Project 2 Report

**Design Description**

Thinking about possible approaches to creating OnlineDating, I chose to use a struct named Node holding the first name, last name, and values of each list entry. Since this is a doubly linked list, I also added two type Node pointers in the Node struct which referenced the next node and previous node on lists. Using these Nodes, I then developed a doubly linked list with each Node representing a person on the list. In OnlineDating, the private members ‘head’ and ‘tail’ point to the first and last items of the linked list respectively. Using these pointers, I traversed both forward and backward through an OnlineDating object’s list to take or use the values in each Node. To keep track of the size of the list, I used a private integer member ‘m\_size’ which was incremented or decremented whenever new names were added or removed.

**Obstacles**

1. One major obstacle was developing an algorithm to effectively order the new names in the makeMatch method. Ensuring that all test cases such as lowercase letters and more were difficult to overcome as well.
2. Dealing with many multiple different values for each node particularly with understanding how to both link the nodes with ‘next’ and ‘previous’ pointers.
3. Keeping track of both head and tail pointers as well as linking nodes in both directions while also keeping track of not dereferencing null pointers.
4. Ensuring that no memory leaks occurred during testing as there were multiple occurrences where I found memory leaks due to creating new Nodes without assigning them onto the OnlineDating list.
5. Developing vigorous testing cases which tested discrete situations such as the use of empty strings or using the same names in two lists but with different values.

**Pseudocode**

**bool makeMatch(const std::string& firstName, const std::string& lastName, const OnlineType& value)**

create new node and store firstName, lastName, and value into it

if list is empty

assign head and tail to new Node with firstName and lastName and value

if not empty, set pointer to point to head of list

first check the first element of the list

if first and last names match with input, then delete n and return false

if comparing last names and the new name should go after the first element,

add new node to the start of the list and set head equal to it

integrate size

return true

if last names match and comparing first names and new name should go before first element of list

add new node to start of the list and set head equal to it

integrate size

return true

if item does not belong at start of the list

traverse through list with pointer initially set to head

for each iteration, check if the first and last name match the input, if so, delete new node and return false

if the last name at location in the list matches lastName input

begin comparing first names

if item belongs at the end of the list (the current position is on tail)

then link item to the end of the list and set tail equal to new node

integrate size

return true

if item belongs before item at end of the list (current position is on tail)

then link item right before end of the list

integrate size

return true

if the traversing pointer reaches a point where the first name has a greater value than firstName

(meaning the new value belongs before where the pointer points)

insert new node before where the traversing pointer points

integrate size

return true

else: last names don't match at a location

repeat the same process of comparing strings as done with first names but instead with last names

if the input did not belong where the traversing pointer was pointing, set the pointer to equal the next node

return false if value was not added anywhere

--------------------------------------------------------------------------------------------

**bool OnlineDating::transformMatch(const string& firstName, const**

**string& lastName, const OnlineType& value)**

Create new traversing pointer p set to head of list

while p is not at the end of the list

if p's first and last name match the input first and last names

set the value of p to equal input value

return true

otherwise, traverse p to point to next node

return false if no value was found

--------------------------------------------------------------------------------------------

**bool OnlineDating::someoneAmongMatches(const std::string& firstName, const**

**std::string& lastName) const**

Works the same way as transformMatch except only returns true or false, does not change any values

--------------------------------------------------------------------------------------------

**bool OnlineDating::blockPreviousMatch(const std::string& firstName, const**

**std::string& lastName)**

check if there the input full name matches a name already on the list

if so, check if the head element matches the name and delete the value and relink list as well as reallocate head

if not head element, traverse through list with traversing pointer(p) set to head

keep iterating p until either p is nullptr or its next node's values equal the full name input

if p is not nullptr

remove the next item from p and relink p with the removed node's next

if we are removing the tail, set the new tail to point

decrement size

return true

if no name was found, return false

--------------------------------------------------------------------------------------------

**bool OnlineDating::lookAtMatches(const std::string& firstName, const**

**std::string& lastName, OnlineType& value) const**

method works the same way as transformMatch except instead of changing the value of the node

the value parameter is changed to set to the value that the node points to

--------------------------------------------------------------------------------------------

**bool mergeMatches(const OnlineDating& odOne, const OnlineDating& odTwo,**

**OnlineDating& odJoined)**

create a temp OnlineDating object to hold the values that will merge together

create string and OnlineType values to hold the two lists values when traversing through them

create integer to mark number of times a unwanted duplicate name occurs

first, for each element in odOne

confirm the match at that element set equal to the temp strings and values created before

the add the match to the temp obj with those same values

for each element in odTwo

confirm match at that element setting the values equal to the second pairs of strings and values created

if we were unable to make a match(indicating duplicate name)

look at the match with the duplicate name in the first list and compare that value

to the value of odTwo's duplicate name

if the values are different

remove that name and integrate the unmatched number

set the odJoined to equal temp OnlineDating obj

if unmatched number is greater than 0, return false

otherwise return true (indicating no unwanted duplicate names)

--------------------------------------------------------------------------------------------

**void authenticateMatches(const std::string& fsearch, const std::string& lsearch,**

**const OnlineDating& odOne, OnlineDating& odResult)**

if odOne has noting in it, return

create temp OnlineDating obj and strings and OnlineType val to hold values of each element in odOne

if either fsearch or lsearch use the wildcard '\*'

then for each element in odOne

confirm the match setting the strings and val created before to hold that elements values

if lsearch equals the last name of the element on odOne

add that element to the temp array

if both fsearch and lsearch use '\*'

then simply set temp = odOne

otherwise

for each element in odOne

confirm the match and compare the first and last names with the input

if those values match, add match to temp

set odResult = temp;

--------------------------------------------------------------------------------------------

void tests() {

OnlineDating x;

*// empty list*

assert(x.howManyMatches() == 0); *// test for size of empty list*

assert(x.noMatches()); *// test if empty*

*//*

*// FILLING VALUES*

*//*

*// adding 7 values to list x*

x.makeMatch("Derek", "Beans", "3");

x.makeMatch("Aue", "Cro", "4");

x.makeMatch("Aue", "Oij", "1");

x.makeMatch("Henrick", "Yro", "10");

x.makeMatch("Tresh", "Yro", "5");

x.makeMatch("Charles", "Cornell", "18");

x.makeMatch("Derek", "Beans", "5");

*// adding 3 values to list y*

OnlineDating y;

y.makeMatch("Boonanza", "Hoo", "5");

y.makeMatch("Aue", "Cro", "4");

y.makeMatch("Charles", "Cornell", "18");

*// adding 2 values to list z*

OnlineDating z;

z.makeMatch("Hoho", "Haha", "5");

z.makeMatch("OIJ", "oij", "10");

*//*

*// TESTS WITH VALUES IN LIST*

*//*

assert(mergeMatches(x, y, z)); *// test to see if mergeMatches returns true with same name and same value in both OD lists*

OnlineType temp; *// variables to hold temp values*

assert(x.howManyMatches() == 6); *// test for 6 names (due to one duplicate name)*

assert(!x.noMatches()); *// test not empty*

assert(!x.makeMatch("Tresh", "Yro", "6")); *// test for same name*

assert(x.makeMatch("", "", "1")); *// test for empty strings*

assert(!y.someoneAmongMatches("Random name", "Not in list")); *// test name not in list*

assert(y.someoneAmongMatches("Boonanza", "Hoo")); *// test for someoneAmongMatches*

assert(x.transformMatch("Derek", "Beans", "5")); *// test if allows transform*

x.lookAtMatches("Derek", "Beans", temp);

assert(temp == "5"); *// test to see if transformMatch changed value and lookA5Matches works*

assert(!x.transformMatch("Dek", "Boons", "6")); *// test if not allows transform with invalid input name*

assert(x.makeOrTransform("Derek", "Beans", "7")); *// test for transform*

assert(x.makeOrTransform("Ronsol", "Zeen", "101")); *// test to see if added*

assert(x.howManyMatches() == 8); *// test for new additions*

assert(y.blockPreviousMatch("Aue", "Cro")); *// test see if removes name from list*

assert(y.howManyMatches() == 2);

assert(!y.blockPreviousMatch("Not on list", "Name")); *// test see for nothing to block*

assert(mergeMatches(x, y, z)); *// test see if returns true with Two Charles Cornells with same value*

x.transformMatch("Charles", "Cornell", "50");

assert(!mergeMatches(x, y, z)); *// test see if returns false with two Charles Cornells with different values*

assert(mergeMatches(x, z, z)); *// test for aliasing*

*// test confirm match to correctly take value from given index*

string first, last;

x.confirmMatch(1, first, last, temp);

assert(first == "Derek" && last == "Beans" && temp == "7");

*// test exchanging x and y lists*

x.tradeMatches(y);

assert(x.howManyMatches() == 2);

assert(y.howManyMatches() == 8);

*// test for authenticate matches with wildcard*

authenticateMatches("\*", "Yro", y, z);

assert(z.howManyMatches() == 2);

assert(z.someoneAmongMatches("Tresh", "Yro"));

*//*

*//COPY CONSTRUCTION AND ASSIGNMENT OPERATION*

*//*

*// creating new lists for copy construction and assignment*

OnlineDating old;

old.makeMatch("Here", "Now", "5");

old.makeMatch("Now", "Awesome", "124");

OnlineDating n;

n = old;

OnlineDating n2;

n2.makeMatch("Forever", "Away", "40");

old = n2;

*// check for copy construction of n*

assert(n.someoneAmongMatches("Now", "Awesome"));

assert(n.howManyMatches() == 2);

*// check for assignment operation on old*

assert(old.howManyMatches() == 1);

*// printing lists forwards and backwards to check outcomes*

*// and linkage*

x.dump();

y.dump();

z.dump();

}