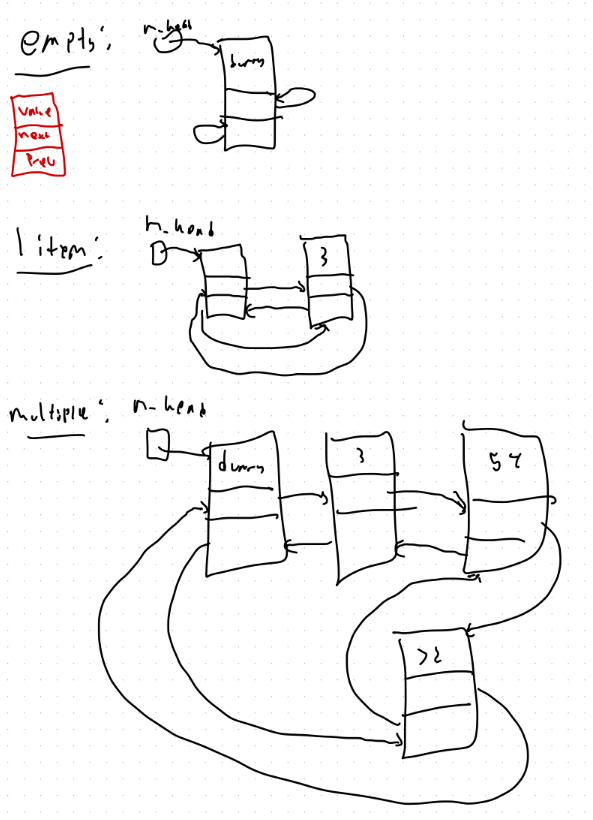
Jason Tran

CS32 Project 2

1/31/23

To implement the Set class, I used the required doubly-link list and made it circular with a dummy node. The dummy node would act as a counter to the nullptr special cases and make it easier to implement, insert/delete. I would always insert stuff at the rear so its next pointer will always be the dummy node, which is simpler and sort the link list right after inserting, from lowest to biggest, from code I used in my HW1. I only really needed a head pointer to access the list and did not use a tail. By using a dummy node, it would mean that the first actual node would be m\_head->next.



**PseudoCode**

**Destructor:**

If size is 0:

Delete dummy node

Return

Else:

Make traversal pointer at first real node

While its not head:

Make a temp ptr to next node

Delete traversal pointer object

Set traversal ptr to the temp ptr

Delete head after

**Copy Constructor:**

Copy size

Make a new list with dummy nodes

Make a traversal ptr for this new list and the other object

While its other object ptr is not at head:

Make a new node

Insert the respective items

New nodes next is head

Previous is the traversal ptr object

The traversal ptr is now the new node

Set head previous to the new node

Advance the other node->next

Sort the fully copied link list by comparing values.

**Assignment Operator:**

If same objects: return

Else: construct temp set = to rhs

Swap the temp and this

**Insert:**

Check if value already contained

If not:

traverse to last node.

Make the new node and input its value

Node's next is head

Update last node next ptr

Node’s prev is last ptr

Update head’s previous to this one

Update size

Sort the new list:

Traverse through each node comparing adjacent values

Stop when hit head.

**Erase:**

If value not in = false

Make a ptr to traverse to the node

Update adjacent nodes ptrs

Delete the object

Decrease size

**Contains:**

Traverse through list

Stop at either the value or if returns to dummy

If it stops at value return true

**Get:**

Check correct bounds

Traverse through the node int pos-1 times

Update the passed in value

**Swap:**

Make temp int to swap sizes

Make temp node ptr to swap heads

**Unite:**

If set1 is the result set:

Make a temp set and add unique set 2 values

Update result set

Return

Else:

Make temp set

Insert all values of set1 by using get

Insert all unique values of set2 by using contains function

Update result set

**butNot:**

If set1 is set2: return empty set

If s1 is result: delete any values of s2 in result

Else:

Make temp set

Insert any items from s1 that is not in s2 with the contains functions

Update result

**Test cases (using unsigned long as itemtype):**

Set test1;

Set test2;

assert(test1.empty()); **//constructed correctly**

assert(test2.empty());

test1.insert(1); **//empty insert**

test1.insert(3); **//normal insert**

test1.insert(2);

test1.insert(4);

test1.insert(2);

assert(test1.size() == 4); **//correctly updates size**

**//contains functions works correctly**

assert(test1.contains(1) && test1.contains(2) && test1.contains(3)

&& test1.contains(4)); **//inserts only unique numbers**

**//get and sorting correctly**

unsigned long x = 9;

test1.get(0, x);

assert(x == 1); **//get functions returns lowest value if pos =0**

test1.get(20, x);

assert(x == 1); **//x stays the same since pos is out of range**

test1.get(3, x);

assert(x == 4); **//x gets the highest value of the linked list**

**//checking assignment operator**

test2 = test1;

assert(test2.size() == 4); **//same size**

assert(test2.contains(1) && test2.contains(2) && test2.contains(3)

&& test2.contains(4)); **//same items**

**//get and sorting correctly**

unsigned long y = 9;

test2.get(0, y);

assert(y == 1); **//get functions returns lowest value if pos =0**

test2.get(20, y);

assert(y == 1); **//x stays the same since pos is out of range**

test2.get(3, y);

assert(y == 4); **//x gets the highest value of the linked list**

**//checking copy constructor**

Set test3 = test1;

assert(test3.size() == 4); **//same size**

assert(test3.contains(1) && test3.contains(2) && test3.contains(3)

&& test3.contains(4)); **//same items**

**//get and sorting correctly with the copy constructor**

unsigned long z = 9;

test2.get(0, z);

assert(z == 1); **//get functions returns lowest value if pos =0**

test2.get(20, z);

assert(z == 1); **//x stays the same since pos is out of range**

test2.get(3, z);

assert(z == 4); **//x gets the highest value of the linked list**

Set test4;

test4.insert(1); **//empty insert**

test4.insert(3); **//normal insert**

test4.insert(2);

test4.insert(4);

test4.insert(2);

test4.insert(5);

test4.insert(7);

test4.insert(6);

assert(test4.size() == 7);**//inserts only unique**

assert(test4.contains(1) && test4.contains(2) && test4.contains(3)

&& test4.contains(4) && test4.contains(5) && test4.contains(6) && test4.contains(7)); **//expected numbers are inserted in**

unsigned long g = 9;

test2.get(2, g); **//get, transfers the 3rd biggest in to g.**

assert(g == 3); **//correctly sorted check**

**//erase**

test4.erase(2);

test4.erase(5);

test4.erase(7);

test4.erase(6);

assert(test4.size() == 3); **//erased these 4 numbers correctly**

assert(test4.contains(7) == false); **//contains is correct**

assert(!test4.erase(6) && test4.size() == 3); **//doesn't erase 6 and keeps same size**

Set test5;

Set test6;

test5.insert(1); **//empty insert**

test5.insert(2); **//normal insert**

test5.insert(3);

test5.insert(4);

test6.insert(6); **//empty insert**

test6.insert(7); **//normal insert**

test6.insert(8);

test6.insert(9);

test6.insert(10);

**//swap**

test5.swap(test6);

assert(test5.size() == 5); **//sizes swapped**

assert(test6.size() == 4);

assert(test5.contains(6) && test5.contains(7) && test5.contains(8) && test5.contains(9) && test5.contains(10)); **//test5 now contains tes6 items**

assert(test6.contains(1) && test6.contains(2) && test6.contains(3) && test6.contains(4)); **//test 6 now contains test5 items**

**//unite**

Set test7;

Set test8;

//1,2,3,4

test7.insert(1);

test7.insert(2);

test7.insert(3);

test7.insert(4);

assert(test7.size() == 4);

**//6,7,8,9,10**

test8.insert(6);

test8.insert(7);

test8.insert(8);

test8.insert(3);

test8.insert(4);

test8.insert(9);

test8.insert(10);

assert(test8.size() == 7);

Set result1;

unite(test7, test8, result1);

assert(result1.size() == 9); **//only unites the unique numbers**

assert(result1.contains(1) && result1.contains(2) && result1.contains(3) && result1.contains(4) && result1.contains(6) && result1.contains(7)

&& result1.contains(8) && result1.contains(9) && result1.contains(10)); **//result1 contains the expected numbers**

**//unite aliasing check**

unite(test7, test8, test7); **//first set and result set are the same**

assert(test7.size() == 9); **//still needs to add in the unique numbers of the second set**

assert(test7.contains(1) && test7.contains(2) && test7.contains(3) && test7.contains(4) && test7.contains(6) && test7.contains(7)

&& test7.contains(8) && test7.contains(9) && test7.contains(10)); **//test7 should also include #s from test8**

**//butNot**

Set test9;

Set test10;

**//1,2,3,4**

test9.insert(1);

test9.insert(2);

test9.insert(3);

test9.insert(4);

**//3,4,5,6**

test10.insert(3);

test10.insert(4);

test10.insert(5);

test10.insert(6);

Set result2;

butNot(test9, test10, result2);

assert(result2.size() == 2); **//only includes 1st set unique items**

assert(result2.contains(1) && result2.contains(2));**//which are 1 and 2**

**//aliasing check**

butNot(test9, test9, test9);

assert(test9.size() == 0);/**/if set1 and set 2 are the same then the result has to be nothing**

**//reinsert the #s**

test9.insert(1);

test9.insert(2);

test9.insert(3);

test9.insert(4);

test10.insert(3);

test10.insert(4);

test10.insert(5);

test10.insert(6);

**//aliasing check for s1 and result**

butNot(test9, test10, test9);

assert(test9.size() == 2); **//should still only contain the 2 unique and make sure none of s2 is in it.**