1. I used a normal doubly linked list for this project. Each node has two pointers and three values. The two pointers are prev, and next, and the three values are firstName, lastName, and value. I also created two pointers for accessing the linked list, which are head and tail. I mainly use the head pointer to access node and its values. The tail pointer helps me when I want to add a new node at the end. I chose this design because it has the fewest elements so that I can make sure I take care of everything. It also helps make my thoughts clear, as I can pay less attention to copying in many values to the nodes.
2. My biggest obstacle was to figure out how to implement the makeMatch function. After reading the spec several times, I noticed that almost every other function is built on makeMatch so I started with this function. However, it was long and complicated: when I want to add a new node, I need to think about where to insert it. The requirement is that I need to put the last names or first names in alphabetical order. I couldn’t sort it out in my head, so I tried to draw a tree graph and list all possibilities as branches. I decided to process the first node individually so that I don’t need to care if the linked list is empty or no. Then I used a loop to find where to insert. I set the conditions to “p->next != nullptr” and “lastName > p->lastName”, so I know the loop stops for one of the two reasons. I went on to process the two conditions separately with if-else statements and process with more possibilities under the two big coditions. The hardest part was that I needed to think about what I should do when the insert needs to be in the very front or at the very end, because that means I also needed to update the head and tail pointer.
3. **makeMatch:**

if the full name is already in the list,

return false

if not in the list,

if is the first match in the list,

make a new match and let head and tail point to it

if is not the first,

proceed to where the new match needs to be inserted while p->next != nullptr and lastName > p->lastName

if needs to be insert to the very front,

make a new match, copy in the name and value, and let head point to it

if needs to be inserted at the very end,

make a new match, copy in the name and value, and let tail point to it

if neither,

make a new match, copy in the name and value, and take care of

its next and prev pointer as well as those of the previous match

and the next match

**transformMatch:**

if the full name is not in the list,

return false

if the full name is already in the list,

find the match with a while loop

copy the passed in value into the value of the match

**makeOrTransform:**

if the full name is not in the list,

make a new match and insert it into the list

if the full name is already in the list,

find the match with a while loop

copy the passed in value into the value of the match

**blockPreviousMatch:**

if the full name is not in the list,

return false

if the full name is in the list,

find the match with a while loop

relink the previous match and next match and delete this match

**someoneAmongMatches:**

loop through the matches until the match is found,

return true

if no such match,

return false

**lookAtMatches:**

if no such match,

return false

else

loop through the matches and find the match with the full name

set the value parameter to the value of this match

return true

**confirmMatch:**

if the i value is invalid

return false

else

use a for loop to find the match at position i

set firstName equal to the firstName of this match

set lastName equal to the lastName of this match

set value equal to the value of this match

return true

**tradeMatches:**

create a temp pointer that points to the head of this linked list

let the head of this linked list point to the head of the other linked list’s head

let the head of the other point to where temp points to

**mergeMatches:**

repeatedly, copy the matches from odOne to odJoined

repeatedly, get the matches from odTwo and check if they are in odJoin

if it is not in odJoined,

make a match and add it to odJoined

if it is in odJoined already,

if the values match, do nothing

else, block previous match

return if the merge was successful

**authenticateMatches:**

if odResult has elements in it,

use a while loop to delete everything

else

if fsearch is \* and:

1. lsearch is not \*,

find all matches with this last name and add them to odResult

1. lsearch is also \*,

copy all matches into odResult

if fsearch is not \* and:

1. lsearch is not \*,

find all matches with this full name and add them to odResult

1. lsearch is \*,

find all matches with this first name and add them to odResult

4. Test Cases:

**void** test1(){

OnlineDating test;

OnlineDating result;

string firstName, lastName;

OnlineType value;

//noMatches

assert(test.noMatches());

//makeMatch

//make a first match

//make a match at the end

//make a match in the middle

//make a match in the very front

test.makeMatch("Cobey", "C", 35 );

test.makeMatch("Dan", "H", 38);

test.makeMatch("Dan", "V", 44);

test.makeMatch("Dion", "V", 45);

test.makeMatch("Dion", "V", 45);

test.makeMatch("", "A", 2);

test.makeMatch("", "" ,5);

//howmanyMatches

//make a new match and test again

//try to make a repeated match, the number should not increase

assert(test.howManyMatches() == 6);

test.makeMatch("Bob", "Dylan", 16);

assert(test.howManyMatches() == 7);

test.makeMatch("Bob", "Dylan", 16);

assert(test.howManyMatches() == 7);

//confirm match

test.confirmMatch(0, firstName, lastName, value);

assert(firstName == "" && lastName == "" && value == 5);

test.confirmMatch(3, firstName, lastName, value);

assert(firstName == "Bob" && lastName == "Dylan" && value == 16);

//transformMatch

test.transformMatch("Bob", "Dylan", 20);

test.confirmMatch(3, firstName, lastName, value);

assert(firstName == "Bob" && lastName == "Dylan" && value == 20);

test.transformMatch("Dion", "V", 66);

test.confirmMatch(6, firstName, lastName, value);

assert(firstName == "Dion" && lastName == "V" && value == 66);

//Assignment Operator

result = test;

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

{

string first;

string last;

**int** val;

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

cerr << first << " " << last << " " << val << endl;

}

//copy constructor

OnlineDating copy(test);

**for** (**int** n = 0; n < copy.howManyMatches(); n++)

{

string first;

string last;

**int** val;

copy.confirmMatch (n, first, last, val);

cerr << first << " " << last << " " << val << endl;

}

//someoneAmongMatches

assert(test.someoneAmongMatches("", "A"));

assert(test.someoneAmongMatches("Bob", "Dylan"));

assert(test.someoneAmongMatches("", ""));

assert(test.someoneAmongMatches("Dion", "V"));

//lookAtMatches

test.lookAtMatches("", "", value);

assert(value == 5);

test.lookAtMatches("Cobey", "C", value);

assert(value == 35);

test.lookAtMatches("", "A", value);

assert(value == 2);

//tradeMatches

OnlineDating trade;

trade.makeMatch ("Kawhi", "Leonard", 2);

trade.makeMatch ("Paul", "George", 13);

trade.makeMatch ("Ivica", "Zubac", 40);

trade.makeMatch ("Reggie", "Jackson", 1);

trade.makeMatch ("Patrick", "Beverley", 21);

trade.tradeMatches(test);

**for** (**int** n = 0; n < test.howManyMatches(); n++)

{

string first;

string last;

**int** val;

test.confirmMatch (n, first, last, val);

cerr << first << " " << last << " " << val << endl;

}

//mergeMatches

OnlineDating merge;

mergeMatches(test, test, merge);

assert(mergeMatches(test, test, merge));

**for** (**int** n = 0; n < merge.howManyMatches(); n++)

{

string first;

string last;

**int** val;

merge.confirmMatch (n, first, last, val);

cerr << first << " " << last << " " << val << endl;

}

mergeMatches(test, trade, merge);

assert(mergeMatches(test, trade, merge));

**for** (**int** n = 0; n < merge.howManyMatches(); n++)

{

string first;

string last;

**int** val;

merge.confirmMatch (n, first, last, val);

cerr << first << " " << last << " " << val << endl;

}

trade.makeMatch("Paul", "George", 13);

mergeMatches(test, trade, merge);

assert(mergeMatches(test, trade, merge));

**for** (**int** n = 0; n < merge.howManyMatches(); n++)

{

string first;

string last;

**int** val;

merge.confirmMatch (n, first, last, val);

cerr << first << " " << last << " " << val << endl;

}

trade.transformMatch("Paul", "George", 31);

assert(!mergeMatches(test, trade, merge));

**for** (**int** n = 0; n < merge.howManyMatches(); n++)

{

string first;

string last;

**int** val;

merge.confirmMatch (n, first, last, val);

cerr << first << " " << last << " " << val << endl;

}

//blockPrevious

OnlineDating test2;

test2.makeMatch("Cobey", "C", 35 );

test2.makeMatch("Dan", "H", 38);

test2.makeMatch("Dan", "V", 44);

test2.makeMatch("Dion", "V", 45);

test2.makeMatch("Dion", "V", 45);

test2.makeMatch("", "A", 2);

test2.makeMatch("", "" ,5);

//block "" "", the head

test2.blockPreviousMatch("", "");

//it should not be in the list anymore, the size of the list should decrease as well

assert(!test2.someoneAmongMatches("", "") && test2.howManyMatches() == 5);

//block "Dion" "V", the tail

test2.blockPreviousMatch("Dion", "V");

//it should not be in the list anymore, the size of the list should decrease as well

assert(!test2.someoneAmongMatches("Dion", "V") && test2.howManyMatches() == 4);

//block "Dion" "H", in the middle of the list

test2.blockPreviousMatch("Dan", "H");

//it should not be in the list anymore, the size of the list should decrease as well

assert(!test2.someoneAmongMatches("Dan", "H") && test2.howManyMatches() == 3);

**for** (**int** n = 0; n < test2.howManyMatches(); n++)

{

string first;

string last;

**int** val;

test2.confirmMatch (n, first, last, val);

cerr << first << " " << last << " " << val << endl;

}

}

**void** test2(){

//authenticateMatches

OnlineDating test;

test.makeMatch("Cobey", "C", 35 );

test.makeMatch("Dan", "H", 38);

test.makeMatch("Dan", "V", 44);

test.makeMatch("Dion", "V", 45);

test.makeMatch("Dion", "V", 45);

test.makeMatch("", "A", 2);

test.makeMatch("", "" ,5);

test.makeMatch("a", "a" ,25);

OnlineDating result;

result.makeMatch("Jerry", "Liu", 2333);

result.makeMatch("name", "random", 10);

//given a non-empty list, see if the result contains only what we want

authenticateMatches("a", "a", test, result);

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

{

string first;

string last;

**int** val;

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

cerr << first << " " << last << " " << val << endl;

}

}

**int** main()

{

test1();

test2();

cerr << "All tests passed " << endl;

}