**Report Project 2 – Joonwon Lee**

1. **Description of the design of your implementation and why you chose it**

For this project, I just implemented a simple doubly linked list. I didn’t really see a significant benefit in a circular list as we weren’t necessarily limited in node count. Additionally, we were organizing the names by alphabetical order so I thought it would be wiser to just stick with a typical doubly linked list and traverse from the beginning to the end to find the right spot/name to either change values, insert, or delete. However, if we had a function like insertAtTail, I could have potentially set up a dummy node to avoid checking for edge cases ex) List is empty.

Picture of a typical doubly linked list

|  |  |  |
| --- | --- | --- |
| **Empty** | **One Node** | **Multiple Nodes** |
| Diagram  Description automatically generated |  |  |

1. **Brief description of notable obstacles you overcame**

Personally, I found inviteGuest to be the most challenging to implement. If we were to only consider the last name it would have been very easy. However, since we had to compare the last name and the firstname and insert the guest node accordingly the comparison cases that needed to be considered rapidly increased. For example, if the last name was equal to the m\_head’s lastname, if we were to only consider lastname we could just simply insert it to the very front and set it as the new node. However, because we took into consideration the first name as well, I had to traverse through every guest until it was either a nullpointer, the lastname changed, or the next node’s first name was alphabetically after the guest I wanted to insert. Though I didn’t really know where to go at first, by carefully thinking the different comparisons we would need to make at each scenario, whether that be last name or first name, I was successfully able to implement the inviteGuest function.

1. **Pseudocode for non-trivial algorithms**

|  |  |
| --- | --- |
| **inviteGuest** | First check if the guest is on the list. If it is return false.  Next check if the guest list is empty. If it is add a new node and assign a name and value to the new node.  Next check if the guest’s name we are inputting is alphabetically equal to or goes in front of the head node. Two cases to consider if this is true.   1. Last name is equal to the head’s last name. Then we need to compare the first names.  * If the input value’s first name goes alphabetically before the head’s first name simply make a new node, assign values, and make it the new head * If the input value’s first name goes alphabetically after the head node then go through the linked list until the last name is no longer the same, the first name that is already on the list goes alphabetically after the input first name, and until the last node. After we found the right spot to put the new guest in make a new node, assign values, and set the previous and next accordingly.  1. Last name goes alphabetically before the head’s last name. Then simply make a new node, assign values, and make it the new head   At this point we know that the guest belongs somewhere in the middle or end of the linked list.  Start from the beginning of the linked list and traverse through until the input last name alphabetically comes before or equals the last name of an existing node, or it reaches the end of the list.   1. If it reached the end of the list then make a new node, assign values, and set it as a tail. 2. If the last name is equal, continue to traverse until the input first name alphabetically comes before an existing node. 3. If the input last name is not equal but alphabetically comes before an existing node, put it before the existing node.   \*Increment m\_size by 1 when new guest is invited\* |
| **alterGuest** | Start from the head of the linked list and traverse until the input full name is equal  If it is equal, then make that full name no longer map to the value it currently maps to, but instead map to the parameter value.  If any change was made return true. If not return false. |
| **inviteOrAlter** | Using the function invited to the wedding, see if the guest is invited.  If the guest is invited, then use the alterGuest function to alter the value.  If the guest is not invited, then use the inviteGuest function to add a new guest to the list. |
| **crossGuestOff** | Start from the head of the linked list and traverse until the input full name is equal.   1. If it traversed through the whole list not having found a match, return false. 2. If the match was the head node, then set the next node of the current head as the new head. If the updated list is not empty set the previous of the new head as null. If the list is empty set the tail as null. 3. If the match was the tail node, then set the previous node of tail as the new tail. Then set the new tail’s next node as null. \*Don’t have to consider the case where list becomes empty as that will be taken care of 2). 4. Means we are deleting a node that is in the middle of the list. To do this set current’s previous node’s next as current node’s next. Then we set the current node’s next node’s previous node as the current node’s previous node.   After adjustments have been made we delete the guest that needs to be crossed off.  Decrement size by 1 and since we did delete a guest we return true. |
| **invitedToTheWedding** | Traverse through the linked list and if it finds an exact match return true. If not return false. |
| **matchInvitedGuest** | Traverse through the linked list and if it finds an exact match set value to the value in the list that that full name maps to. If a change was made to the parameter value that was passed in by reference return true. If not return false. |
| **verifyGuestOnTheList** | First check if the index is valid. If the index is less than 0 or greater than or equal to size of the linked list, return false.  At this point we know that the index is valid.  Go to the linked list’s index value and set firstName, lastName, value (variables that are passed in by reference) to the node at the index’s first name, last name, and value. |
| **swapWeddingGuests** | To swap the list just swap the pointer of the head and tail.  Since I decided to make a private “size” variable we need to also swap the size variables accordingly. |
| **joinGuests** | Have a bool variable that will save our result; make the default true  Have a variable for last name, first name and value, and value for comparison  Check for aliasing – If odOne and odTwo are referring to the same guest make odJoined a copy of odOne and return true.  First make an empty WeddingGuest list (call it temp for now) where we will add our guest to when they meet the conditions listed on spec  First loop through each guest in the first list  If the guest is also on the second list but the values are not the same make bool variable false  If the guest is also on the second list but the values are the same inviteGuest to temp  If the guest is not on the second list inviteGuest to temp  Now loop through each guest in the second list  If the guest is also on the first list but the values are not the same make bool variable false  If the guest is also on the first list but the values are the same inviteGuest to temp  If the guest is not on the first list inviteGuest to temp  Make odJoined to temp  Return the bool variable |
| **attestGuests** | Make a string variable of first and last name and make a GuestType value  Then make an empty WeddingGuest list (call it temp for now) where we will add our guest if they match the search condition  If both first name and last name where wild carded temp would be a copy of odOne.  Else  Loop through the guest list  If there was an exact match for full name – add the guest to the list  Else if the first name was wild carded but the last name matched – add the guest to the list  Else if the last name was wild carded but the first name matched – add the guest to the list  Make the odResult be temp |

**4)**

#include "WeddingGuest.h"

#include <string>

#include <iostream>

#include <cassert>

using namespace std;

void test() {

/////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////

// Change Typedef to std::string

/////////////////////////////////////////////////////////////////

/////////////////////////////////////////////////////////////////

// // // default constructor

// WeddingGuest lal;

// // For an empty list:

// assert(lal.guestCount() == 0); // test size

// assert(lal.noGuests()); // test empty

// assert(!lal.crossGuestOff("Malik", "Monk")); // nothing to erase

// assert(!lal.alterGuest("Edwin", "Ambrosio", "Hi")); //Nothing to alter

// assert(lal.inviteOrAlter("Edwin", "Ambrosio", "Hi")); //We invite the professor as the list is empty and therefore returns true

// assert(lal.guestCount() == 1); //see if professor was successfully added into the list

// assert(lal.crossGuestOff("Edwin", "Ambrosio")); //Cross professor off from the list as we dont want him as our guest

// assert(lal.guestCount() == 0); //see if we successfully crossed the professor off

// assert(lal.noGuests()); //verify that the list is empty

// assert(lal.inviteGuest("David", "Smallberg", "Bye")); //Invite Prof Smallberg to the wedding as he is not on the list

// assert(lal.inviteGuest("Carey", "Nachenberg", "Nice")); //Invite Prof Nachenberg to the wedding as he is not on the list

// assert(lal.inviteGuest("Paul", "Eggert", "What")); //Invite Prof Eggert to the wedding as he is not on the list

// string firstName, lastName, value;

// assert(lal.inviteGuest("Bob", "Allan", "First")); //Invite Bob Allan to the wedding to see if he gets correctly placed at the front of the list

// assert(lal.verifyGuestOnTheList(0, firstName, lastName, value)); //Verify that index 0 is a valid index and save whatever the name and values are in that index (hopefully it is Bob Allan with value First)

// assert(firstName == "Bob" && lastName == "Allan" && value == "First"); //Verify that the first name on the list is Bob Allan with value First

// assert(lal.inviteGuest("Ben", "Ambrosio", "Second")); //Invite Ben Ambrosio to the wedding to see if he gets correctly placed as the second guest on the list in front of Edwin Ambrosio (Same last name but first name comes alphabetically first)

// assert(lal.verifyGuestOnTheList(1, firstName, lastName, value)); //Verify that index 1 is a valid index and save whatever the name and values are in that index (hopefully it is Ben Ambrosio with value Second)

// assert(firstName == "Ben" && lastName == "Ambrosio" && value == "Second"); //Verify that the second name on the list is Ben Ambrosio with value Second

// assert(lal.inviteGuest("Junior", "Nachenberg", "Android")); //Invite Junior Nachenberg to see if he gets placed as the sixth guest on the list and after Carey Nachenberg (Same last name but first name comes alphabetically later)

// assert(lal.inviteGuest("Amy", "Nachenberg", "No")); //Invite Amy Nachenberg to see if she gets placed as the fourth guest on the list and before Carey Nachenberg (Same last name but first name comes alphabetically first)

// assert(lal.verifyGuestOnTheList(3, firstName, lastName, value)); //Verify that index 3 is a valid index and save whatever the name and values are in that index (hopefully it is Amy Nachenberg with value No)

// assert(firstName == "Amy" && lastName == "Nachenberg" && value == "No"); //Verify that the fourth name on the list is Amy Nachenberg with value No

// assert(lal.verifyGuestOnTheList(2, firstName, lastName, value)); //Verify that index 2 is a valid index and save whatever the name and values are in that index (hopefully it is Paul Eggert with value What)

// assert(firstName == "Paul" && lastName == "Eggert" && value == "What"); //Verify that the third name on the list is Paul Eggert with value What

// assert(lal.verifyGuestOnTheList(4, firstName, lastName, value)); //Verify that index 4 is a valid index and save whatever the name and values are in that index (hopefully it is Carey Nachenberg with value Nice)

// assert(firstName == "Carey" && lastName == "Nachenberg" && value == "Nice"); //Verify that the fifth name on the list is Carey Nachenberg

// lal.alterGuest("Paul", "Eggert", "CS33"); //Alter Paul Eggert to say CS33 (we know Paul Eggert is third on the list (index 2))

// assert(lal.verifyGuestOnTheList(2, firstName, lastName, value)); //Verify that index 2 is a valid index and save whatever the name and values are in that index (hopefully it is Paul Eggert with value CS33)

// assert(firstName == "Paul" && lastName == "Eggert" && value == "CS33"); //Verify that the third name on the list is Paul Eggert with value CS33

// lal.alterGuest("Carey", "Nachenberg", "Florentine Deception"); //Alter Carey Nachenberg to say Florentine Deception (we know Carey Nachenberg is fifth on the list (index 4))

// assert(lal.verifyGuestOnTheList(4, firstName, lastName, value)); //Verify that index 4 is a valid index and save whatever the name and values are in that index (hopefully it is Carey Nachenberg with value Florentine Deception)

// assert(firstName == "Carey" && lastName == "Nachenberg" && value == "Florentine Deception"); //Verify that the fifth name on the list is Carey Nachenberg with value Florentine Deception

// //Since we tested the case when name wasn't on the list, we can test the case when name is on the list for inviteOrAlter

// assert(lal.inviteOrAlter("Ben", "Ambrosio", "CS32")); //We alter what Edwin Ambrosio is saying to CS32 (we know Edwin Ambrosio is second on the list (index 1))

// assert(lal.verifyGuestOnTheList(1, firstName, lastName, value)); //Verify that index 1 is a valid index and save whatever the name and values are in that index (hopefully it is Ben Ambrosio with value CS32)

// assert(firstName == "Ben" && lastName == "Ambrosio" && value == "CS32"); //Verify that the second name on the list is Ben Ambrosio with value CS32

// assert(lal.inviteOrAlter("Paul", "Eggert", "CS31")); //We alter what Paul Eggert is saying to CS31 (we know Paul Eggert is third on the list (index 2))

// assert(lal.verifyGuestOnTheList(2, firstName, lastName, value)); //Verify that index 2 is a valid index and save whatever the name and values are in that index (hopefully it is Paul Eggert with value CS31)

// assert(firstName == "Paul" && lastName == "Eggert" && value == "CS31"); //Verify that the third name on the list is Paul Eggert with value CS31

// //Now we test out the invitedToWedding function

// assert(lal.invitedToTheWedding("Bob", "Allan")); //We verify that Bob Allan is invited to the wedding (we know Bob Allan is first on the list (index 0))

// assert(lal.invitedToTheWedding("Ben", "Ambrosio")); //We verify that Ben Ambrosio is invited to the wedding (we know Ben Ambrosio is second on the list (index 1))

// assert(lal.invitedToTheWedding("Paul", "Eggert")); //We verify that Paul Eggert is invited to the wedding (we know Paul Eggert is third on the list (index 2))

// assert(lal.invitedToTheWedding("Amy", "Nachenberg")); //We verify that Amy Nachenberg is invited to the wedding (we know Amy Nachenberg is fourth on the list (index 3))

// assert(lal.invitedToTheWedding("Carey", "Nachenberg")); //We verify that Carey Nachenberg is invited to the wedding (we know Carey Nachenberg is fifth on the list (index 4))

// assert(!lal.invitedToTheWedding("Edwin", "Ambrosio")); //We verify that Edwin Ambrosio is not invited to the wedding

// assert(!lal.invitedToTheWedding("Donald", "Trump")); //We verify that Donald Trump is not invited to the wedding

// assert(!lal.invitedToTheWedding("Joe", "Biden")); //We verify that Donald Trump is not invited to the wedding

// //We can cross guest off and check if both our cross guest off and invitedToWedding function works properly

// assert(lal.crossGuestOff("Bob", "Allan")); //We cross off Bob Allan (we know Bob Allan is first on the list (index 0))

// assert(!lal.invitedToTheWedding("Bob", "Allan")); //We verify that Bob Allan is not invited to the wedding as we crossed him off

// assert(lal.crossGuestOff("David", "Smallberg")); //We verify that David Smallberg is invited to the wedding (we know David Smallberg is last on the list (index 6))

// assert(!lal.invitedToTheWedding("David", "Smallberg")); //We verify that David Smallberg is not invited to the wedding as we crossed him off

// assert(!lal.crossGuestOff("Donald", "Trump")); //We verify that we can't cross Donald Trump off because he was not invited in the first place

// assert(!lal.crossGuestOff("Joe", "Biden")); //We verify that we can't cross Joe Biden off because he was not invited in the first place

// //We can check if our matchInvitedGuest works as expected

// assert(lal.matchInvitedGuest("Ben", "Ambrosio", value)); //We know Ben Ambrosio is still on the guest list and hopefully the value is changed to CS32

// assert(value == "CS32"); //We verify that the value is CS32

// assert(!lal.matchInvitedGuest("David", "Smallberg", value)); //We know David Smallberg is not on the guest list and hopefully the value is unchanged

// assert(value == "CS32"); //We verify that the value is still CS32

// assert(lal.matchInvitedGuest("Paul", "Eggert", value)); //We know Paul Eggert is still on the guest list and hopefully the value is changed to CS31

// assert(value == "CS31"); //We verify that the value is CS31

// assert(!lal.matchInvitedGuest("Donald", "Trump", value)); //We know Donald Trump is not on the guest list and hopefully the value is unchanged

// assert(value == "CS31"); //We verify that the value is still CS31

// assert(lal.inviteGuest("Donald", "Trump", "Trump Tower")); //We invite Donald Trump to the wedding (we know Donald Trump is not on the guest list)

// assert(lal.matchInvitedGuest("Donald", "Trump", value)); //We know Donald Trump is on the guest list and hopefully the value is changed to Trump Tower

// assert(value == "Trump Tower"); //We verify that the value is Trump Tower

// assert(lal.inviteOrAlter("Donald", "Trump", "Weird Hair")); //We alter what Donald Trump is saying to Weird Hair (we know Donald Trump is on the guest list)

// assert(lal.matchInvitedGuest("Donald", "Trump", value)); //We know Donald Trump is on the guest list and hopefully the value is changed to Weird Hair

// assert(value == "Weird Hair"); //We verify that the value is Weird Hair

// WeddingGuest lal2;

// lal2 .inviteGuest("Kyrie", "Irving", "Kyrie Irving"); //We invite Kyrie Irving to the wedding (we know Kyrie Irving is not on the guest list)

// lal2 .inviteGuest("Klay", "Thompson", "Klay Thompson"); //We invite Klay Thompson to the wedding (we know Klay Thompson is not on the guest list)

// lal2 .inviteGuest("Draymond", "Green", "Draymond Green"); //We invite Draymond Green to the wedding (we know Draymond Green is not on the guest list)

// lal2 .inviteGuest("Stephen", "Curry", "Stephen Curry"); //We invite Stephen Curry to the wedding (we know Stephen Curry is not on the guest list)

// lal2 .inviteGuest("Kawhi", "Leonard", "Kawhi Leonard"); //We invite Kawhi Leonard to the wedding (we know Kawhi Leonard is not on the guest list)

// //We can check if our swapWeddingGuests works as expected

// lal.swapWeddingGuests(lal2);

// // for (int i = 0; i < lal.guestCount(); i++) {

// // string firstName;

// // string lastName;

// // string value;

// // lal.verifyGuestOnTheList(i, firstName, lastName, value);

// // cout << "First Name: " << firstName << " Last Name: " << lastName << " Value: " << value << endl;

// // }

// //Should print out:

// // First Name: Stephen Last Name: Curry Value: Stephen Curry

// // First Name: Draymond Last Name: Green Value: Draymond Green

// // First Name: Kyrie Last Name: Irving Value: Kyrie Irving

// // First Name: Kawhi Last Name: Leonard Value: Kawhi Leonard

// // First Name: Klay Last Name: Thompson Value: Klay Thompson

// //Comment out the for loop only after verification is complete

// // //We can check if our copy constructor works as expected

// WeddingGuest lal3(lal);

// // for (int i = 0; i < lal3.guestCount(); i++) {

// // string firstName;

// // string lastName;

// // string value;

// // lal3.verifyGuestOnTheList(i, firstName, lastName, value);

// // cout << "First Name: " << firstName << " Last Name: " << lastName << " Value: " << value << endl;

// // }

// //Should print out:

// // First Name: Stephen Last Name: Curry Value: Stephen Curry

// // First Name: Draymond Last Name: Green Value: Draymond Green

// // First Name: Kyrie Last Name: Irving Value: Kyrie Irving

// // First Name: Kawhi Last Name: Leonard Value: Kawhi Leonard

// // First Name: Klay Last Name: Thompson Value: Klay Thompson

// //Commment out the for loop after verification is complete

// // //We can check if our assignment operator works as expected

// WeddingGuest lal4;

// lal4.inviteGuest("Harry", "Potter", "Harry Potter"); //We invite Harry Potter to the wedding (we know Harry Potter is not on the guest list)

// lal4.inviteGuest("Hermione", "Granger", "Hermione Granger"); //We invite Hermione Granger to the wedding (we know Hermione Granger is not on the guest list)

// lal4.inviteGuest("Ron", "Weasley", "Ron Weasley"); //We invite Ron Weasley to the wedding (we know Ron Weasley is not on the guest list)

// lal = lal4;

// // for (int i = 0; i < lal.guestCount(); i++) {

// // string firstName;

// // string lastName;

// // string value;

// // lal.verifyGuestOnTheList(i, firstName, lastName, value);

// // cout << "First Name: " << firstName << " Last Name: " << lastName << " Value: " << value << endl;

// // }

// //Should print out:

// // First Name: Hermione Last Name: Granger Value: Hermione Granger

// // First Name: Harry Last Name: Potter Value: Harry Potter

// // First Name: Ron Last Name: Weasley Value: Ron Weasley

// //Comment out the for loop after verification is complete

// // //See if assignment operator correctly deals with aliasing

// lal4 = lal4;

// // for (int i = 0; i < lal.guestCount(); i++) {

// // string firstName;

// // string lastName;

// // string value;

// // lal4.verifyGuestOnTheList(i, firstName, lastName, value);

// // cout << "First Name: " << firstName << " Last Name: " << lastName << " Value: " << value << endl;

// // }

// //Should print out:

// // First Name: Hermione Last Name: Granger Value: Hermione Granger

// // First Name: Harry Last Name: Potter Value: Harry Potter

// // First Name: Ron Last Name: Weasley Value: Ron Weasley

// //Comment out the for loop after verification is complete

// //Test if our joinedGuest function works properly and only have guest names that meet the condition listed on the spec

// //Tests the case where if there exists a full name that appears in both odOne and odTwo with the same corresponding value in both, then odJoined must contain

// //an element with that full name and value in this case it is Lebron James 23

// //Also we can make sure that if a full name appears in exactly one of odOne and odTwo then odJoined must contain an element consisting of that full name and corresponding value

// //in this case it is Anthony Davis 3, Malik Monk 11, and Russell Westbrook 0

// WeddingGuest first;

// first.inviteGuest("Anthony", "Davis", "3");

// first.inviteGuest("Lebron", "James", "23");

// first.inviteGuest("Malik", "Monk", "11");

// WeddingGuest second;

// second.inviteGuest("Lebron", "James", "23");

// second.inviteGuest("Russell", "Westbrook", "0");

// WeddingGuest joined;

// joined.inviteGuest("Donald", "Trump", "Trump Tower"); //Donald Trump should not be on the list according to the spec

// joined.inviteGuest("Joe", "Biden", "President"); //Joe Biden should not be on the list according to the spec

// joined.inviteGuest("George", "Washington", "First President"); //Anthony Davis should be on the list according to the spec

// assert(joinGuests(first, second, joined)); //should return true because there isn't any name that appears in both list but with different values

// // for(int i = 0; i < joined.guestCount(); i++) {

// // string firstName;

// // string lastName;

// // string value;

// // joined.verifyGuestOnTheList(i, firstName, lastName, value);

// // cout << "First Name: " << firstName << " Last Name: " << lastName << " Value: " << value << endl;

// // }

// //Should print out:

// // First Name: Anthony Last Name: Davis Value: 3

// // First Name: Lebron Last Name: James Value: 23

// // First Name: Malik Last Name: Monk Value: 11

// // First Name: Russell Last Name: Westbrook Value: 0

// //Test the case where if there exists a full name that appears in both odOne and odTwo but with different corresponding values, then this function returns false but still

// //only have the guest names that meet the condition listed on the spec

// first.inviteGuest("Kawhi", "Leonard", "2");

// second.inviteGuest("Kawhi", "Leonard", "3");

// assert(!joinGuests(first, second, joined)); //should return false because there is a Kawhi appears in both list but with different values

// // for(int i = 0; i < joined.guestCount(); i++) {

// // string firstName;

// // string lastName;

// // string value;

// // joined.verifyGuestOnTheList(i, firstName, lastName, value);

// // cout << "First Name: " << firstName << " Last Name: " << lastName << " Value: " << value << endl;

// // }

// //Should print out only:

// // First Name: Anthony Last Name: Davis Value: 3

// // First Name: Lebron Last Name: James Value: 23

// // First Name: Malik Last Name: Monk Value: 11

// // First Name: Russell Last Name: Westbrook Value: 0

// //Comment out the for loop after verification is complete

// //This part is to check aliasing

// assert(joinGuests(first, first, joined));

// // for(int i = 0; i < joined.guestCount(); i++) {

// // string firstName;

// // string lastName;

// // string value;

// // joined.verifyGuestOnTheList(i, firstName, lastName, value);

// // cout << "First Name: " << firstName << " Last Name: " << lastName << " Value: " << value << endl;

// // }

// //should print out first as joined would simply be a copy of first:

// // First Name: Anthony Last Name: Davis Value: 3

// // First Name: Lebron Last Name: James Value: 23

// // First Name: Kawhi Last Name: Leonard Value: 2

// // First Name: Malik Last Name: Monk Value: 11

// assert(!joinGuests(first, second, first)); //should return false because there a Kawhi Leonard appears in both list but with different values

// // for(int i = 0; i < first.guestCount(); i++) {

// // string firstName;

// // string lastName;

// // string value;

// // first.verifyGuestOnTheList(i, firstName, lastName, value);

// // cout << "First Name: " << firstName << " Last Name: " << lastName << " Value: " << value << endl;

// // }

// //Putting first in result shouldn't affect anything:

// // First Name: Anthony Last Name: Davis Value: 3

// // First Name: Lebron Last Name: James Value: 23

// // First Name: Malik Last Name: Monk Value: 11

// // First Name: Russell Last Name: Westbrook Value: 0

// //The joined list must be empty since both list are empty

// WeddingGuest empty1;

// WeddingGuest empty2;

// joinGuests(empty1, empty2, joined);

// // for(int i = 0; i < joined.guestCount(); i++) {

// // string firstName;

// // string lastName;

// // string value;

// // joined.verifyGuestOnTheList(i, firstName, lastName, value);

// // cout << "First Name: " << firstName << " Last Name: " << lastName << " Value: " << value << endl;

// // }

// //Shouldnt print anything out

// //Comment out the for loop after verification is complete

// WeddingGuest test;

// WeddingGuest result;

// // Case when we look at the second parameter only

// test.inviteGuest("Steve", "A", "Apple");

// test.inviteGuest("Tim", "A", "Apple");

// test.inviteGuest("Elon", "T", "Tesla");

// test.inviteGuest("Bill", "M", "Microsoft");

// test.inviteGuest("Mark", "M", "Meta");

// test.inviteGuest("Jeff", "A", "Amazon");

// test.inviteGuest("Steve", "Speilberg", "Movie");

// result.inviteGuest("John", "Ma", "A");

// result.inviteGuest("Paul", "McCartney", "M");

// result.inviteGuest("George", "Harrison", "H");

// result.inviteGuest("Ringo", "Starr", "S");

// attestGuests("\*", "A", test, result);

// // for (int i = 0; i < result.guestCount(); i++) {

// // string firstName;

// // string lastName;

// // string value;

// // result.verifyGuestOnTheList(i, firstName, lastName, value);

// // cout << "First Name: " << firstName << " Company First Letter: " << lastName << " Company Full Name: " << value << endl;

// // }

// // result should print out:

// // First Name: Jeff Company First Letter: A Company Full Name: Amazon

// // First Name: Steve Company First Letter: A Company Full Name: Apple

// // First Name: Tim Company First Letter: A Company Full Name: Apple

// //Comment out the for loop after verification is complete

// //Case when we look at the first parameter only

// attestGuests("Steve", "\*", test, result);

// // for (int i = 0; i < result.guestCount(); i++) {

// // string firstName;

// // string lastName;

// // string value;

// // result.verifyGuestOnTheList(i, firstName, lastName, value);

// // cout << "First Name: " << firstName << " Company First Letter: " << lastName << " Company Full Name: " << value << endl;

// // }

// // result should only print out:

// // First Name: Steve Company First Letter: A Company Full Name: Apple

// // First Name: Steve Company First Letter: Speilberg Company Full Name: Movie

// // Comment out the for loop after verification is complete

// //Case when we look at both the first and second parameter

// attestGuests("Steve", "A", test, result);

// // for (int i = 0; i < result.guestCount(); i++) {

// // string firstName;

// // string lastName;

// // string value;

// // result.verifyGuestOnTheList(i, firstName, lastName, value);

// // cout << "First Name: " << firstName << " Company First Letter: " << lastName << " Company Full Name: " << value << endl;

// // }

// // result should only print out:

// // First Name: Steve Company First Letter: A Company Full Name: Apple

// attestGuests("\*", "\*", test, result);

// // for (int i = 0; i < result.guestCount(); i++) {

// // string firstName;

// // string lastName;

// // string value;

// // result.verifyGuestOnTheList(i, firstName, lastName, value);

// // cout << "First Name: " << firstName << " Company First Letter: " << lastName << " Company Full Name: " << value << endl;

// // }

// // result should be just a copy of test

// // Comment out the for loop after verification is complete

// //Check if function behaves correctly result is also passed in as odOne

// attestGuests("\*", "A", test, test);

// // for (int i = 0; i < test.guestCount(); i++) {

// // string firstName;

// // string lastName;

// // string value;

// // test.verifyGuestOnTheList(i, firstName, lastName, value);

// // cout << "First Name: " << firstName << " Company First Letter: " << lastName << " Company Full Name: " << value << endl;

// // }

// // test should print out:

// // First Name: Jeff Company First Letter: A Company Full Name: Amazon

// // First Name: Steve Company First Letter: A Company Full Name: Apple

// // First Name: Tim Company First Letter: A Company Full Name: Apple

// ///////////////////////////////////////////////////////////////////

// ///////////////////////////////////////////////////////////////////

// //Change Typedef to int and comment out the previous test cases

// ///////////////////////////////////////////////////////////////////

// ///////////////////////////////////////////////////////////////////

// WeddingGuest intTest;

// assert(intTest.guestCount() == 0); // test size and verify that it is empty

// assert(intTest.noGuests()); // test empty

// assert(!intTest.crossGuestOff("Malik", "Monk")); // nothing to erase (possible edge case)

// assert(!intTest.alterGuest("Edwin", "Ambrosio", 1)); //Nothing to alter

// assert(intTest.inviteOrAlter("Edwin", "Ambrosio", 1)); //We invite the professor as the list is empty and therefore returns true

// assert(intTest.guestCount() == 1); //see if professor was successfully added into the list

// assert(intTest.crossGuestOff("Edwin", "Ambrosio")); //Cross professor off from the list as we dont want him as our guest

// assert(intTest.guestCount() == 0); //see if we successfully crossed the professor off

// assert(intTest.noGuests()); //verify that the list is empty

// assert(intTest.inviteGuest("David", "Smallberg", 2)); //Invite Prof Smallberg to the wedding as he is not on the list

// assert(intTest.inviteGuest("Carey", "Nachenberg", 3)); //Invite Prof Nachenberg to the wedding as he is not on the list

// assert(intTest.inviteGuest("Paul", "Eggert", 4)); //Invite Prof Eggert to the wedding as he is not on the list

// // for (int i = 0; i < intTest.guestCount(); i++) {

// // string firstName;

// // string lastName;

// // int value;

// // intTest.verifyGuestOnTheList(i, firstName, lastName, value);

// // cout << "First Name: " << firstName << " Last Name: " << lastName << " Integer Value: " << value << endl;

// // }

// // Should print out:

// // First Name: Paul Last Name: Eggert Integer Value: 4

// // First Name: Carey Last Name: Nachenberg Integer Value: 3

// // First Name: David Last Name: Smallberg Integer Value: 2

// string firstName, lastName;

// int valueInt;

// assert(intTest.inviteGuest("Bob", "Allan", 5)); //Invite Bob Allan to the wedding to see if he gets correctly placed at the front of the list

// assert(intTest.verifyGuestOnTheList(0, firstName, lastName, valueInt)); //Verify that index 0 is a valid index and save whatever the name and values are in that index (hopefully it is Bob Allan with value 5)

// assert(firstName == "Bob" && lastName == "Allan" && valueInt == 5); //Verify that the first name on the list is Bob Allan with value 5

// assert(intTest.inviteGuest("Ben", "Ambrosio", 6)); //Invite Ben Ambrosio to the wedding to see if he gets correctly placed as the second guest on the list in front of Edwin Ambrosio (Same last name but first name comes alphabetically first)

// assert(intTest.verifyGuestOnTheList(1, firstName, lastName, valueInt)); //Verify that index 1 is a valid index and save whatever the name and values are in that index (hopefully it is Ben Ambrosio with value 6)

// assert(firstName == "Ben" && lastName == "Ambrosio" && valueInt == 6); //Verify that the second name on the list is Ben Ambrosio with value 6

// assert(intTest.inviteGuest("Junior", "Nachenberg", 7)); //Invite Junior Nachenberg to see if he gets placed as the sixth guest on the list and after Carey Nachenberg (Same last name but first name comes alphabetically later)

// assert(intTest.inviteGuest("Amy", "Nachenberg", 8)); //Invite Amy Nachenberg to see if she gets placed as the fourth guest on the list and before Carey Nachenberg (Same last name but first name comes alphabetically first)

// assert(intTest.verifyGuestOnTheList(3, firstName, lastName, valueInt)); //Verify that index 3 is a valid index and save whatever the name and values are in that index (hopefully it is Amy Nachenberg with value 8)

// assert(firstName == "Amy" && lastName == "Nachenberg" && valueInt == 8); //Verify that the fourth name on the list is Amy Nachenberg with value 8

// assert(intTest.verifyGuestOnTheList(2, firstName, lastName, valueInt)); //Verify that index 2 is a valid index and save whatever the name and values are in that index (hopefully it is Paul Eggert with value 4)

// assert(firstName == "Paul" && lastName == "Eggert" && valueInt == 4); //Verify that the third name on the list is Paul Eggert with value 4

// assert(intTest.verifyGuestOnTheList(4, firstName, lastName, valueInt)); //Verify that index 4 is a valid index and save whatever the name and values are in that index (hopefully it is Carey Nachenberg with value 3)

// assert(firstName == "Carey" && lastName == "Nachenberg" && valueInt == 3); //Verify that the fifth name on the list is Carey Nachenberg with value 3

// assert(intTest.alterGuest("Ben", "Ambrosio", 123)); //See if our alterguest function changes the Ben Ambrosio value correctly to 123

// assert(intTest.verifyGuestOnTheList(1, firstName, lastName, valueInt));

// assert(firstName == "Ben" && lastName == "Ambrosio" && valueInt == 123);

// assert(!intTest.alterGuest("Donald", "Trump", 5)); //verify that alterGuest function returns false when there is no guest that was passed in through parameter

// assert(intTest.inviteOrAlter("Adam", "Levine", 5)); //verify that since Adam Levine is not on the list, he gets added onto the list with value 5

// assert(intTest.inviteOrAlter("Ben", "Ambrosio", 11)); //verify that since Ben Ambrosio is on the list, his value gets changed to 11

// assert(intTest.crossGuestOff("Bob", "Allan")); //verify that crossGuestOff function can cross off the guest from the list that is at the front

// assert(intTest.crossGuestOff("Amy", "Nachenberg")); //verify that crossGuestOff function can cross off the guest from the list that is in the middle

// assert(intTest.crossGuestOff("David", "Smallberg")); //verify that crossGuestOff function can cross off the guest from the list that is at the end

// assert(!intTest.crossGuestOff("Joe", "Biden)")); //verify that crossGuest off function can't cross off the guest that is not on the list

// assert(intTest.invitedToTheWedding("Carey", "Nachenberg")); //verify that invitedToTheWedding function can return true when the guest is on the middle of the list

// assert(!intTest.invitedToTheWedding("Joe", "Biden")); //verify that invitedToTheWedding function can return false when the guest is not on the list

// assert(intTest.invitedToTheWedding("Ben", "Ambrosio")); //verify that invitedToTheWedding function can return true when the guest is first on the list

// assert(intTest.invitedToTheWedding("Junior", "Nachenberg")); //verify that invitedToTheWedding function can return true when the guest is at the end of the list

// assert(intTest.matchInvitedGuest("Ben", "Ambrosio", valueInt)); //verify that matchInvitedGuest function can return true when the guest is on the list and map the valueInt to 11 which Ben Ambrosio is mapped to

// assert(valueInt == 11);

// assert(!intTest.matchInvitedGuest("Joe", "Biden", valueInt)); //verify that matchInvitedGuest function can return false when the guest is not on the list

// assert(valueInt == 11); //left unchanged

// assert(intTest.matchInvitedGuest("Adam", "Levine", valueInt)); //verify that matchInvitedGuest function can return true when the guest is on the list and map the valueInt to 5 which Adam Levine is mapped to

// assert(valueInt == 5);

// WeddingGuest intTest1;

// intTest1.inviteGuest("Steve", "Jobs", 7);

// intTest1.inviteGuest("Bill", "Gates", 8);

// intTest1.inviteGuest("Elon", "Musk", 9);

// intTest1.inviteGuest("Tim", "Cook", 10);

// intTest1.inviteGuest("Jeff", "Bezos", 11);

// intTest1.inviteGuest("Mark", "Zuckerberg", 12);

// intTest.swapWeddingGuests(intTest1);

// // for (int i = 0; i < intTest.guestCount(); i++) {

// // string firstName;

// // string lastName;

// // int value;

// // intTest.verifyGuestOnTheList(i, firstName, lastName, value);

// // cout << "First Name: " << firstName << " Last Name: " << lastName << " Integer Value: " << value << endl;

// // }

// // Swapped intTest guests with intTest1

// // intTest should print:

// // First Name: Jeff Last Name: Bezos Integer Value: 11

// // First Name: Tim Last Name: Cook Integer Value: 10

// // First Name: Bill Last Name: Gates Integer Value: 8

// // First Name: Steve Last Name: Jobs Integer Value: 7

// // First Name: Elon Last Name: Musk Integer Value: 9

// // First Name: Mark Last Name: Zuckerberg Integer Value: 12

// // Comment out for loop

// WeddingGuest intTest2; //Edge case when list is empty

// assert(intTest2.noGuests()); //should return true because there is no guest

// assert(intTest2. guestCount() == 0); //should return 0 because there is no guest

// assert(intTest2.inviteGuest("Bob", "Allan", 5)); //should return true because Bob Allan was added to the list

// assert(intTest2.crossGuestOff("Bob", "Allan")); //should return true because Bob Allan was removed from the list

// assert(!intTest2.crossGuestOff("Bob", "Allan")); //should return false because Bob Allan is not on the list rather its empty

// assert(!intTest2.alterGuest("Bob", "Allan", 5)); //should return false because Bob Allan is not on the list rather its empty

// assert(!intTest2.invitedToTheWedding("Bob", "Allan")); //verify that the list is empty and therefore there is no invited guest for this wedding therefore function should return false

// assert(!intTest2.matchInvitedGuest("Bob", "Allan", valueInt)); //Since the list is empty, this function should return false

// assert(!intTest2.verifyGuestOnTheList(0, firstName, lastName, valueInt)); //Since the list is empty, this function should return false

// intTest.swapWeddingGuests(intTest2); //swap intTest with intTest2

// assert(intTest.noGuests()); //should return true because there is no guest in intTest now

// WeddingGuest intOne, intTwo, intResult;

// intOne.inviteGuest("Steve", "Jobs", 7);

// intOne.inviteGuest("Bill", "Gates", 8);

// intOne.inviteGuest("Elon", "Musk", 9);

// intOne.inviteGuest("Tim", "Cook", 10);

// intOne.inviteGuest("Jeff", "Bezos", 11);

// intOne.inviteGuest("Mark", "Zuckerberg", 12);

// intOne.inviteGuest("Steve", "Speilberg", 13);

// intOne.inviteGuest("Melinda", "Gates", 14);

// intTwo.inviteGuest("Steve", "Jobs", 8);

// intTwo.inviteGuest("Bill", "Gates", 8);

// intTwo.inviteGuest("Elon", "Musk", 9);

// intResult.inviteGuest("Mike", "Tyson", 9);

// intResult.inviteGuest("Kawhi", "Leonard", 10);

// joinGuests(intOne, intTwo, intResult);

// // for (int i = 0; i < intResult.guestCount(); i++) {

// // string firstName;

// // string lastName;

// // int value;

// // intResult.verifyGuestOnTheList(i, firstName, lastName, value);

// // cout << "First Name: " << firstName << " Last Name: " << lastName << " Integer Value: " << value << endl;

// // }

// // Verify that Steve Jobs, Mike Tyson, Kawhi Leonard is not on the list and it should print:

// // First Name: Jeff Last Name: Bezos Integer Value: 11

// // First Name: Tim Last Name: Cook Integer Value: 10

// // First Name: Bill Last Name: Gates Integer Value: 8

// // First Name: Melinda Last Name: Gates Integer Value: 14

// // First Name: Elon Last Name: Musk Integer Value: 9

// // First Name: Steve Last Name: Speilberg Integer Value: 13

// // First Name: Mark Last Name: Zuckerberg Integer Value: 12

// //Check for aliasing

// assert(joinGuests(intOne, intOne, intResult)); //int result should be a copy of intOne

// //intResult should print:

// // First Name: Jeff Last Name: Bezos Integer Value: 11

// // First Name: Tim Last Name: Cook Integer Value: 10

// // First Name: Bill Last Name: Gates Integer Value: 8

// // First Name: Melinda Last Name: Gates Integer Value: 14

// // First Name: Steve Last Name: Jobs Integer Value: 7

// // First Name: Elon Last Name: Musk Integer Value: 9

// // First Name: Steve Last Name: Speilberg Integer Value: 13

// // First Name: Mark Last Name: Zuckerberg Integer Value: 12

// WeddingGuest intTemp = intOne;

// assert(!joinGuests(intTemp, intTwo, intTemp)); //Should return false because Steve Jobs is in both list but have different values;

// //intTemp should print:

// // First Name: Jeff Last Name: Bezos Integer Value: 11

// // First Name: Tim Last Name: Cook Integer Value: 10

// // First Name: Bill Last Name: Gates Integer Value: 8

// // First Name: Melinda Last Name: Gates Integer Value: 14

// // First Name: Elon Last Name: Musk Integer Value: 9

// // First Name: Steve Last Name: Speilberg Integer Value: 13

// // First Name: Mark Last Name: Zuckerberg Integer Value: 12

// attestGuests("Steve", "\*", intOne, intResult);

// // intResult Should print out:

// // First Name: Steve Last Name: Jobs Integer Value: 7

// // First Name: Steve Last Name: Speilberg Integer Value: 13

// attestGuests("\*", "Gates", intOne, intResult);

// // // intResult Should print out:

// // // First Name: Bill Last Name: Gates Integer Value: 8

// // // First Name: Melinda Last Name: Gates Integer Value: 14

// // attestGuests("\*", "\*", intOne, intResult);

// // // intResult should be a copy of intOne

// // attestGuests("", "", intOne, intResult);

// // // intResult should be empty

// // //Check if function works correctly when faced aliasing

// attestGuests("\*", "Gates", intOne, intOne);

// for (int i = 0; i < intOne.guestCount(); i++) {

// string firstName;

// string lastName;

// int value;

// intOne.verifyGuestOnTheList(i, firstName, lastName, value);

// cout << "First Name: " << firstName << " Company First Letter: " << lastName << " Int Value: " << value << endl;

// }

// //should print:

// // First Name: Bill Company First Letter: Gates Int Value: 8

// // First Name: Melinda Company First Letter: Gates Int Value: 14

///////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////

//Change Typedef to double and comment out the previous test cases

///////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////

// WeddingGuest doubleTest;

// double valueDouble;

// assert(doubleTest.noGuests()); //should return true because there is no guest

// assert(doubleTest. guestCount() == 0); //should return 0 because there is no guests

// assert(!doubleTest.crossGuestOff("Bob", "Allan")); //should return false because Bob Allan is not on the list and therefore it cant be removed from the list

// assert(!doubleTest.alterGuest("Bob", "Allan", 5)); //should return false because Bob Allan is not on the list and therefore it cant be altered

// assert(!doubleTest.invitedToTheWedding("Bob", "Allan")); //should return false because Bob Allan is not on the list therefore not invited

// assert(!doubleTest.matchInvitedGuest("Bob", "Allan", valueDouble)); //should return false because Bob Allan is not on the list therefore no change can be made to valueDouble

// doubleTest.inviteGuest("Carey", "Nachenberg", 3.5);

// doubleTest.inviteGuest("Joe", "Biden", 4.5);

// doubleTest.inviteGuest("Ben", "Ambrosio", 5.5);

// doubleTest.inviteGuest("Adam", "Levine", 6.5);

// doubleTest.inviteGuest("Amy", "Nachenberg", 7.5);

// doubleTest.inviteGuest("David", "Smallberg", 8.5);

// // for (int i = 0; i < doubleTest.guestCount(); i++) {

// // string firstName;

// // string lastName;

// // double value;

// // doubleTest.verifyGuestOnTheList(i, firstName, lastName, value);

// // cout << "First Name: " << firstName << " Last Name: " << lastName << " Double Value: " << value << endl;

// // }

// //Verfiy that the list is in alphabetical order

// doubleTest.alterGuest("Joe", "Biden", 5.5);

// double value;

// assert(doubleTest.matchInvitedGuest("Joe", "Biden", value)); //should return true because Joe Biden is on the list and value should be 5.5

// assert(value == 5.5); //should return 5.5 if alterGuest is working correctly

// assert(!doubleTest.alterGuest("Donald", "Trump", value)); //should return false because Donald Trump is not on the list and therefore it cant be altered

// assert(value == 5.5); //should have left value unchaged

// assert(doubleTest.inviteOrAlter("Donald", "Trump", 100)); //should return always return true and Donald Trump should have been added on the list

// assert(doubleTest.inviteOrAlter("Joe", "Biden", 100)); //should return always return true and Joe Biden should now map to 100

// assert(!doubleTest.crossGuestOff("Iron", "Man")); //Returns false because Iron Man is not on the list

// assert(doubleTest.crossGuestOff("Donald", "Trump")); //Donald Trump should now not be on the list

// string doubleFirst, doubleLast;

// assert(doubleTest.verifyGuestOnTheList(0, doubleFirst, doubleLast, value)); //should return true because the first guest is Ben Ambrosio and value should be 5.5

// assert(doubleFirst == "Ben" && doubleLast == "Ambrosio" && value == 5.5); //should return true

// assert(doubleTest.verifyGuestOnTheList(1, doubleFirst, doubleLast, value)); //should return true because the second guest is Joe Biden and value should be 100

// assert(doubleFirst == "Joe" && doubleLast == "Biden" && value == 100); //should return true

// assert(doubleTest.verifyGuestOnTheList(2, doubleFirst, doubleLast, value)); //should return true because the third guest is Adam Levine and value should be 6.5

// assert(doubleFirst == "Adam" && doubleLast == "Levine" && value == 6.5); //should return true

// assert(!doubleTest.verifyGuestOnTheList(-1, doubleFirst, doubleLast, value)); //should return false because the index is out of bounds

// assert(!doubleTest.verifyGuestOnTheList(6, doubleFirst, doubleLast, value)); //should return false because the index is out of bounds

// WeddingGuest doubleSwap;

// doubleTest.swapWeddingGuests(doubleSwap); //should swap the two lists

// assert(doubleTest.guestCount() == 0); //should return 0 because the doubleTest list should be empty

// assert(doubleSwap.guestCount() == 6); //should return 6 because the doubleTest1 list should have 6 guests

// WeddingGuest doubleSwap1;

// doubleSwap1.inviteGuest("Steve", "Jobs", 7.5);

// doubleSwap1.inviteGuest("Elon", "Musk", 8.5);

// doubleSwap1.inviteGuest("Mark", "Zuckerberg", 9.5);

// doubleTest.swapWeddingGuests(doubleSwap1); //should swap the two lists

// assert(doubleTest.guestCount() == 3); //should return 3 because the doubleTest list should have 3 guests

// assert(doubleSwap1.guestCount() == 0); //should return 0 because the doubleSwap1 list should be empty

// WeddingGuest doubleOne, doubleTwo, doubleResult;

// doubleOne.inviteGuest("Steve", "Jobs", 7.5);

// doubleOne.inviteGuest("Bill", "Gates", 8.5);

// doubleOne.inviteGuest("Elon", "Musk", 9.5);

// doubleOne.inviteGuest("Tim", "Cook", 10.5);

// doubleOne.inviteGuest("Jeff", "Bezos", 11.5);

// doubleOne.inviteGuest("Mark", "Zuckerberg", 12.5);

// doubleOne.inviteGuest("Steve", "Speilberg", 13.5);

// doubleOne.inviteGuest("Melinda", "Gates", 14.5);

// doubleTwo.inviteGuest("Steve", "Jobs", 8.5);

// doubleTwo.inviteGuest("Bill", "Gates", 8.5);

// doubleTwo.inviteGuest("Elon", "Musk", 9.5);

// doubleResult.inviteGuest("Mike", "Tyson", 9.5);

// doubleResult.inviteGuest("Kawhi", "Leonard", 10.5);

// joinGuests(doubleOne, doubleTwo, doubleResult);

// // for (int i = 0; i < doubleResult.guestCount(); i++) {

// // string firstName;

// // string lastName;

// // double value;

// // doubleResult.verifyGuestOnTheList(i, firstName, lastName, value);

// // cout << "First Name: " << firstName << " Last Name: " << lastName << " Double Value: " << value << endl;

// // }

// //Should only print out:

// // First Name: Jeff Last Name: Bezos Double Value: 11.5

// // First Name: Tim Last Name: Cook Double Value: 10.5

// // First Name: Bill Last Name: Gates Double Value: 8.5

// // First Name: Melinda Last Name: Gates Double Value: 14.5

// // First Name: Elon Last Name: Musk Double Value: 9.5

// // First Name: Steve Last Name: Speilberg Double Value: 13.5

// // First Name: Mark Last Name: Zuckerberg Double Value: 12.5

// //Check for aliasing

// assert(joinGuests(doubleOne, doubleOne, doubleResult)); //doubeResult should be a copy of doubleOne

// //doubleResult should print:

// // First Name: Jeff Last Name: Bezos Double Value: 11.5

// // First Name: Tim Last Name: Cook Double Value: 10.5

// // First Name: Bill Last Name: Gates Double Value: 8.5

// // First Name: Melinda Last Name: Gates Double Value: 14.5

// // First Name: Steve Last Name: Jobs Double Value: 7.5

// // First Name: Elon Last Name: Musk Double Value: 9.5

// // First Name: Steve Last Name: Speilberg Double Value: 13.5

// // First Name: Mark Last Name: Zuckerberg Double Value: 12.5

// WeddingGuest doubleTemp = doubleOne;

// assert(!joinGuests(doubleTemp, doubleTwo, doubleTemp)); //This should not affect bee the same as having an oridinary doubleResult; also returns false because there are Steve Jobs with different values

// //doubleTemp should print:

// // First Name: Jeff Last Name: Bezos Double Value: 11.5

// // First Name: Tim Last Name: Cook Double Value: 10.5

// // First Name: Bill Last Name: Gates Double Value: 8.5

// // First Name: Melinda Last Name: Gates Double Value: 14.5

// // First Name: Elon Last Name: Musk Double Value: 9.5

// // First Name: Steve Last Name: Speilberg Double Value: 13.5

// // First Name: Mark Last Name: Zuckerberg Double Value: 12.5

// attestGuests("Steve", "\*", doubleOne, doubleResult);

// //doubleResult should print out:

// // First Name: Steve Last Name: Jobs Double Value: 7.5

// // First Name: Steve Last Name: Speilberg Double Value: 13.5

// attestGuests("\*", "Gates", doubleOne, doubleResult);

// // doubleResult should print out:

// // First Name: Bill Last Name: Gates Double Value: 8.5

// // First Name: Melinda Last Name: Gates Double Value: 14.5

// attestGuests("\*", "\*", doubleOne, doubleResult);

// // //doubleResult Should be a copy of doubleOne

// attestGuests("", "", doubleOne, doubleResult);

// // //doubleResult should be empty

// // //Check if function works correctly when faced aliasing

// attestGuests("\*", "Gates", doubleOne, doubleOne);

// //doubleOne should print:

// // First Name: Bill Company First Letter: Gates Double Value: 8.5

// // First Name: Melinda Company First Letter: Gates Double Value: 14.5

}

int main() {

test();

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

return 0;

}