For this particular project I choose a double linked list without a tail pointer, this implementation does have next and previous pointers within the struct. The reason being is because I am able to move in two different directions and compare two sets of first names, last names at once. This was extremely helpful with the function makematch. My struct is defined in the private sector of my class, the reason being is because we were not allowed to implement it in the public interface or outside the interface of the class. This struct contains two strings each for first and last name, a Online type variable which I choose to be an int for the majority of my tests but could also be a string. It contains a pointer next and a pointer prev of type Node. In the private interface I also have a pointer of type Node to indicate the head (beginning) of the list.

Notable obstacles

The make match function was perhaps the most pointer heavy functions I have ever made, there are countless conditions of if it's in this position do this or if it's at the beginning do this. What I did was think of the most cases I could think of. Draw it out on my whiteboard and make sure the pointers were in the correct spot. Another function I also struggled with was the Block Previous Match for similar reasons. It would confuse me where the pointers should go that way the list remained continuous and I didn’t accidently cause a memory leak. Lastly the authenticatematches was a weird function, for some reason when I passed the lists by reference the next pointer address would become corrupted, I tried about 3 different methods but I was unable to make it work. It still complies and stores the important information but the next pointers are corrupted (actually only the last one hence I cannot ask for the size of this list).

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

**lastName, const OnlineType& value)**

{

if (someoneAmongMatches(firstName, lastName)) // if this returns true then there is a match

return false; //we don’t need to add anything

else if (head == nullptr)/ /name is not in list and list is empty

{

add values and pointers to node

return true;

}

else // list has contents

{

Node\* ptr = head;

Node\* p = head; // twio pi=ointers initialized at the head

while (ptr != nullptr) //keep iterating thorough the list

{

if (ptr->lastName < lastName)// if current last name is smaller than new Lastname

{

if (ptr->next == nullptr) //we are at the end of the list

{

creating and storing info in new node

return true;

}

else if (ptr->next->lastName > lastName) // if next lastname is bigger than new one

{

creating and storing info in new node

return true;

}

else

p = ptr = ptr->next; //move one node forward

continue;

}

else if (ptr->lastName == lastName) //last names match now we need to see which one precedes each other in first names

{

if (ptr->next != nullptr) //if list is not at its end keep iterating

{

while (ptr->next != nullptr) iterate until end of list

{

if (ptr->firstName < firstName && ptr->next->lastName> lastName) // if current first name is smaller and next last name is bigger then store new name here

{

creating and storing info in new node

return true;

}

else if (ptr->firstName > firstName && ptr->prev->firstName < firstName) //current first name is bigger than input first name and smaller than previous first name

{

creating and storing info in new node

return true;

}

else //keep it moving

{

p = ptr = ptr->next; //move node

continue;

}

}

}

else if (ptr->firstName > firstName) //current first name is bigger than input first name and smaller than previous first name

{

// creating and storing info in new node

return true;

}

else if (ptr->firstName < firstName //if current first name is smaller than firstname

{

creating and storing info in new node

return true;

}

}

else if (ptr->lastName > lastName) //if current last name is bigger than new lastname

{

if (ptr->next == nullptr || ptr->prev == nullptr)// we are at the end or beginning of the list, place new node in front

{

creating and storing info in new node

head = temp;

return true;

}

else if (ptr->lastName > lastName && ptr->prev->lastName < lastName) //current last name is bigger than input last name and smaller than previous last name)

{

// creating and storing info in new node

return true;

}

}

else

p = ptr = ptr->next; //keep going

}

}

return true; // I only added this because it was givig me an error in g32

}

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

**std::string& lastName)**

{

if (someoneAmongMatches(firstName, lastName)) // true if it is in the list

{

Node\* ptr = head; //node pointer at beginning of list

while (ptr != nullptr) //iterate through the whole list

{

if (ptr->firstName == firstName && ptr->lastName == lastName) //if full name matches //break

{

break;

}

else

{

ptr = ptr->next; //if not full match keep iterating

}

}

if (ptr->next == nullptr && ptr->prev == nullptr) if it’s the last node

{

head = nullptr;

delete ptr;

ptr = nullptr;

return true;

}

else if (ptr->prev == nullptr) If it’s the first node

{

head = ptr->next; change pointers to allow head to be replaced

ptr->next->prev = nullptr;

}

else if (ptr->next == nullptr) if it’s the last node

{

ptr->prev->next = nullptr; second to last node should now point to nullptr

ptr->next = nullptr;

}

else // somewhre in the middle

{

ptr->prev->next = ptr->next; //switch the avlues of the previous and next node

ptr->next->prev = ptr->prev;

}

delete ptr; //delete the node that is pointed by the pointer

return true; // return true;

}

else // it is not in the list

return false;

}

**/// NON-CLASS Functions/////////**

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

{

bool Non\_C = true; //flag that will keep track if the two lists had the same full name but different values

if (odJoined.noMatches() != true) //if non empty it will return false and go into the loop, will do this until it is empty

{

OnlineDating EMT;

odJoined = EMT; //assigned an emepty one

}

if (odJoined.noMatches()) // true if odJoined is empty

{ // if its empty then I can add all the elements of odOne in there and later on compare them to those in odTwo

for (int j = 0; j < odTwo.howManyMatches(); j++) //iterate through the size of it

{

std::string first;

std::string last;

OnlineType data;

odTwo.confirmMatch(j, first, last, data);

if (odOne.someoneAmongMatches(first, last)) // odOne also contains the full name next step is to check the data type

//if identical copy it to odJoined else dont copy (aliasing)

{

OnlineType data\_2;

odOne.lookAtMatches(first, last, data\_2);

if (data == data\_2) //have the same OnlineType value

{

odJoined.makeMatch(first, last, data); // add full name and value to odjoined

}

else // Onlinetype values differ DONT COPY!!!!!!!!!!!!!!

{

Non\_C = false; //flag that shows that copy was avoided

}

}

else

{

odJoined.makeMatch(first, last, data); // add it to Odjoined

}

}

for (int i = 0; i < odOne.howManyMatches(); i++) //repeat the porcess for thr other list

{

std::string first;

std::string last;

OnlineType data;

odOne.confirmMatch(i, first, last, data);

if (odJoined.someoneAmongMatches(first, last)) // odJoined already contains the full name we dont need to add it again,

//want to make sure aliasing doesnt happen

{ //if first part returns true then it means that both lists have the same copy with same value

continue;

}

else if (odTwo.someoneAmongMatches(first, last)) // if this returns true it means that its in odtwo but not odjoined hence they have different values

{

continue;

}

else

{

odJoined.makeMatch(first, last, data); // add it to Odjoined

}

}

}

else // odjoined is not empty, need to empty it

{

mergeMatches(odOne, odTwo, odJoined); // call it again, for safety reasons

}

return Non\_C; //return true or false depending on flag;

}

**void authenticateMatches(const std::string& fsearch, const std::string& lsearch, const OnlineDating& odOne, OnlineDating& odResult)**

{

OnlineDating temp; // this will be a copy

odResult.tradeMatches(temp); //exchange with temp

bool first\_asterisk = (fsearch == "\*");

bool last\_asterisk = (lsearch == "\*");

if (odOne.someoneAmongMatches(fsearch, lsearch))

{

for (int k = 0; k < odOne.howManyMatches(); k++)

{

std::string first;

std::string last;

OnlineType data;

odOne.confirmMatch(k, first, last, data);

if (fsearch == first) // if true, no need to check first names

{

if (lsearch == last) // if true, no need to change anything

{

odResult.makeMatch(first, last, data); //add to list

continue;

}

}

}

}

else if (first\_asterisk == true && last\_asterisk == true) // copy the whole linkedList

{

temp = odOne;

odResult.tradeMatches(temp);

return;// check first names

}

for (int k = 0; k < odOne.howManyMatches(); k++)

{

std::string first;

std::string last;

OnlineType data;

odOne.confirmMatch(k, first, last, data);

if (lsearch == last)

{

if (first\_asterisk)

{

odResult.makeMatch(first, last, data); //add it to list

continue;

}

}

else if (first == fsearch)

{

if (last\_asterisk) // if true

{

odResult.makeMatch(first, last, data);

continue;

}

}

}

return;

}

**TESTCASES**

FOR ALL TESTCASES I SWITCHED ONLINETYPE TO INT!!!!!!

OnlineDating UCLA; // constructtor

OnlineDating USC;//constructor

assert(UCLA.howManyMatches() == 0); //empty list

assert(UCLA.noMatches());m// empty list

assert(UCLA.makeMatch("Aero", "Space", 11)); //add to list beginning

assert(UCLA.makeMatch("Aero", "Nautics", 22)); //sort out same first name

assert(!UCLA.makeMatch("Aero", "Space", 11)); //duplicate

assert(UCLA.makeMatch("X", "Space", 121)); //same last name diff FN

assert(UCLA.howManyMatches() == 3); //check size

assert(UCLA.makeMatch("EE", "EE", 9)); //beginning

assert(UCLA.howManyMatches() == 4); //check size

assert(UCLA.makeMatch("EEE", "EE", 17)); //middle insertion

assert(UCLA.howManyMatches() == 5); //check size

assert(UCLA.transformMatch("Aero", "Space", 87));//change value, ret true

assert(!UCLA.transformMatch("Mech", "Eng", 00)); // not in list return false

assert(UCLA.makeOrTransform("Mech", "Eng", 00)); // adds it return true

assert(UCLA.someoneAmongMatches("Mech", "Eng")); //checks if it was added

assert(UCLA.makeOrTransform("Mech", "Eng", 99)); // return true and changes val

assert(UCLA.blockPreviousMatch("Mech", "Eng")); // deletes it and return true

assert(!UCLA.someoneAmongMatches("Mech", "Eng")); //checks if it was deleted

OnlineType n; //to store value

assert(UCLA.lookAtMatches("Aero", "Space", n));// return true since its in the list

assert(n == 87); // checks if it extracted the value correctly

string F; //to store value

string L; //to store value

OnlineType V; // to store value

assert(UCLA.confirmMatch(0, F, L, V)); // returns true since its a valid index

assert(F == "EE" && L == "EE" && V == 9);// checks if it extracted the right values

assert(!UCLA.confirmMatch(18, F, L, V));// returns false since its not a valid index

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

{

string first;

string last; // obtain the information of the node and confirm it

int val;

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

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

}

USC = UCLA; //assignment operator deep copy

OnlineDating okCupid;

assert(okCupid.makeMatch("Lauren", "U", 23)); //add to it

assert(okCupid.makeMatch("James", "H", 29)); //add to it

assert(okCupid.howManyMatches() == 2); //check size

string first, last;

int a;

assert(okCupid.confirmMatch(0, first, last, a) && a == 29); //confrim contents

assert(okCupid.confirmMatch(1, first, last, a) && (first ==

"Lauren" && a == 23)); //confirm contents

UCLA = okCupid;//assignment operator and copy constructor

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

{

string first;

string last;

int val;

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

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

} //check that bottom and top for loop output the same thing

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

{

string first;

string last;

int val;

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

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

}

OnlineDating lal; //declare it

// For an empty list:

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

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

assert(!lal.blockPreviousMatch("Marc", "Gasol")); //nothing to erase

OnlineDating One;

assert(One.makeMatch("Dwight", "Howard", 39)); //adding to it

assert(One.makeMatch("Lebron", "James", 23)); //adding to it

assert(One.makeMatch("Javale", "McGee", 7)); //additng to iy

assert(One.howManyMatches() == 3); //checking size

OnlineDating Two;

assert(Two.makeMatch("Lebron", "James", 23)); //addint to it

assert(Two.makeMatch("Dennis", "Schroeder", 17)); //addint to it

assert(Two.howManyMatches() == 2); //checking size

OnlineDating Three; //declaring

mergeMatches(One, Two, Three); // should return Lebron James 23 since they both have the same value

assert(Three.howManyMatches() == 4); //check size

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

{

string first;

string last;

int val;

Three.confirmMatch(n, first, last, val); //check contents and confirm

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

}

OnlineDating Four; //initializes new class

assert(Four.makeMatch("Lebron", "James", 6)); //adds to it

assert(Four.makeMatch("Dennis", "Schroeder", 17));//addts to it

assert(Four.howManyMatches() == 2); //checks size

mergeMatches(Two, Four, Three); //merges

cout << endl << endl; //spacing

assert(Three.howManyMatches() == 1); //checks if size is correct

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

{

string first;

string last;

int val;

Three.confirmMatch(n, first, last, val); //outputs the contents

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

}

OnlineDating CSU; //initializes new class

assert(CSU.makeMatch("Aero", "Space", 11)); //add to it

assert(CSU.makeMatch("Aero", "Nautics", 22)); // add to it

assert(!CSU.makeMatch("Aero", "Space", 11)); //duplicate

assert(CSU.makeMatch("X", "Space", 121)); //same last name diff FN

authenticateMatches("Aero", "\*", CSU, One); //authenticate word Aero

assert(One.howManyMatches() == 2); //checks size after edits

std::string First; //store string

std::string Last;

OnlineType val;

One.confirmMatch(0, First, Last, val);

assert(First == "Aero" && Last == "Nautics" && val == 22); //checks if it was done correctly

One.confirmMatch(1, First, Last, val);

assert(First == "Aero" && Last == "Space" && val == 11); //checks second Node values

OnlineDating coffeeMeetsBagel;

OnlineDating RES;

assert(coffeeMeetsBagel.makeMatch("Cobey", "C", 35));

assert(coffeeMeetsBagel.makeMatch("Dan", "H", 38));

assert(coffeeMeetsBagel.makeMatch("Dan", "V", 44));

assert(coffeeMeetsBagel.makeMatch("Dion", "V", 45)); //assemble a List

authenticateMatches("Dan", "\*", coffeeMeetsBagel, RES); //tries to edit it

RES.confirmMatch(0, First,Last,val);

assert(First == "Dan" && Last == "H" && val == 38); //checks if it obtained correct valuesfor 0

RES.confirmMatch(1, First, Last, val);

assert(First == "Dan" && Last == "V" && val == 44); //checks if it obtained correct values for 1

CHECKTYPE(&OnlineDating::operator=, OnlineDating & //this was given to us

(OnlineDating::\*)(const OnlineDating&));

CHECKTYPE(&OnlineDating::noMatches, bool

(OnlineDating::\*)() const);

CHECKTYPE(&OnlineDating::howManyMatches, int

(OnlineDating::\*)() const);

CHECKTYPE(&OnlineDating::makeMatch, bool (OnlineDating::\*)

(const std::string&, const std::string&, const

OnlineType&));

CHECKTYPE(&OnlineDating::transformMatch, bool

(OnlineDating::\*)(const std::string&, const std::string&,

const OnlineType&));

CHECKTYPE(&OnlineDating::makeOrTransform, bool

(OnlineDating::\*)(const std::string&, const std::string&,

const OnlineType&));

CHECKTYPE(&OnlineDating::blockPreviousMatch, bool

(OnlineDating::\*)

(const std::string&, const std::string&));

CHECKTYPE(&OnlineDating::someoneAmongMatches, bool

(OnlineDating::\*)(const std::string&, const std::string&)

const);

CHECKTYPE(&OnlineDating::lookAtMatches, bool

(OnlineDating::\*)

(const std::string&, const std::string&, OnlineType&)

const);

CHECKTYPE(&OnlineDating::confirmMatch, bool (OnlineDating::\*)

(int, std::string&, std::string&, OnlineType&)

const);

CHECKTYPE(&OnlineDating::tradeMatches, void

(OnlineDating::\*)(OnlineDating&));

CHECKTYPE(mergeMatches, bool (\*)(const OnlineDating&, const

OnlineDating&, OnlineDating&));

CHECKTYPE(authenticateMatches, void (\*)(const std::string&,

const std::string&, const OnlineDating&, OnlineDating&));

I also tested for memory leak but I found it redundant to place that code here since we were given that code by the Professor/Ta’s.