**Project 2 Report**

**Implementation**

I decided to use a doubly liked list with a head and tail pointer as the private variables for the implementation. I decided to include a tail pointer as it makes the list easier to traverse in the reverse direction as the tail serves as a starting point. However, in hindsight, this project did not include any function that required such a method for reverse traversing. Instead, because I included a tail pointer, I had to make sure that its location is being updated when make changes to the last node. In some functions, this complicated the code as I had to separately consider this edge case. I finally decided to stick with this implementation, however, as I believe it’s a good design practice and will be helpful for future functions.

\*Insert drawing of empty and non-empty linked list with values in each node. \*

**Obstacles overcome**

1. Since this was my first time implementing a doubly linked list with a tail pointer, I encountered numerous bad access errors by accidentally following null pointers. Eventually, I got a hang of constantly updating each of a) head’s next pointer b) node’s next and previous pointers and c) tail’s previous pointer.
2. I also found the spec a bit confusing since the skeleton provided did not have the Node struct. Therefore, I was unsure where to include it and which values it should hold. However, this was clarified after reading some posts on Piazza.
3. I was initially struggling with implementing the non-member functions as I could not think of a way to traverse a list without starting with its head pointer. However, I was able to figure this out by looking at the spec’s way of outputting a list’s values using the guestCount and verifyGuestOnTheList member functions. I also struggled with the aliasing error as my initial implementation of the member functions involved deleting odJoined (or odResult) first and then inviting the guests that met a certain criterion. I eventually found a way to work around this by creating a temp WeddingGuest.

**Pseudocode for non-trivial algorithms:**

Note: some functions have not been listed here because their implementation was either identical to a function in HW1 for the singly linked list or they have been implemented in almost the same way as another function that has been listed here. (For example, verifyGuestOnTheList, invitedToTheWedding and matchInvitedGuest are very similar in their working so only verifyGuestOnTheList has been listed below)

1. **inviteGuest:**

create a new node and set its values as required by the function’s parameters

if list is empty:

make head and tail point to n

set n’s next and prev pointers to nullptrs

create a node pointer p and assign it to head

while p is not the nullptr{

if (n->lastName > p->lastName) advance p

else if (n->lastName == p->lastName) {

if (n->firstName > p->firstName) advance p

else if (n->firstName == p->firstName) return false

else if neither then break the loop

}

else if neither then break the loop

}

If p is the nullptr here, add the node to the end of the list

Otherwise, add it between p->prev and p and adjust the linking pointers accordingly

1. **AlterGuest**

create a node pointer p and assign it to head

advance p until it either equals the nullptr OR both first and last name match the function parameters

if p equals the nullptr (which implies it reached the end without matching) return false

otherwise p->value = value and return true

1. **crossGuestOff**

create a node pointer p and assign it to head

advance p until it either equals the nullptr OR both first and last name match the function parameters

if p equals the nullptr (which implies it reached the end without matching) return false

otherwise p points to the node that needs to be deleted

if p points to the head, delete the first node and adjust links

if p points to the tail, delete the last node and adjust links

if p points to a node in the middle, adjust the previous and next nodes links to exclude this node and then delete the node

return true indicating that a node has been deleted

1. **verifyGuestOnTheList**

if i is out of range (i < 0 or i >= guestCount()) then return false

otherwise create a node pointer equal to head and traverse the list i number of times to reach the desired node

copy that node’s firstName, lastName and value into the function parameters respectively

1. **joinGuests**

create a temp WeddingGuest that will eventually be assigned to odJoined

use verifyGuestOnTheList and guestCount to traverse through odOne:

if any of the names in odOne match an invited guest in odTwo, store the value and:

if the respective values in odOne and odTwo are equal, add the guest to odJoined

otherwise prepare to return false

else (i.e. no match):

add the guest to odJoined

repeat this process symmetrically, traversing odTwo and checking for matches with odOne, adding to odJoined when required

odJoined = temp

return true unless at some point, the if-branch that prepared to return false was entered, in which case return false.

1. **attestGuests**

create a temp WeddingGuest that will eventually be assigned to odResult

traverse through odOne and check the following cases:

if (fsearch and lsearch are both wildcards) assign temp to odOne

else if (one is a wildcard and the other isn’t) add all guests to odResult where the other matches

else if neither are wildcards, (if such a guest exists) add the guest whose first name and last name both match fsearch and lsearch respectively.

odResult = odOne

**List of test cases:** (copy pasted from my main routine with explanations as comments)

Note: All cases were tested with GuestType int as it was the most convenient and would work the same way with any other type.

WeddingGuest groomsmen;

groomsmen.inviteGuest ("Tony", "Ambrosio", 40);

groomsmen.inviteGuest("", "", -1);

groomsmen.inviteGuest ("Mike", "Wu", 43);

groomsmen.inviteGuest ("Robert", "Wells", 44);

groomsmen.inviteGuest("Roberu", "Wells", 10);

groomsmen.inviteGuest("Andrew", "Wells", 10);

groomsmen.inviteGuest("X-Last", "Y-Guy", 99);

groomsmen.inviteGuest("A-First", "AA-Guy", 0);

assert(!groomsmen.inviteGuest("Robert", "Wells", 10)); // duplicate returns false

GuestType val;

assert(groomsmen.matchInvitedGuest("Robert", "Wells", val));

assert (val == 44); // and keeps the first entry's value

assert(groomsmen.guestCount() == 8); // checking guest count's functionality

string first; string last;

assert(groomsmen.verifyGuestOnTheList(0, first, last, val));

assert(first == "" && last == "" && val == -1); // verifying first in order

assert(groomsmen.verifyGuestOnTheList(7, first, last, val));

assert(first == "X-Last" && last == "Y-Guy" && val == 99); // verifying last in order

assert(groomsmen.alterGuest("Tony", "Ambrosio", 3)); // returns true if name exists

assert(groomsmen.matchInvitedGuest("Tony", "Ambrosio", val));

assert(val == 3); // updates value correctly

val = -999;

assert(!groomsmen.alterGuest("Mike", "Wuu", 2)); // false if guest doesn't exis

assert(!groomsmen.matchInvitedGuest("Mike", "Wuu", val)); // false " "

assert(val == -999); // should leave the value unchanged

assert(groomsmen.inviteOrAlter("Tony", "Ambrosio", 41)); // always returns true

assert(groomsmen.matchInvitedGuest("Tony", "Ambrosio", val));

assert(val == 41); // updates correctly

assert(groomsmen.inviteOrAlter("Tony", "Ambro", 69)); // always returns true

assert(groomsmen.invitedToTheWedding("Tony", "Ambro")); // invites correctly

assert(groomsmen.guestCount() == 9); // increments guest count

WeddingGuest groomsmen\_copy(groomsmen); // checking that copy constructor works

assert(groomsmen\_copy.guestCount() == groomsmen.guestCount());

for (int n = 0; n < groomsmen.guestCount(); n++)

{

string first1;

string last1;

int val1;

groomsmen.verifyGuestOnTheList (n, first1, last1, val1);

string first2;

string last2;

int val2;

groomsmen\_copy.verifyGuestOnTheList (n, first2, last2, val2);

assert(first1 == first2 // comparing each node individually

&& last1 == last2

&& val1 == val2);

}

WeddingGuest groomsmen\_copy2;

groomsmen\_copy2 = groomsmen; // checking that assignment op works

assert(groomsmen\_copy2.guestCount() == groomsmen.guestCount());

for (int n = 0; n < groomsmen.guestCount(); n++)

{

string first1;

string last1;

int val1;

groomsmen.verifyGuestOnTheList (n, first1, last1, val1);

string first2;

string last2;

int val2;

groomsmen\_copy2.verifyGuestOnTheList (n, first2, last2, val2);

assert(first1 == first2 // comparing each node individually

&& last1 == last2

&& val1 == val2);

}

assert(groomsmen.crossGuestOff("Mike", "Wu"));

assert(!groomsmen.invitedToTheWedding("Mike", "Wu")); // not invited anymore

assert(groomsmen.guestCount() == 8); // decrements guestCount

assert(!groomsmen.crossGuestOff("Andwho", "Wells")); // false if not invited to the wedding in the first place

assert(groomsmen.invitedToTheWedding("Andrew", "Wells")); // this guy is still there

assert(groomsmen.guestCount() == 8); // no reason to decrement

WeddingGuest small;

assert(small.noGuests());

assert(!small.crossGuestOff("", ""));

small.inviteGuest("Nakul", "Khambhati", 13);

assert(!small.noGuests());

assert(small.guestCount() == 1);

assert(small.crossGuestOff("Nakul", "Khambhati"));

assert(small.noGuests());

small.inviteGuest("Lonely", "Person", 1);

WeddingGuest bridesmaids; // using spec's example to test functions that take as input multiple lists

bridesmaids.inviteGuest("Serra", "Park", 39);

bridesmaids.inviteGuest("Saadia", "Parker", 37);

assert(!bridesmaids.invitedToTheWedding ("",""));

bridesmaids.inviteGuest("Patricia", "Kim", 39);

bridesmaids.inviteGuest("", "", 21);

bridesmaids.inviteGuest("Kristin", "Livingston", 38);

assert(bridesmaids.invitedToTheWedding ("", ""));

bridesmaids.crossGuestOff("Patricia", "Kim");

assert(bridesmaids.guestCount() == 4

&& bridesmaids.invitedToTheWedding("Serra", "Park")

&& bridesmaids.invitedToTheWedding ("Saadia", "Parker")

&& bridesmaids.invitedToTheWedding ("Kristin", "Livingston")

&& bridesmaids.invitedToTheWedding ("", ""));

val = -999;

bridesmaids.matchInvitedGuest("Kristi", "Livingston", val);

assert(val == -999);

assert(bridesmaids.guestCount() == 4); // based on previous actions, there should be 4 guests

bridesmaids.swapWeddingGuests(groomsmen);

assert(bridesmaids.guestCount() == 8);

assert(groomsmen.guestCount() == 4); // counts have been swapped

assert(bridesmaids.verifyGuestOnTheList(7, first, last, val));

assert(first == "X-Last" && last == "Y-Guy"); // successfully swapped

WeddingGuest empty;

small.swapWeddingGuests(empty);

assert(small.noGuests()); // swap works with empty list as well

assert(empty.guestCount() == 1);

assert(empty.invitedToTheWedding("Lonely", "Person")); // successfully swapped

groomsmen.swapWeddingGuests(bridesmaids); // swap them back for the rest of the test cases

WeddingGuest joined;

assert(!joinGuests(groomsmen, bridesmaids, joined)); // returns false because the empty string is in both with different values

assert(joined.guestCount() == 10); // excludes both empty strings: 8 + 4 - 2 = 10

bridesmaids.alterGuest("", "", -1);

assert(joinGuests(groomsmen, bridesmaids, joined)); // returns true because the empty string is in both with the same value now

assert(joined.guestCount() == 11); // excludes one empty strings: 8 + 4 - 1 = 11

groomsmen.crossGuestOff("", "");

bridesmaids.crossGuestOff("", "");

assert(joinGuests(groomsmen, bridesmaids, joined)); // returns true now because no duplicates

assert(joined.guestCount() == 10); // 7 + 3 = 10

assert(joinGuests(groomsmen, bridesmaids, groomsmen));

assert(groomsmen.guestCount() == 10); // can put result in one of the two lists, no aliasing error

assert(joinGuests(bridesmaids, bridesmaids, joined)); // all same, should return true

for (int n = 0; n < bridesmaids.guestCount(); n++) // checking that this copies bridesmaids into joined

{

string first1;

string last1;

int val1;

bridesmaids.verifyGuestOnTheList (n, first1, last1, val1);

string first2;

string last2;

int val2;

joined.verifyGuestOnTheList (n, first2, last2, val2);

assert(first1 == first2 // comparing each node individually

&& last1 == last2

&& val1 == val2);

}

WeddingGuest one;

one.inviteGuest("Pete", "Best", 3);

one.inviteGuest("John", "Lennon", 1);

one.inviteGuest("Paul", "McCartney", 2);

one.inviteGuest("Nakul", "Khambhati", 13);

WeddingGuest two;

two.inviteGuest("Pete", "Best", 6);

two.inviteGuest("George", "Harrison", 4);

two.inviteGuest("Ringo", "Starr", 5);

two.inviteGuest("Nakul", "Khambhati", 13);

WeddingGuest three;

assert(!joinGuests(one, two, three));

assert(!three.invitedToTheWedding("Pete", "Best")); // diff values so not added

assert(three.invitedToTheWedding("Nakul", "Khambhati")); // same value so added

assert(three.guestCount() == 5); // 4 + 4 - 2 (petes) - 1 (nakul)

WeddingGuest similarNames;

similarNames.inviteGuest("Nakul", "K", 35);

similarNames.inviteGuest("Nakul", "S", 38);

similarNames.inviteGuest("Khushi", "K", 44);

similarNames.inviteGuest("Khushi", "S", 45);

similarNames.inviteGuest("Arjun", "P", 0);

WeddingGuest result;

attestGuests("\*", "K", similarNames, result); // testing first wild card

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

assert(result.invitedToTheWedding("Nakul", "K"));

assert(result.invitedToTheWedding("Khushi", "K"));

attestGuests("Nakul", "\*", similarNames, result); // testing second wild card

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

assert(result.invitedToTheWedding("Nakul", "K"));

assert(result.invitedToTheWedding("Nakul", "S"));

attestGuests("\*", "\*", similarNames, result); // testing both wild cards

assert(result.guestCount() == 5);

attestGuests("Nakul", "K", similarNames, result); // testing specific

assert(result.guestCount() == 1);

assert(result.invitedToTheWedding("Nakul", "K"));

attestGuests("Arjun", "P", similarNames, result); // testing specific

assert(result.guestCount() == 1);

assert(result.invitedToTheWedding("Arjun", "P"));

attestGuests("Khushi", "P", similarNames, result); // testing specific

assert(result.noGuests());

assert(!result.invitedToTheWedding("Khushi", "P"));

attestGuests("\*", "\*", similarNames, similarNames);

assert(similarNames.guestCount() == 5); // aliasing error

attestGuests("Nakul", "\*", similarNames, similarNames);

assert(similarNames.guestCount() == 2); // aliasing error

cerr << "All test cases passed." << endl;