\_set1.contains("jump") == **false**);

assert(test\_set1.contains("walk") == **false**);

assert(test\_set1.contains("swim") == **true**);

ItemType x;

*//Tests get*

assert(test\_set1.get(2,x) == **false**);

assert(test\_set1.get(0,x) == **true** && x == "run");

cerr << "set 1: ";

test\_set1.dump();

cerr << endl;

Set test\_set2;

test\_set2.insert("walk");

test\_set2.insert("crawl");

test\_set2.insert("swim");

assert(test\_set2.size() == 3);

assert(test\_set2.contains("swim") == **true**);

assert(test\_set2.contains("run") == **false**);

assert(test\_set2.contains("crawl") == **true**);

assert(test\_set2.insert("crawl") == **false**);

ItemType y;

assert(test\_set2.get(-1, y) == **false**);

cerr << "set 2: ";

test\_set2.dump();

cerr << endl;

*//Tests swap*

test\_set1.swap(test\_set2);

cerr << "new set 1: ";

test\_set1.dump();

cerr << endl;

cerr << "new set 2: ";

test\_set2.dump();

cerr << endl;

Set test\_set\_result;

test\_set\_result.insert("start off");

cerr << "result before unite: " << endl;

test\_set\_result.dump();

cerr << endl;

*//Tests unite*

unite(test\_set1, test\_set2, test\_set\_result);

cout << "result after unite: " << endl;

test\_set\_result.dump();

cerr << endl;

*//Tests copy constructor*

Set test\_set\_result2 = test\_set1;

cerr << "result2 before subtract: " << endl;

test\_set\_result2.dump();

cerr << endl;

subtract(test\_set1, test\_set2, test\_set\_result2); *//subtract is not working...*

cout << "result2 before subtract: " << endl;

test\_set\_result2.dump();

cerr << endl;

*// Tests assignment operator*

Set test\_operator;

test\_operator = test\_set\_result2;

cerr << "test assignment operator: " << endl;

test\_operator.dump();

cerr << endl;

cout << "Passed all tests" << endl;

}