Description of Design

Within the WeddingGuest class, I created a Node Struct. Each WeddingGuest has two pointers of type Node (one head that points to the first node in the list and one tail that points to the last node in the list) and a variable called m\_num which stores the number of Nodes in the list. Each Node, which represents a guest on the list, has two pointers of type Node (one that points to the node before it and one that points to the node after it), a first name variable and a last name variable, each of type string, and a value variable of type GuestType.

Obstacles

One of the obstacles I overcame was the inviteGuest function as a whole. Because of the many different cases that I had to consider, as well as how those different cases affected how the guest would be implemented into the list, this function was the most difficult and time consuming. I also realized as I continued to write my program that almost every other function in the spec relied upon this function in order to work. If there was just one small mistake in this function, the rest of my program would have errors. This meant that this obstacle grew larger and larger as the code went on, as I would be less and less sure where my errors were coming from.

Another obstacle I overcame was in the attestGuests function. As I tested this function, I kept getting an incorrect output and could not figure out why. I forgot that as I recursed through the code, if I crossed a name off the list, that the variable m\_num would decrement, but my i value would increment, meaning that I would skip a Node. This resulted in some problems that I had to work through.

Pseudocode

inviteGuest(){

If name is already on list- return false;

If list is empty, add a new node to the front of the list, make it the new head and tail, assign values and give it next and prev pointers- return true;

Make a new pointer that points to the head of the list;

Loop through, the list, incrementing the pointer each time through the loop;

If the pointer points to a node with the same last name as the given lastName, move on;

Check if the pointer points to the last node.

If not, make new pointer to point to the last node with the same last name;

Recurse through code until we get to the last node with same last name;

If we are now at the end of the list, add a new node, assign values and make it the new tail- break out of loop;

If not, add this name to the end of this list of people with this last name- break out of loop;

If we did not get to the last name before finding a person with a name that ours comes before, check if we are at the first node;

If so, assign values and make node the new head- break out of loop;

If not, add node before the name that comes after alphabetically, assign values and prev/next pointers- break out of loop;

If last names are different, repeat this process starting after looking through the list of people with the same last name- check for first name that comes after ours, and check if we are at the beginning or end of the list;

Break out of loop;

Increment m\_num and return true;

}

AlterGuest(){

If invitedToTheWedding() is false, return false;

A pointer p points to the head;

While p is not the nullptr, run the loop;

Check if pointer first and last names are equal to parameter first and last names;

If so, set the Node that p is pointing to to the value given- return true;

If not, set p to the next node;

Once loop is done, return false;

}

inviteOrAlter(){

If alterGuest() is true with parameters given, return true;

If not, call inviteGuest() with parameters given- return true;

}

crossGuestOff(){

If name is not on list, return false;

If list is empty, return false;

Recurse through list with a pointer;

If the first and last name of the node being pointed to match the parameter first and last names, move on;

If not, move the pointer to the next node;

If there is only one value, and it is the name we are looking for, delete the pointer, set head and tail to nullptr, decrement m\_num, and return true;

If pointer points to head, get rid of the head;

If pointer points to tail, get rid of the tail;

If none of these cases are true, set prev and next pointers so that the node is no longer in the loop;

Once outside of if statement, decrement m\_num, delete the pointer, and return true;

If loop is completed, return false;

}

invitedToTheWedding(){

Go through loop with a pointer, as long as pointer does not point to nullptr;

If the first and last name of the node being pointed to equal the parameters given, return true;

If not, make pointer point to next node;

If we make it through the entire list, return false;  
}

matchInvitedGuest(){

If name is not on list, return false;

Create pointer and loop to go through list as long as pointer does not point to nullptr;

If the first and last name of the node being pointed to match the parameters given, set the value parameter equal to the node’s value and return true;

If not, set the pointer to the next node;

If we make it through the loop, return false;

}

verifyGuestOnTheList(){

Make sure i is in bounds;

If not, return false;

Create a pointer and a loop that stops when we get to the node at the ith index;

Assign the parameter values equal to the values of the node that is pointed to and return true;

}

swapWeddingGuests(){

Set a temporary pointer equal to head;

Set the head given in the parameter equal to head;

Set head equal to the head given in the parameter;

Repeat this process for tail and m\_num;

}

joinGuests(){

Create boolean variable set to true to return later;

If one of the original lists is empty, set the new list equal to the non-empty list and return true;

If both lists are empty, set the new list equal to one of the empty lists and return true;

Create variables to store first and last name, 2 different values and set the new list equal to the first list;

Create a for loop that runs as long as verifyGuestOnTheList() returns true for the second list- for the parameters, put the variables with nothing in them that we just created, so that each time through the loop we have the next set of names/value;

Attempt to invite the guest- if the name is already on the list, check if the names have the same values;

If so, set the bool variable to false and cross the name off the list;

Once the loop completes, return the bool variable;

}

attestGuests(){

Create boolean variables corresponding to if the first and last names can be wildcards, respectively;

Created two strings for first and last names and a GuestType variable;

Set the bool variables to true if their corresponding string is a wildcard;

Set the result list equal to the first list;

If both strings are wildcards, return;

If the first name is a wildcard but the last name isn’t, check each node in the new list with a loop to see if the last names match;

If not, cross the guest off and decrement the loop variable;

Return at the end of the loop;

If the last name is a wildcard but the first name isn’t, check each node in the new list with a loop to see if the first names match;

If not, cross the guest off and decrement the loop variable;

Return at the end of the loop;

If neither string is a wildcard, loop through the list;

Check if each node has the exact same first and last names;

If not, cross the name off and decrement the loop variable;

Return after the loop is done;

}

Test Cases

Constructors/Assignment Operators

WeddingGuest g; //test default constructor

WeddingGuest g2(g); //test copy constructor

WeddingGuest g3;

g3 = g2; //test assignment operator

g3 = g3; //test for aliasing

Trivial Member Functions

WeddingGuest g;

assert(g.noGuests()); //test noGuests function

assert(g.guestCount() == 0); //test guestCount function when 0 guests

assert(g.guestCount() == 4); //test guestCount with multiple guests

The following test cases will be writing in the following format:

**Function()**

(example, parameters, test) //explanation of test

**inviteGuests()**

(“”,””,20) //test case with empty strings

(“John”, “Doe”, 0) //test case with actual name

(“Pete”, “Best”, 20)

(“John”, “Best”, 20) //test for same last name  
(“Pete”, “Best”, 20) //test for same name

(“pete”, “best”, 20) //test for same name with no capitals

(“Pete”, “Best”, 20)

(“Paul”, “Best”, 20)

(“Play”, “Best”, 20)

(“Zeke”, “Best”, 20) // test for name at end of list

(“Pete”, “Best”, 20)

(“Paul”, “Best”, 20)

(“Play”, “Best”, 20)

(“Adam”, “Best”, 20) //Test for beginning of list

**alterGuest()**

(“Paul”, “Best”, 20) //test for name on list

(“Chris”, “Best”, 21) //test for name not on list

(“Paul”, “Best”, 20) //test for name with same value

(“Paul”, “Best”, 21) //check empty list

(“paul”, “best”, 21) //no capitals

**inviteOrAlter()**

(“Chris”, “Head”, 20) //name not on list

(“Chris”, “Head”, 21) //name on list

(“Chris”, “Head”, 20) //name on list but same value

(“Christopher”, “Head”, 20) //name not on list but different value

**crossGuestOff()**

(“Jane”, “Doe”, 20) //name not on list

(“Chris”, “Head”, 20) //name on list

(“Chris”, “Head”, 20) //list is empty

(“Head”, “Head”, 20) //cross off head

(“Jane”, “Tail”, 20) //cross off tail

(“Jane”, “Doe”, 20) //there is only one item in list and it is the one we’re looking for

**invitedToTheWedding()**

(“Joe”, “D”, 20) //not on list

(“Jane”, “Doe”, 20) //on the list

(“Joe”, “Tail”, 20) //is the tail

(“Joe”, “Head”, 20) //is the head

**matchInvitedGuest()**

(“Craig”, “Eloh”, 20) //name not on list

(“Craig”, “Eloh”, 20) //name on list with same value

(“Craig”, “Eloh”, 22) //name on list with different value

(“Craig”, “Eloh”, 20) //empty list

(“”, “”, 20) //empty strings

**verifyGuestOnTheList()**

(50, “Craig”, “Eloh”, 20) //i is out of bounds

(5, “”, “”, 20) //empty strings

(5, “Craig”, “Eloh”, 20) //normal test

(10, “Craig”, “Eloh”, 20) //i is tail

(0, “Craig”, “Eloh”, 20) //i is first element

**swapWeddingGuests()**

WeddingGuest groomsmen;

groomsmen.inviteGuest("Pete", "Best", 5);

groomsmen.inviteGuest("John", "B", 4);

groomsmen.inviteGuest ("Paul", "Wudd", 3);

groomsmen.inviteGuest ("Mike", "au", 2);

WeddingGuest g;

g.inviteGuest ("mIke", "Wu", 44);

g.inviteGuest ("Mic", "Au", 37);

g.inviteGuest ("Mike", "Au", 38);

//test for different sized lists

WeddingGuest groomsmen;

groomsmen.inviteGuest("Pete", "Best", 5);

groomsmen.inviteGuest("John", "B", 4);

groomsmen.inviteGuest ("Paul", "Wudd", 3);

groomsmen.inviteGuest ("Mike", "au", 2);

WeddingGuest g;

g.inviteGuest ("mIke", "Wu", 44);

g.inviteGuest ("Mic", "Au", 37);

g.inviteGuest ("Mike", "Au", 38);

g.inviteGuest ("Long", "A", 41);

//test for same size lists

WeddingGuest groomsmen;

WeddingGuest g;

g.inviteGuest ("mIke", "Wu", 44);

g.inviteGuest ("Mic", "Au", 37);

g.inviteGuest ("Mike", "Au", 38);

g.inviteGuest ("Long", "A", 41);

//one list is empty

WeddingGuest groomsmen;

groomsmen.inviteGuest("Pete", "Best", 5);

groomsmen.inviteGuest("John", "B", 4);

groomsmen.inviteGuest ("Paul", "Wudd", 3);

groomsmen.inviteGuest ("Mike", "au", 2);

WeddingGuest g;

//other list is empty

WeddingGuest groomsmen;

WeddingGuest g;

//both lists empty

**joinGuests()**

(empty, full, new) //one list is empty

(full, empty, new) //other list is empty

(empty, empty, new) //both lists are empty

(ten, twenty, new) //different sizes

(“Joe”, “Tail”, 20) (“Joe”, “Tail”, 20) (“Joe”, “Tail”, 20)

(“Joe”, “Tail”, 20) (“Joe”, “Tail”, 20) (“Joe”, “Tail”, 20) //both have the exact same elements

(“Joe”, “Tail”, 20) (“Joe”, “Tail”, 20) (“Joe”, “Tail”, 20)

(“Joe”, “Tail”, 22) (“Joe”, “Tail”, 22) (“Joe”, “Tail”, 22) //same names different values

**attestGuests()**

(“\*”,”\*” ,old, new) //both wildcards

(“\*”,”Head” ,old, new) //first wildcard

(“Chris”,”\*” ,old, new) //last wildcard

(“Chris”,”Head” ,old, new) //neither wildcard

(“Chris”,”\*” ,old, new) //no matches

(“Chris”,”\*” ,old, new) //yes matches

(“Chris”,”\*” ,old, new) //