Sari Abu-Hamad

205-600-741

CS32 Project 2

**Design of implementation:**

I implemented a doubly linked list that is not circular and does not contain a dummy node but contains both a head and tail node to represent the class WeddingGuest. Each node represents a guest and contains the first name, last name, GuestType variable, a pointer to the next node, and a pointer to the previous node.

I chose this implementation so that each node could represent 1 guest and hold all values associated with it. The incorporation of the tail and head nodes allow me to access the end and beginning of the list and the double linkage with previous and next node pointers allows me to easily navigate the list, which overall provides an ease of access to and navigation of the different guest information throughout the list.

**Notable Obstacles:**

A notable obstacle I overcame was end behavior for the inviteGuest function. I had trouble with my linkedList going out of bounds in certain cases when adding a guest. I solved this by writing out my pseudocode and being very precise with certain scenarios around the head and tail nodes. Through this method, in addition to debugging a multitude of test cases, I was able to create an algorithm that would handle the scenarios I could come up with.

**Pseudo code:**

*Destructor*

* return if list contains no elements
* if list is 1 element big
  + destroy head and return
* loop through list while deleting previous
* delete tail to account for last Node

*Copy Constructor*

* initialize counter
* loop through the other list until reach nullptr
  + if counter is zero
    - initialize the list with a head and then increment counter
  + if counter is not zero
    - add a new node and increment size count variable
* if i is the size of the other list, it has fullly looped through so the Node used to loop through this list is the tail
* if the other list is size 0, initialize this list like in the default constructor

*Assignment Operator*

* check if this list is equal to the list passed to prevent aliasing
* if the size of the current list is 0
  + do nothing
* if the size of the current list is 1
  + delete the head
* if the size of the current list is > 1
  + loop through it deleting each previous node and then the tail
* if the other list's head is null
  + initialize this list like the default constructor then return it
* if the other list's head is not null
  + copy the values of the other list's head into the current head and increment size counter
* go to the next value in the other list
* loop through the other list until all values are copied into the new list
* return this list

*inviteGuest*

* if the head of this list is null
  + add the new guest with the values passed
  + increment size
  + return true
* if the next value is null
  + if the values of head are equal to the passed first and last names
    - return false
  + if they are not the same
    - add a new guest in the correct alphabetical order
    - increment size
    - return true
* if the last name passed should be earlier alphabetically than head's last name
  + add the guest before head
  + increment size
  + return true
* loop through the existing list, and if the current first and last name are found return false
* loop through the list again until the value for last name should come after (alphabetically) the last name in the list
  + if the next value is going to be nullptr
    - it will be reaching the end and thus need to check if the lastName should go before or after the end
      * add guest as such
      * increment size
      * return true
* if the node stopped at has the same last name, check first name
  + loop through the elements while the last name is the same and the firstName is earlier in the alphabetical order
    - if the next value is going to be nullptr
      * it will be reaching the end and thus need to check if the firstName should go before or after the end
        + add guest as such
        + increment size
        + return true
* if the element is currently at the head and the new guest has to be added before the head
  + add guest before head
  + increment size
  + return true
* If none of the previous cases were triggered
  + guest should be added in the middle of list
    - add guest while keeping track of prev and next in the list
    - increment size
    - return true;

*alterGuest*

* loop through list
  + if full name is found set the new value to the one passed and return true
* if not found return false

*inviteOrAlter*

* loop through list
  + if full name is found set the new value to the one passed and return true
* if not found, add it and return true

*crossGuestOff*

* loop through list
  + ​​if guest is found
    - check if guest is at head to account for out of bounds errors
      * delete node accordingly while making sure list stays intact
      * return true
    - check if guest is at tail to account for out of bounds errors
      * delete node accordingly while making sure list stays intact
      * return true
    - If not, then it is in the middle
      * delete node accordingly while making sure list stays intact
      * return true
  + If not found return false

*invitedToWedding*

* loop through list
  + if found return true
* If not found return false

*matchInvitedGuest*

* loop through list
  + if found
    - set value of GuestType passed to the value assigned to the guest in the list
    - return true
* if not found, return false

*verifyGuestOnList*

* if passed location of guest node is out of bounds
  + return false
* loop until reach the element of interest
  + collect the values at that element in the variables passed by reference
  + return true

*swapWeddingGuests*

* declare temporary WeddingGuest
* Using assignment operator (=)
  + assign the values of the temporary list to those of this list
  + assign the values of this list to those of the passed list
  + assign the values of the other list to those of the temporary list

*joinGuests*

* use bool to track that there are no different values
  + initially true
* if there are no guests in either list
  + set the joined list equal to the non empty list
  + return true or false depending on bool value
* set the joined list equal to the first list
* loop through the second list capturing each value of the guests
  + if the guest is already on list 1 capture the GuestType value given on list 1
  + if the values are not equal
    - set the bool to false and cross the guest off the joined list
  + if the guest is not on list 1 add it to the joined list using inviteGuest
* return true or false depending on bool value

*attestGuests*

* if both first and last are wildcards
  + assign values of list one to result list
  + return
* create a tempList to store list one values to account for aliasing before setting odResult to an empty list
* if the first search is the wildcard
  + loop through the list of values from list one
    - if the last name is equal to the one being searched for
      * add it to the result list
* if the last search is the wildcard
  + loop through the list of values from list one
    - if the first name is equal to the one being searched for
      * add it to the result list
* if neither first or last are wildcards
  + if list 1 contains both names being searched for
    - add it to the resulting list

return

**Test Cases:**

WeddingGuest groomsmen;

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

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

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

groomsmen.inviteGuest ("Justin", "Sandobal", 37);

groomsmen.inviteGuest ("Nelson", "Villaluz", 38);

groomsmen.inviteGuest ("Long", "Le", 41);

groomsmen.inviteGuest ("Robert", "Wells", 49);

WeddingGuest groomsmen2;

groomsmen2 = groomsmen;

for (int n = 0; n < groomsmen2.guestCount(); n++)

{

string first;

string last;

int val;

groomsmen.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

/\*

Should Print

**Tony Ambrosio 40**

**Long Le 41**

**Justin Sandobal 37**

**Nelson Villaluz 38**

**Robert Wells 44**

**Mike Wu 43**

\*/

Ensure that both assignment operator and inviteGuest work properly to put the contents in alphabetical order (and not add the second occurrence of the same name) and then assign them into the new WeddingGuest object with simply the assignment operator. This also tests if verifyGuestOnTheList works to provide the data for first last and val to be displayed.

// default constructor

WeddingGuest lal;

lal.inviteGuest("Billy", "Bob", "32"); //add one guest

WeddingGuest lal2(lal);

string first,last,value;

assert(lal2.guestCount() == 1 && lal2.verifyGuestOnTheList(0, first, last, value) && first == "Billy" && last == "Bob" && value == "32"); //tests if values were passed correctly to the new list

assert(lal.guestCount() == 1 && lal.verifyGuestOnTheList(0, first, last, value) && first == "Billy" && last == "Bob" && value == "32"); //tests if values are still in original list

Ensures that the copy constructor works to create a copy of the existing list and holds the correct and same values within both lists.

// default constructor

WeddingGuest lal;

lal.inviteGuest("Billy", "Bob", "32"); //add guests

lal.inviteGuest("Benny", "Bob", "31");

lal.inviteGuest("Bobby", "Bob", "30");

WeddingGuest lal2(lal);

string first,last,value;

assert(lal2.guestCount() == 3 && lal2.verifyGuestOnTheList(0, first, last, value) && first == "Benny" && last == "Bob" && value == "31");

assert(lal.guestCount() == 3 && lal.verifyGuestOnTheList(1, first, last, value) && first == "Billy" && last == "Bob" && value == "32");

Ensures that the copy constructor works with more than 1 guest added and places them correctly in order within both lists

WeddingGuest groomsmen;

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

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

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

groomsmen.alterGuest("Robert", "Wells", 37); //same full name

groomsmen.inviteGuest ("Nelson", "Villaluz", 38);

groomsmen.inviteGuest ("Long", "Le", 41);

groomsmen.inviteGuest ("Billy", "Wells", 49); //same last name (order by first)

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

{

string first;

string last;

int val;

groomsmen.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

/\***T**

Should Print:**ny Ambrosio 40**

**Long Le 41**

**Nelson Villaluz 38**

**Billy Wells 49**

**Robert Wells 37**

**Mike Wu 43**

\*/

Tests if inviteGuest can handle ordering by first name alphabetically when given two inputs of the same last name in addition to testing if alterGuest works to change the mapped value of a Guest found in the list

// default constructor

WeddingGuest lal;

lal.inviteGuest("Billy", "Bob", "32"); //add one guest

// For an empty list:

lal.inviteOrAlter("Billy", "Bob", "34"); //should alter the value of the guest to 34

string first, last, value;

assert(lal.verifyGuestOnTheList(0, first, last, value) == true && value == "34");

Tests that inviteOrAlter can handle changing the value of an existing guest

// default constructor

WeddingGuest lal;

lal.inviteGuest("Billy", "Bob", "32"); //add one guest

// For an empty list:

lal.inviteOrAlter("Billy", "Jean", "34"); //should alter the value of the guest to 34

string first, last, value;

assert(lal.verifyGuestOnTheList(1, first, last, value) == true && value == "34" && last == "Jean"); //checks that Billy Jean 34 was added

Tests that inviteOrAlter can handle inviting a guest that was not previously on the list

WeddingGuest groomsmen;

groomsmen.inviteGuest ("Zack", "Wells", 39);

groomsmen.inviteGuest ("Dan", "Wells", 40); //same last name

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

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

groomsmen.inviteGuest ("Justin", "Sandobal", 37);

groomsmen.inviteGuest ("Nelson", "Villaluz", 38);

groomsmen.inviteGuest ("Long", "Le", 41);

groomsmen.inviteGuest ("Robert", "Wells", 49);

groomsmen.inviteGuest ("Tony", "Sandobal", 68);

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

{

string first;

string last;

int val;

groomsmen.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

/\*Should Print

**Long Le 41**

**Justin Sandobal 37**

**Tony Sandobal 40**

**Nelson Villaluz 38**

**Dan Wells 40**

**Robert Wells 44**

**Zack Wells 39**

**Mike Wu 43**

\*/

Ensures that inviteGuest can handle ordering multiple guests of the same last name in addition to more than one group of multiples with the same last name

WeddingGuest groomsmen;

groomsmen.inviteGuest ("Zack", "Wells", 39);

groomsmen.inviteGuest ("Dan", "Wells", 40);

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

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

groomsmen.inviteGuest ("Justin", "Sandobal", 37);

groomsmen.inviteGuest ("Nelson", "Villaluz", 38);

groomsmen.inviteGuest ("Long", "Le", 41);

groomsmen.inviteGuest ("Robert", "Wells", 49);

groomsmen.inviteGuest ("Tony", "Sandobal", 68);

WeddingGuest groomsmen2;

groomsmen2.inviteGuest ("Peter", "Piper", 12);

groomsmen.swapWeddingGuests(groomsmen2);

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

{

string first;

string last;

int val;

groomsmen.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

cout << endl;

for (int n = 0; n < groomsmen2.guestCount(); n++)

{

string first;

string last;

int val;

groomsmen2.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

/\*Should Print

**Peter Piper 12**

**Long Le 41**

**Justin Sandobal 37**

**Tony Sandobal 68**

**Nelson Villaluz 38**

**Dan Wells 40**

**Robert Wells 44**

**Zack Wells 39**

**Mike Wu 43**

\*/

Ensures that the swapWeddingGuest function works correctly and can handle different sizes of lists. Also checks that the definition of the assignment operator within the class, which is used to implement the swapWeddingGuest function, works correctly.

WeddingGuest groomsmen;

groomsmen.inviteGuest ("Zack", "Wells", 39);

groomsmen.inviteGuest ("Dan", "Bob", 40);

groomsmen.inviteOrAlter("Dan", "Bob", 32);

groomsmen.inviteOrAlter("Bro", "Bob", 35);

groomsmen.inviteOrAlter("Billy", "Jean", 31);

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

{

string first;

string last;

int val;

groomsmen.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

/\*Should Print

**Bro Bob 35**

**Dan Bob 32**

**Billy Jean 31**

**Zack Wells 39**

\*/

Tests to ensure that inviteOrAlter will work as intended to change an existing guest’s value or add that guest to the list in order if they are not found.

WeddingGuest groomsmen;

groomsmen.inviteGuest ("Zack", "Wells", 39);

groomsmen.inviteGuest ("Dan", "Bob", 40);

WeddingGuest groomsmen2;

groomsmen2.inviteGuest ("Billy", "Bob", 23);

groomsmen2.inviteGuest ("Zack", "Wells", 73);

groomsmen2.inviteGuest ("Jonathan", "Wells", 12);

groomsmen2.inviteGuest ("Dan", "Bob", 40);

groomsmen2.inviteGuest ("Bobby", "Bob", 40);

WeddingGuest combinedList;

cout << "First List: " << endl;

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

{

string first;

string last;

int val;

groomsmen.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

cout << endl << "Second List: " << endl;

for (int n = 0; n < groomsmen2.guestCount(); n++)

{

string first;

string last;

int val;

groomsmen2.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

assert(joinGuests(groomsmen, groomsmen2, combinedList) == false);

cout << endl << "Combined List: " << endl;

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

{

string first;

string last;

int val;

combinedList.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

/\*Should Print

**First List:**

**Dan Bob 40**

**Zack Wells 39**

**Second List:**

**Billy Bob 23**

**Bobby Bob 40**

**Dan Bob 40**

**Jonathan Wells 12**

**Zack Wells 73**

**Combined List:**

**Billy Bob 23**

**Bobby Bob 40**

**Dan Bob 40**

**Jonathan Wells 12**

\*/

Tests that joinGuests works to fulfill the rule that when there are different values corresponding to the same name, that name should not appear on the combined List and it should return false. Also tests if joinGuests can add other elements normally even if list 1 and list 2 are different sizes. Also tests if joinGuests can handle 2 names with the same values and have it appear once.

WeddingGuest groomsmen;

WeddingGuest groomsmen2;

groomsmen2.inviteGuest ("Billy", "Bob", 23);

groomsmen2.inviteGuest ("Zack", "Wells", 73);

groomsmen2.inviteGuest ("Jonathan", "Wells", 12);

groomsmen2.inviteGuest ("Dan", "Bob", 40);

groomsmen2.inviteGuest ("Bobby", "Bob", 40);

WeddingGuest combinedList;

cout << "First List: " << endl;

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

{

string first;

string last;

int val;

groomsmen.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

cout << endl << "Second List: " << endl;

for (int n = 0; n < groomsmen2.guestCount(); n++)

{

string first;

string last;

int val;

groomsmen2.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

assert(joinGuests(groomsmen, groomsmen2, combinedList) == true);

cout << endl << "Combined List: " << endl;

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

{

string first;

string last;

int val;

combinedList.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

/\*Should Print

**First List:**

**Second List:**

**Billy Bob 23**

**Bobby Bob 40**

**Dan Bob 40**

**Jonathan Wells 12**

**Zack Wells 73**

**Combined List:**

**Billy Bob 23**

**Bobby Bob 40**

**Dan Bob 40**

**Jonathan Wells 12**

**Zack Wells 73**

\*/

Tests that joinGuests can handle an empty list and return true.

WeddingGuest groomsmen;

groomsmen.inviteGuest ("Cobey", "C", 35);

groomsmen.inviteGuest ("Dan", "H", 38);

groomsmen.inviteGuest ("Dan", "V", 44);

groomsmen.inviteGuest ("Dion", "V", 48);

cout << "First List: " << endl;

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

{

string first;

string last;

int val;

groomsmen.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

WeddingGuest result;

attestGuests("Dan", "\*", groomsmen, result);

cout << endl << "Result: " << endl;

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

{

string first;

string last;

int val;

result.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

/\*Should Print

**First List:**

**Cobey C 35**

**Dan H 38**

**Dan V 44**

**Dion V 48**

**Result:**

**Dan H 38**

**Dan V 44**

\*/

Tests that attestGuests works with a wildcard for the last name and result contains guests with only the matching first name sought after within the list passed along with it

WeddingGuest groomsmen;

groomsmen.inviteGuest ("Caitlyn", "J", 71);

groomsmen.inviteGuest ("Khloe", "K", 37);

groomsmen.inviteGuest ("Kim", "K", 40);

groomsmen.inviteGuest ("Kanye", "W", 44);

cout << "First List: " << endl;

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

{

string first;

string last;

int val;

groomsmen.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

WeddingGuest result;

attestGuests("\*", "K", groomsmen, result);

cout << endl << "Result: " << endl;

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

{

string first;

string last;

int val;

result.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

/\*Should Print

**First List:**

**Caitlyn J 71**

**Khloe K 37**

**Kim K 40**

**Kanye W 44**

**Result:**

**Khloe K 37**

**Kim K 40**

\*/

Tests that attestGuests works with a wildcard for the first name and result contains guests of only the matching last name sought after within the list passed along with it

WeddingGuest groomsmen;

groomsmen.inviteGuest ("Cobey", "C", 35);

groomsmen.inviteGuest ("Dan", "H", 38);

groomsmen.inviteGuest ("Dan", "V", 44);

groomsmen.inviteGuest ("Dion", "V", 48);

cout << "First List: " << endl;

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

{

string first;

string last;

int val;

groomsmen.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

WeddingGuest result;

attestGuests("\*", "\*", groomsmen, result);

cout << endl << "Result: " << endl;

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

{

string first;

string last;

int val;

result.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

/\*Should Print

**First List:**

**Cobey C 35**

**Dan H 38**

**Dan V 44**

**Dion V 48**

**Result:**

**Cobey C 35**

**Dan H 38**

**Dan V 44**

**Dion V 48**

\*/

Tests that attestGuests works with a wildcard for the first and last names and result contains a copy of the list passed along with it.

WeddingGuest groomsmen;

groomsmen.inviteGuest ("Caitlyn", "J", 71);

groomsmen.inviteGuest ("Khloe", "K", 37);

groomsmen.inviteGuest ("Kim", "K", 40);

groomsmen.inviteGuest ("Kanye", "W", 44);

cout << "First List: " << endl;

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

{

string first;

string last;

int val;

groomsmen.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

WeddingGuest result;

attestGuests("Kim", "K", groomsmen, result);

cout << endl << "Result: " << endl;

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

{

string first;

string last;

int val;

result.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

/\*Should Print

**First List:**

**Caitlyn J 71**

**Khloe K 37**

**Kim K 40**

**Kanye W 44**

**Result:**

**Kim K 40**

\*/

Tests to see if attestGuest works even with no wildcards passed to it. Also checks if matchInvitedGuest works in finding the specified invited guest

WeddingGuest groomsmen;

groomsmen.inviteGuest ("Caitlyn", "J", 71);

groomsmen.inviteGuest ("Khloe", "K", 37);

groomsmen.inviteGuest ("Kim", "K", 40);

groomsmen.inviteGuest ("Kanye", "W", 44);

cout << "First List: " << endl;

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

{

string first;

string last;

int val;

groomsmen.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

WeddingGuest result;

attestGuests("", "", groomsmen, result);

cout << endl << "Result: " << endl;

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

{

string first;

string last;

int val;

result.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

/\*Should Print

**First List:**

**Caitlyn J 71**

**Khloe K 37**

**Kim K 40**

**Kanye W 44**

**Result:**

\*/

Tests to see if attestGuests works with no matching values in the list to fsearch and lsearch (and empty strings)

WeddingGuest groomsmen;

groomsmen.inviteGuest ("Caitlyn", "J", 71);

groomsmen.inviteGuest ("Khloe", "K", 37);

groomsmen.inviteGuest ("Kim", "K", 40);

groomsmen.inviteGuest ("Kanye", "W", 44);

cout << "First List: " << endl;

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

{

string first;

string last;

int val;

groomsmen.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

WeddingGuest result;

result.inviteGuest("Sir", "Edwin", 39);

attestGuests("\*", "K", groomsmen, result);

cout << endl << "Result: " << endl;

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

{

string first;

string last;

int val;

result.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

/\*Should Print

**First List:**

**Caitlyn J 71**

**Khloe K 37**

**Kim K 40**

**Kanye W 44**

**Result:**

**Khloe K 37**

**Kim K 40**

\*/

Tests if attestGuests works as expected even if the result WeddingGuest list already has a value

WeddingGuest groomsmen;

groomsmen.inviteGuest ("Caitlyn", "J", 71);

groomsmen.inviteGuest ("Khloe", "K", 37);

groomsmen.inviteGuest ("Kim", "K", 40);

groomsmen.inviteGuest ("Kanye", "W", 44);

cout << "First List: " << endl;

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

{

string first;

string last;

int val;

groomsmen.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

attestGuests("\*", "K", groomsmen, groomsmen); //pass same list to attestGuests

cout << endl << "Result: " << endl;

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

{

string first;

string last;

int val;

groomsmen.verifyGuestOnTheList (n, first, last, val);

cout << first << " " << last << " " << val << endl;

}

/\*Should Print

**First List:**

**Caitlyn J 71**

**Khloe K 37**

**Kim K 40**

**Kanye W 44**

**Result:**

**Khloe K 37**

**Kim K 40**

\*/

Tests that even when the lists passed to attestGuests point to the same list (aliasing), the result still contains the correctly searched terms

// default constructor

WeddingGuest lal;

lal.inviteGuest("Billy", "Bob", "32"); //add one guest

// For an empty list:

assert(lal.guestCount() == 1); // test size

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

assert(lal.crossGuestOff("Billy", "Bob")); // erases guest

Tests that guestCount, noGuests, and crossGuestoff function correctly even with just 1 guest

// 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

Tests that guestCount, noGuests, and crossGuestOff function correctly even with an empty list

#include "WeddingGuest.h"

#include <string>

#include <iostream>

#include <cassert>

using namespace std;

void test() {

WeddingGuest eliteSingles;

assert(eliteSingles.inviteGuest("Jackie", "S",

"jackies@elitesingles.com"));

assert(eliteSingles.inviteGuest("Mark", "P",

"markp@elitesingles.com"));

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

string first, last, e;

assert(eliteSingles.verifyGuestOnTheList(0, first, last, e)

&& e == "markp@elitesingles.com");

assert(eliteSingles.verifyGuestOnTheList(1, first, last, e)

&& (first == "Jackie" && e == "jackies@elitesingles.com"));

return;

}

int main() {

test();

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

return 0;

}

Test that GuestType can handle being a string as well; should cout “Passed all tests\n”