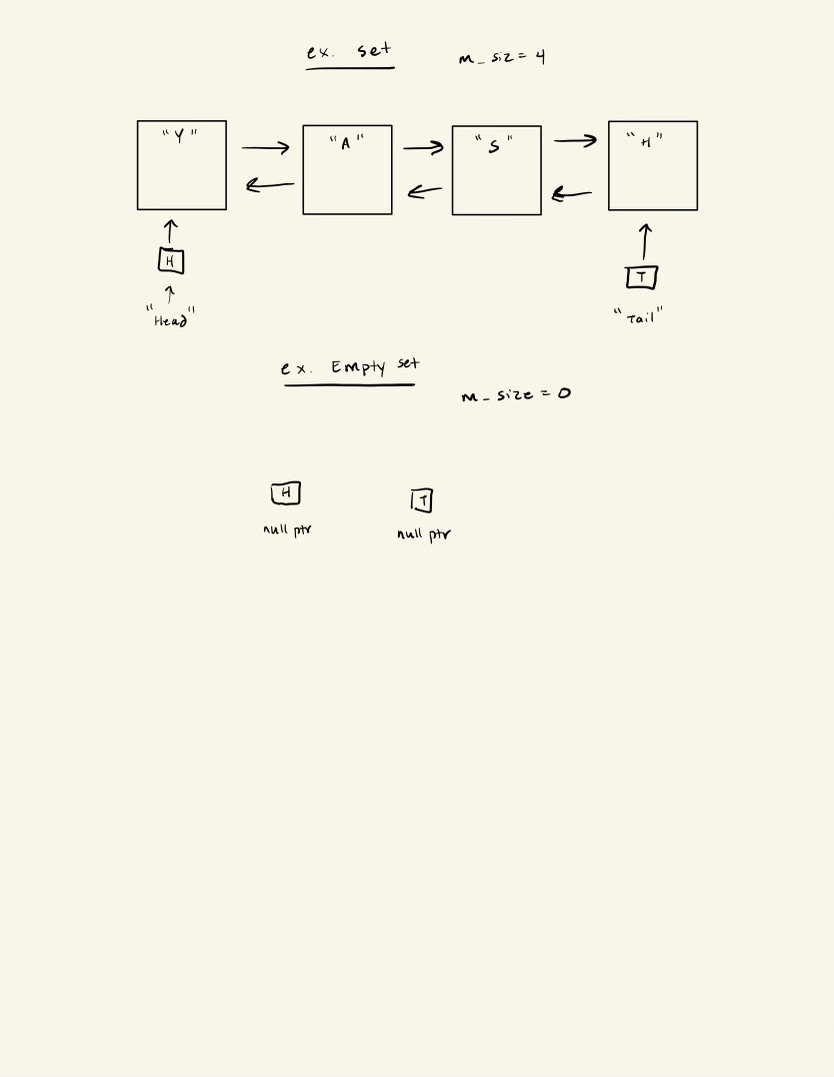
**Yash Shah CS 32 Project 2 4/20/21 --- 405565567**

**Design of Doubly-Linked List Implementation ---**

My design of a doubly-linked list implementation begins with the creation of the structure Node that stores a datatype ItemType which holds a value that can be stored in a Node. The structure Node also holds a next pointer and previous pointer to ensure that each item can be parsed through in either way (forward and backward). Each Node has a pointer Head and Tail to indicate the beginning and end of the linked list-- his is defined in the class implementation as a head and tail of object type Node-- and a size to hold the size of the linked list. Here is a picture to guide the design of the linked list:



List is not circular, no dummy node.

**Pseudocode for non-trivial algorithms ---**

**Destructor –**

Create a node pointer at the head of the linked list.

Loop until p hits a null pointer –

within the loop, create a temp value that holds the next p value and delete the current p value. then set p equal to the next element of the linked list

**Copy constructor –**

Check whether Set other’s linked list is empty, that is, when the head and tail pointer are both null pointers.

If they are, set the Set this’s head and tail pointer as null pointes as well.

If they are not, create a node that head is set to. set the head’s value as the other head’s value. Then set the previous and next positions of the nodes to null pointers. Then set the tail value to the head’s value of this object. Now you can create two pointers, the first that refers to the head of the list this, and second that refers to the head of the list other. Iterate through the list of other, and for each next data value for the this pointer, have it set to the next data value for other pointer. Then you link the next and previous pointers of the this pointer, increment the this and other pointer, and assign the tail to the last node.

Finally, you set the size of this object to the other objects.

**Assignment Operator –**

If the objects contain the same elements of the list, the function simply just returns the this object.

If not, set a temporary copy of other and swap the set of this with the temporary copy.

**Insert –**

If the value already exists in the set, return false.

If not, create a new node p and assign its data value to the value inputted into the function.

If the list is empty, then you have just one element in the list that is both the head and the tail of the list.

If the list is not empty, then add the value to the front of the list while shifting the rest of the elements accordingly.

Increment the size of the list and return true.

**Erase – FINISH**

Check if the list is empty or doesn’t contain the value needed to be erased. If so, return false.

If not, if the value being erased is either at the head, it sets the head to the next pointer and deletes the previous head pointer, decrements the size, and returns true. Similarly, if the value being erased is at the tail, it sets the tail to the previous pointer and deletes the next tail pointer, decrements the size, and returns true.

If its neither the head or tail and its in the middle of the list, a pointer increments through the list and stops until one before the element that needs to be taken out and deleted. The pointer works around the element and sets the next to next element as the next element, and disconnects the element from the linked list. then we use the destructor to delete that element and recover our memory. It then decrements the size and returns true.

If true hasn’t already been returned, the function will assume something went wrong and will return false;

**Contains –**

Check if the list is empty. If so, return false.

If not, iterate through the list until you either find the value that you are looking for and return true, or don’t find the value and return false.

**Get –**

Check if the list is empty or the position is out of range (less than 0 or greater than the size of the list). if one of these is true, return false.

If not, create a double while loop that iterates through the loop and compares each value with every other value in the set. Track for each value how many values are greater than it and store it in a variable num\_less. If the num\_less is equal to the pos input, then set the value to the data value of the value being compared with every other element in the list.

return true if and when the value is found, if not return false.

**Swap –**

Check that the list of this object and other object aren’t empty, and that they don’t already equal each other. If that’s true, then just return and exit the function.

If not, then swap the nodes of the heads, the tails, and the value of the sizes between the two objects.

**Unite –**

Loop through separate loops of both s1 and s2, and get the value of each element using the get function defined above. Then insert that value into result (the insert function checks to ensure that value isn’t already contained in result).

**Difference --**

Loop through separate loops of both s1 and s2, and get the value of each element using the get function defined above. Then check to make sure the other list doesn’t already contain that element, and if so, insert that value into result (the insert function checks to ensure that value isn’t already contained in result). If not, the value isn’t inserted into result.

**List of Test Cases ---**

// test constructor & empty //

/\*

Set set;

assert(set.empty());

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

\*/

// test copy constructornot //

/\*

Set ss;

ss.insert("1");

ss.insert("2");

ss.insert("3");

ss.insert("4");

ss.insert("5");

ss.insert("6");

Set ss2(ss);

assert(ss2.size() == 6); // tests copy constructor for multiple elements

Set ss3;

ss3.insert("1");

Set ss4(ss3);

assert(ss4.size() == 1); // tests constructor for one element

Set ss5;

Set ss6(ss5); //

assert(ss6.size() == 0); // tests constructor for no element

\*/

// test assignment operator -- done//

//test copy constructor

/\*

Set set1;

Set set2;

set2.insert("Y");

set2.insert("A");

set2.insert("s");

set2.insert("H");

­­­­

set1 = set2; // values copy over

\*/

// test size //

//test 1 -- size is 0

/\*

Set set;

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

\*/

//test 2 -- size is any finite amount

/\*

Set set;

set.insert("1");

set.insert("2");

set.insert("3");

set.insert("4");

set.insert("5");

set.insert("6");

set.insert("7");

set.insert("8");

set.insert("9");

set.insert("10");

assert(set.size() == 10);

\*/

// test contains//

//test 1 -- tests existing and non-existing values

/\*

Set set;

set.insert("1");

set.insert("2");

set.insert("3");

set.insert("4");

set.insert("5");

set.insert("6");

set.insert("7");

set.insert("8");

set.insert("9");

set.insert("10");

assert(set.contains("1")); //checks tail

assert(set.contains("10")); //checks head

assert(set.contains("5")); //checks middle

assert(!set.contains("chalupa")); // checks non-existant value

\*/

// test insert //

/\*

Set set;

assert(set.empty()); //check empty before insert

set.insert("firstVal"); //add initial value works

set.insert("secondVal"); //adding multiple values works

set.insert("thirdVal");

set.insert("firstVal"); // already contains value, doesn't insert

assert(set.size() == 3); //check m\_size updates

\*/

// test erase //

//test 1 -- erase first value from list

/\*

Set set;

set.insert("firstVal");

set.insert("secondVal");

set.insert("thirdVal");

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

//"first" value from list because at head position

set.erase("thirdVal");

assert(set.size() == 2); // check m\_size updates

set.erase("chalupa"); //can't erase a value that doesn't exist

assert(set.size() == 2); //make sure m\_size doesn't change

\*/

//test 2 -- erase middle value from list

/\*

Set set;

set.insert("firstVal");

set.insert("secondVal");

set.insert("thirdVal");

assert(set.size() == 3); //should equal 3

set.erase("secondVal");

assert(set.size() == 2); //should equal 2

\*/

//test 3 -- erase last value from list

/\*

Set set;

set.insert("firstVal");

set.insert("secondVal");

set.insert("thirdVal");

set.insert("fourthVal");

set.insert("fifthVal");

set.insert("sixthVal");

assert(set.size() == 6); //should equal 6

set.erase("sixthVal");

assert(set.size() == 5); //should equal 5

\*/

//test 4 -- erase non-existant value (should not change anything)

/\*

Set set;

set.insert("firstVal");

set.insert("secondVal");

set.insert("thirdVal");

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

set.erase("fourthVal");

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

\*/

//test 5 -- erase when only one element in list

/\*

Set set;

set.insert("firstVal");

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

set.erase("firstVal");

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

\*/

//test 6 -- erase when no value in array

/\*

Set set;

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

set.erase("secondVal");

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

\*/

// test get //

/\*

ItemType ex;

Set ss;

ss.insert("wubba");

ss.insert("lubba");

ss.insert("dub");

assert(ss.get(0, ex) && ex == "wubba"); //pos = 0 and correct val

assert(ss.get(1, ex) && ex == "lubba"); //pos < m\_size and correct val

assert(ss.get(2, ex) && ex == "dub"); //pos < m\_size and correct val

assert(ss.get(3, ex) == false); //pos = m\_size

assert(ss.get(10, ex) == false); //pos > m\_size

assert(ss.get(-5, ex) == false); //pos < 0

\*/

// test swap //

/\*

Set set1;

Set set2;

Set set3;

Set set4;

Set set5;

set1.insert("H");

set1.insert("S");

set1.insert("A");

set1.insert("Y");

set2.insert("Y");

set2.insert("A");

set2.insert("S");

set2.insert("H");

set2.insert("a");

set2.insert("b");

set2.insert("c");

set2.insert("d");

set2.insert("e");

set2.insert("f");

set2.insert("g");

set2.insert("h");

set1.swap(set2); //tests with same sizes

set2.swap(set3); //tests for difference in sizes

set3.swap(set4); //tests for when one set is empty and other is not

set5.swap(set3); //tests for when both sets are empty

\*/

// test unite & difference //

/\*

Set ss1;

ss1.insert("2");

ss1.insert("8");

ss1.insert("3");

ss1.insert("9");

ss1.insert("5");

Set ss2;

ss2.insert("6");

ss2.insert("3");

ss2.insert("8");

ss2.insert("5");

ss2.insert("10");­­­­­

Set ss3;

ss3.insert("1");

ss3.insert("2");

ss3.insert("3");

Set ss4;

ss4.insert("1");

ss4.insert("2");

ss4.insert("3");

Set ss5;

Set ss6;

Set result;

Set result2;

result2.insert("9");

result2.insert("10");

Set result3;

result3.insert("1");

result3.insert("2");

result3.insert("3");

Set result4;

unite(ss1,ss2,result); //different sets with some similar values

unite(ss1,ss3,result3); //ss1 is a different set, but ss3 is the same set as result3

unite(ss3,ss4,result3); //all sets are the same

unite(ss3,ss4,result); //2 input sets are the same but result set is different

unite(ss5,ss6,result4); //both input sets are empty

difference(ss1, ss2, result2); //different sets with some similar values

difference(ss1, ss3, result3); //ss1 is a different set, but ss3 is the same set as result3

difference(ss3, ss3, result3); //all sets are the same

difference(ss2, ss3, result); //2 input sets are the same but result set is different

difference(ss5, ss6, result4); //both input sets are empty

\*/