**Project 2**

**A description of the design of your implementation and why you chose it. (A couple of sentences will probably suffice, perhaps with a picture of a typical List and an empty List. Is your list circular? Does it have a dummy node? What's in your nodes?)**

The structure I have used is a doubly linked list. My list is not circular to better keep track of the front and end, and does not use a dummy node.

Each node contains 3 public data variables. These are first name, last name, and value. They also have “prev” and “next” pointers that indicate the names in the list that are directly in front and behind them, in alphabetical order.

The node struct is declared as a private member in the WeddingGuest class.

In the list nodes, the items are of type std::string.

The typical list will look like:

head 🡨🡪 node 🡨🡪 node … 🡪 nullptr

**A brief description of notable obstacles you overcame**.

I first tried using a circular list, thinking this would help with efficiency because the algorithm would be able to traverse in either direction to find a given name. However, I found this to be difficult to keep track of the head and tail, since I couldn’t set while conditions to “while (p != nullptr)”, for example.

By switching to a noncircular list, I was able to better track the head and ends of the list.

I also tried using a tail pointer in addition to the head pointer I have in the program. This later also proved to be unnecessary because I could easily get to the tail by using a while loop and iterating through the list.

The inviteGuest function was especially challenging to write, because I had trouble accounting for all the possible cases in which a guest could be added to the list. When there is a last name match, the algorithm should check first names to see where the new name falls within the last names. There could also be several people with the same last names, in which case being the first person with the last name and the last person with the last name may require different checks (e.g. Adam Smith vs Robert Smith).

I have also always had trouble writing copy constructors. This is something I need to practice more and get used to writing.

**Pseudocode for non-trivial algorithms**

**bool WeddingGuest::inviteGuest(const std::string& firstName, const std::string& lastName, const GuestType& value) {**

if name is already in list,

return false.

Otherwise, increase size by 1

Create new node and assign values

If list was empty or head’s last name is greater or equal to lastName

Link new node in front of head

Set new node as head

Return true

Else, If node should go in middle or end of list

Iterate to where node should be placed (match last names)

Break;

If adding node to end of list,

Link new node to end

Return true

Relink node to middle of the list

Return true;

}

**bool WeddingGuest::alterGuest(const std::string& firstName, const std::string& lastName, const GuestType& value) {**

while pointer is not null,

if full name matches, break

increment pointer to next

if pointer is not null,

copy pointer’s value into value

return true

otherwise, return false.

}

**bool WeddingGuest::crossGuestOff(const std::string& firstName, const std::string& lastName) {**

// If the full name is equal to a full name currently in the list, remove the full name and value from the list and return

// true. Otherwise, make no change to the list and return false.

if not invited or not on guest list

make no change and return false.

subtract 1 from number of guests.

//CASE 1:------------------- FRONT

if full name matches the first name in the list,

if list only contains head,

delete head

return true

set pointer to node to be deleted

set head to next pointer

unlink the node to be deleted

delete node and return true.

//CASE 2:------------------- MIDDLE/END

While p is not null,

If the full name matches p->next,

Break

Iterate to next pointer

//at this point, p should be the node BEFORE the deleted node.

If p is not nullpointer

Set node to be deleted to p->next

Relink nodes around the node to be deleted

If last node is not null,

link the prev-> pointer of the node after the deleted node ( killMe->next->prev) to p

delete node;

return true;

**bool WeddingGuest::invitedToTheWedding(const std::string& firstName, const std::string& lastName) const {**

// Return true if the full name is equal to a full name currently in the list, otherwise false.

While p is not null,

If full name matches

Return true.

Iterate to next node

If no matches and p is null,

return false.

}

**void WeddingGuest::swapWeddingGuests(WeddingGuest& other)** {

// Exchange the contents of this list with the other one.

introduce temporary placeholder pointers head and tail

introduce tempoary int tempsize to store map size

assign other size to original map

assign original size to other size

assign contents of other to original map

assign contents of original map to other

}

bool joinGuests(const WeddingGuest& odOne, const WeddingGuest& odTwo, WeddingGuest& odJoined) {

if the joined list is not empty,

create an empty list

swap result with empty list

create bool valuesSame to be true;

introduce temporary variables to store first name, last name, two values

for each item in odOne,

verify that guest is on the list

check whether each item is contained in odTwo.

If name is distinct (not in temp2)

insert into result

otherwise,

check if values are equal. if equal,

insert into odJoined

otherwise, set valuesSame to false and do not put into the combined list.

loop through all items in odTwo

get tempval of item and check whether name is already in odJoined

if values are not the same

set valuesSame to false

do not put into result

otherwise, put into odJoined

if name is not already in odJoined, put into odJoined

return valuesSame;

}

**void attestGuests(const std::string& fsearch, const std::string& lsearch, const WeddingGuest& odOne, WeddingGuest& odResult) {**

if the result list is not empty,

create an empty list

swap result with empty list

assign placeholder variables to first name, last name, value

if both fsearch and lsearch are wildcards,

odResult is equal to odone.

Otherwise,

For each item in odOne

Verify name is on the list

If firstname is wildcard and last name is equal to lsearch,

Add to odResult

If lastname is wildcard and first name is equal to fsearch,

Add to odResult

If last and first name both match searched terms,

Add to odResult

}

**A list of test cases that would thoroughly test the functions. Be sure to indicate the purpose of the tests.**

|  |  |
| --- | --- |
| **TEST CASE** | REASON |
| WeddingGuest emptyList;  assert(emptyList.guestCount() == 0);  assert(emptyList.noGuests());  assert(!emptyList.crossGuestOff("empty", "list"));  WeddingGuest l1;  l1.inviteGuest("bae", “abe”, "24");  l1.inviteGuest("bae", “greg”, "133");  l1.inviteGuest(“abe”, “greg”, "24");  l1.inviteGuest(“aaa”, “aaa”, "10");  l1.inviteGuest(“abe”, "bae", "18");  l1.inviteGuest(“greg”, “abe”, "34");  l1.inviteGuest(“greg”, "bae", "42");  l1.inviteGuest(“aaa”, “greg”, "20");  l1.inviteGuest(“aaa”, "bae", "76");  l1.inviteGuest(“greg”, “aaa”, "56");  l1.inviteGuest(“greg”, “greg”, "0");  l1.inviteGuest(“abe”, “aaa”, "13");  l1.inviteGuest(“aaa”, “abe”, "27");  assert(l1.guestCount() == 13);  l1.inviteGuest(“greg”, “greg”, "0");  assert(l1.guestCount() == 13);    string first;  string last;  string value;  l1.verifyGuestOnTheList(0, first, last, value);  assert(value == "0");  l1.verifyGuestOnTheList(5, first, last, value);  assert(value == "5");  l1.verifyGuestOnTheList(13, first, last, value);  assert(value == "13");  WeddingGuest l2;  l2.inviteGuest(“alice”, “alice”, "0");  l2.inviteGuest(“dan”, “alice”, "1");  l2.inviteGuest(“alice”, “dan”, "2");  l2.inviteGuest(“dan”, “dan”, "3");  assert(l2.guestCount() == 4);  WeddingGuest l3 = l1;  assert(l3.guestCount() == 16);  l3.verifyGuestOnTheList(0, first, last, value);  assert(value == "0");  l3.verifyGuestOnTheList(5, first, last, value);  assert(value == "5");  l3.verifyGuestOnTheList(15, first, last, value);  assert(value == "15");  l1 = l2;  assert(l1.guestCount() == 4);  l1.verifyGuestOnTheList(0, first, last, value);  assert((first == “alice” && last == “alice” && value == "0"));  l1.verifyGuestOnTheList(1, first, last, value);  assert((first == “dan” && last == “alice” && value == "1"));  l1.verifyGuestOnTheList(2, first, last, value);  assert((first == “alice” && last == “dan” && value == "2"));  l1.verifyGuestOnTheList(3, first, last, value);  assert((first == “dan” && last == “dan” && value == "3"));  assert(l2.alterGuest(“alice”, “alice”, "100"));  assert(l2.verifyGuestOnTheList(0, first, last, value) && value == "100");  assert(l2.alterGuest(“hilary”, “hilary”, "200"));  assert(l2.verifyGuestOnTheList(15, first, last, value) && value == "200");  assert(l2.alterGuest(“dan”, “dan”, "300"));  assert(!l2.alterGuest(“jack”, “jack”, "300"));  assert(!l2.alterGuest(“jack”, “alice”, "300"));  assert(l1.inviteOrAlter(“alice”, “alice”, "13"));  assert(l1.verifyGuestOnTheList(0, first, last, value) && value == "13");  assert(l1.inviteOrAlter(“alice”, “dan”, "25"));  assert(l1.verifyGuestOnTheList(2, first, last, value) && value == "25");  assert(l1.inviteOrAlter(“greg”, “greg”, "264"));  assert(l1.verifyGuestOnTheList(4, first, last, value) && value == "264");  assert(l1.inviteOrAlter("", "", ""));  assert(l3.crossGuestOff(“alice”, “alice”));  assert(!l3.invitedToTheWedding(“alice”, “alice”) && l3.verifyGuestOnTheList(0, first, last, value) && value == "1");  assert(l3.guestCount() == 15);  assert(l3.crossGuestOff(“hilary”, “hilary”));  assert(!l3.invitedToTheWedding(“hilary”, “hilary”) && l3.verifyGuestOnTheList(13, first, last, value) && value == "14");  assert(l3.guestCount() == 14);  assert(l3.crossGuestOff(“dan”, “dan”));  assert(!l3.invitedToTheWedding(“dan”, “dan”) && l3.verifyGuestOnTheList(3, first, last, value) && value == "4");  assert(l3.guestCount() == 13); assert(!l3.crossGuestOff(“leo”, “leo”));  WeddingGuest l4 = l1;  string testValue = "no matches";  assert(!l4.matchInvitedGuest(“jack”, “jack”, testValue) && testValue == "no matches");  assert(l4.matchInvitedGuest(“alice”, “alice”, testValue) && testValue == "0");  assert(l4.matchInvitedGuest(“dan”, “dan”, testValue) && testValue == "5");  assert(l4.matchInvitedGuest(“greg”, “greg”, testValue) && testValue == "10");  assert(l4.matchInvitedGuest(“hilary”, “hilary”, testValue) && testValue == "15");  WeddingGuest l5;  l5.inviteGuest(“alice”, “alice”, "0");  l5.inviteGuest(“dan”, “dan”, "1");  assert(l5.guestCount() == 2 && l4.guestCount() == 16);  l4.swapWeddingGuests(l5);  assert(l5.guestCount() == 16 && l4.guestCount() == 2);  assert(l4.verifyGuestOnTheList(0, first, last, value) && first == “alice” && last == “alice” && value == "0");  assert(l4.verifyGuestOnTheList(1, first, last, value) && first == “dan” && last == “dan” && value == "1");  assert(l5.verifyGuestOnTheList(15, first, last, value) && first == “hilary” && last == “hilary” && value == "15");  WeddingGuest bob1;  bob1.inviteGuest(“alice”, “alice”, "0");  bob1.inviteGuest(“dan”, “dan”, "0");  bob1.inviteGuest(“greg”, “greg”, "3");  bob1.inviteGuest(“hilary”, “hilary”, "3");  bob1.inviteGuest(“ethan”, “ethan”, “24”);  WeddingGuest abcd;  abcd.inviteGuest(“jack”, “jack”, "0");  abcd.inviteGuest(“leo”, “leo”, "0");  abcd.inviteGuest(“dan”, “dan”, "3");  abcd.inviteGuest(“greg”, “greg”, "3");  WeddingGuest mergedList;  mergedList.inviteGuest(“jack”, “jack”, "0");  assert(!joinGuests(bob1, abcd, mergedList));  joinGuests(bob1, abcd, mergedList);  assert(mergedList.guestCount() == 5 && !mergedList.invitedToTheWedding(“dan”, “dan”) && mergedList.invitedToTheWedding(“greg”, “greg”));  assert(mergedList.verifyGuestOnTheList(0, first, last, value) && first == “alice” && last == “alice” && value == "0");  assert(mergedList.verifyGuestOnTheList(1, first, last, value) && first == “greg” && last == “greg” && value == "3");  assert(mergedList.verifyGuestOnTheList(4, first, last, value) && first == “leo” && last == “leo” && value == "0");    WeddingGuest testAttest;  testAttest.inviteGuest("Cobey", “greg”, "27");  testAttest.inviteGuest("Tina", "Hu", "30");  testAttest.inviteGuest("Tina", "Jack", "32");  testAttest.inviteGuest("Dion", "Jack", "37");  WeddingGuest xyz;  attestGuests("Tina", "\*", testAttest, xyz);  assert(xyz.guestCount() == 2 && xyz.invitedToTheWedding("Tina", "Hu") && xyz.invitedToTheWedding("Tina", "Jack"));  attestGuests("\*", "V", testAttest, xyz);  assert(xyz.guestCount() == 2 && xyz.invitedToTheWedding("Tina", "Jack") && xyz.invitedToTheWedding("Dion", "Jack"));  attestGuests("\*", "\*", testAttest, xyz);  assert(xyz.guestCount() == 4);  attestGuests("Tina", "H", testAttest, xyz);  assert(xyz.guestCount() == 1 && xyz.invitedToTheWedding("Tina", "Hu")); | Testing:  Empty list  guestCount  noGuests  crossGuestOff  ensure inviteGuest will not insert duplicates  Checking guestCount  Testing verifyGuestOnTheList  First element  Elements in between  Last element  guestCount  Testing whether copy constructor made a deep copy  Testing assignment operator  Testing verifyGuestOnTheList  Whether assignment operator correctly reassigned list 1.  Testing alterGuest  verifyGuestOnTheList    alterGuest when the given names do not exist  inviteOrAlter  testing  verifyGuestOnTheList  Testing crossGuestOff  m\_size was subtracted correctly  Testing matchInvitedGuest  First value  Middle value  Last value  Testing inviteGuest  swapWeddingGuests  matchInvitedGuest  Testing swapWeddingGuests  verifyGuestOnTheList  Testing joinGuests  Testing when mergedList is not empty!  Testing attestGuests  Last name wild card  First name wild card  Both first and last name wild cards  Neither first or last name wild cards |