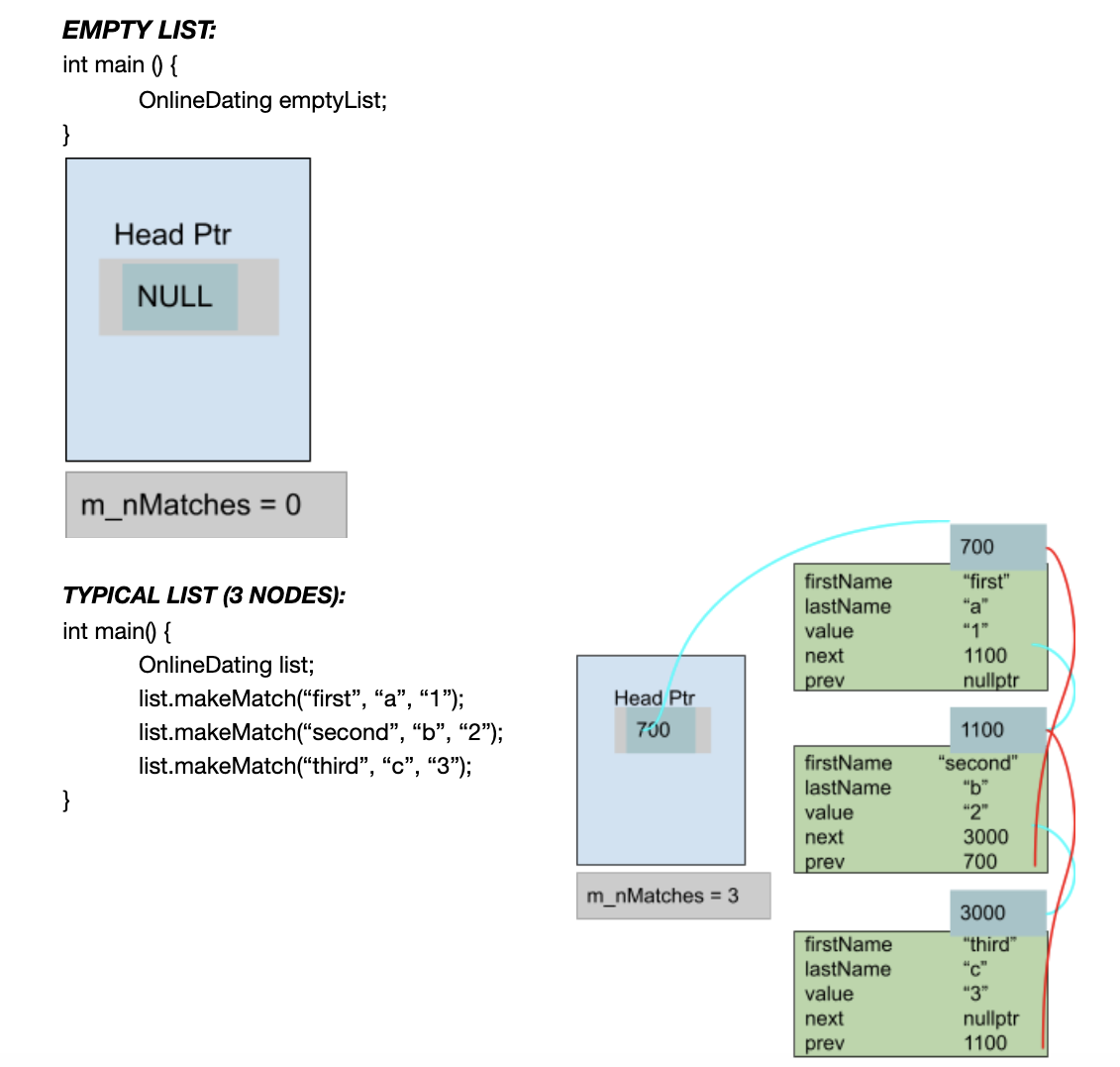
**PROJECT 2 REPORT**

***IMPLEMENTATION:***

*//* My doubly linked list is implemented with two private member variables, including a pointer to a head node and an int variable m\_nMatches that keeps track of the number of elements (nodes) in each list. These nodes each have 3 public data variables that contain the first name, last name, and value variable of each data element, as well as pointers to a “prev” and “next” node.

**

***OBSTACLES***

|  |
| --- |
| The first obstacle I overcame was trying to account for every possible test case within my makeMatch function. After creating separate loops to account for inserting in the front or the back of the node, I manually drew out different situations in which a node would be inserted between two other nodes. This came out to four different cases through my calculations, in which I determined what conditions were required for the node to be placed appropriately. I used these conditions as part of an “if” statement that determined when my cutoff for pointer traversal would be. This cutoff should always come at the pointer before insertion. Using this method allowed me to always connect a new node between two valid nodes of the correct alphabetical conditions. |
| The second obstacle I overcame was determining how to link nodes within the copy constructor. By drawing a picture of the OnlineDating linked list, I was able to determine when and where to assign each nodes’ next and prev pointers. |
| The third obstacle I overcame was isolating a memory leak while testing multiple functions in linux. I isolated the leak eventually by narrowing down my int main() function on the linux compiler, so I realized my leak was rising from my blockPreviousMatch function. That made me realize that I needed to delete each node, and not just re-link its surrounding nodes. |
| The fourth and final obstacle I overcame was organizing and compressing my code. I made my code more legible by replacing redundant code with more efficient loops, replacing traversals with calls to member functions like someoneAmongMatches, and maintaining consistency with the names for my pointers. |

***PSEUDOCODE***

|  |
| --- |
| **bool OnlineDating::makeMatch(firstName, lastName, value)**  **{**  **if (the full name is already in the list)**  **return false;**  **++m\_nMatches;**  **allocate a new node**  **set new node firstName;**  **set new node lastName;**  **set new node value;**  **if (the list is empty or the node should go first)**  **{**  **insert new node to front;**  **return true;**  **}**  **if (the new node should be appended at the back)**  **{**  **append new node to back;**  **return true;**  **}**  **Node \*b = head;**  **increment b to point at the point above insertion**  **link new node to b and b->next;**  **return true;**  **}**  **bool OnlineDating::blockPreviousMatch(firstName, lastName)**  **{**  **if (the full name is already in the list)**  **return false;**    **--m\_nMatches;**  **if (the full name matches the head of the list)**  **re-link second node in list**  **delete current head;**  **Node \*p = head;**  **increment p to either the last node or the first matching node;**    **if (the last node matches the full name)**  **unlink the last node from the list**  **if (a middle value matches the first name)**  **link the nodes surrounding it to each other**  **delete the target node;**  **return true;**  **}**  **bool mergeMatches (odOne, odTwo, adjoined)**  **{**  **set default return value to true;**  **make empty OnlineDating listCombo to hold results;**  **make empty OnlineDating badMatch to keep track of matching full names with different values;**  **make empty OnlineDating goodMatch to keep track of matching full names with same values;**    **// create strings to be filled with confirmMatch function**  **string first2;**  **string last2;**  **string val2;**  **// check every element of odTwo for matches with odOne**  **for (every element of odTwo)**  **{**  **fill first2, last2, and val2 with the values of the kth node of odTwo;**  **string checkVal;**  **if (the full name in the kth node of odTwo matches any full name in odOne)**  **{**  **if (the values of these nodes match)**  **add this element to the goodMatch list;**  **else if (the values of these nodes do not match)**  **{**  **add this element to the badMatch list;**  **change returnValue to false;**  **}**  **}**  **}**  **for (every element of odOne)**  **{**  **fill first, last, and val with the values in the ith element of odOne;**    **if (the ith element of odOne does not match any element in badMatch)**  **add this element to the listCombo (results) list;**  **}**  **for (every element of odTwo)**  **{**  **fill first, last, and val with the values in the ith element of odOne;**  **if (the ith element of odOne does not match any element in badMatch OR goodMatch)**  **add this element to the listCombo (results) list;**  **}**  **trade listCombo with odJoined;**  **return returnValue;**  **}**  **void authenticateMatches(search, search, odOne, odResult)**  **{**  **OnlineDating list; // create an empty list to be filled with results**  **if (both search terms are “wild cards”)**  **call assignment operator to make a copy of odOne and place it into list;**  **else**  **{**  **for (every element of odOne)**  **{**  **string first;**  **string last;**  **string val;**  **fill first, last, and val with the values in the ith element of odOne;**  **if (firstName search was wild card)**  **add every odOne element with the given last name into list;**  **else if (lastName search was wild card)**  **add every odOne element with the given first name into list;**  **else if (both first and last name were searched)**  **add odOne element matching the given full name into list;**  **}**  **}**  **swap odResult with the filled list;**  **}** |

***TEST CODE***

|  |
| --- |
| #include "OnlineDating.h"  #include <iostream>  #include <cassert>  **using** **namespace** std;  **typedef** std::string OnlineType;  // TESTING  **int** main()  {  // Default Constructor  OnlineDating emptyList;    // Tests for an empty list  assert(emptyList.howManyMatches() == 0); // howManyMatches  assert(emptyList.noMatches()); // noMatches  assert(!emptyList.blockPreviousMatch("empty", "list")); // blockPreviousMatch (nothing to block)    // Default Constructor, Copy Constructor, Assignment Operator, and makeMatch alphabetical order  // each element of list one has a value equal to the position that it should be sorted into  OnlineDating list1; // DC  list1.makeMatch("b", "d", "13");  list1.makeMatch("b", "a", "1");  list1.makeMatch("d", "a", "3");  list1.makeMatch("c", "c", "10");  list1.makeMatch("d", "b", "7");  list1.makeMatch("a", "d", "12");  list1.makeMatch("a", "b", "4");  list1.makeMatch("b", "c", "9");  list1.makeMatch("b", "b", "5");  list1.makeMatch("d", "d", "15");  list1.makeMatch("c", "a", "2");  list1.makeMatch("c", "b", "6");  list1.makeMatch("a", "c", "8");  list1.makeMatch("a", "a", "0");  list1.makeMatch("d", "c", "11");  list1.makeMatch("c", "d", "14");  assert(list1.howManyMatches() == 16);  list1.makeMatch("a", "a", "0");  assert(list1.howManyMatches() == 16); // confirms that makeMatch won't insert a matching full name  // confirms that the list was sorted into the correct order  string first;  string last;  string value;  list1.confirmMatch(0, first, last, value);  assert(value == "0");  list1.confirmMatch(5, first, last, value);  assert(value == "5");  list1.confirmMatch(10, first, last, value);  assert(value == "10");  list1.confirmMatch(15, first, last, value);  assert(value == "15");  OnlineDating list2;  list2.makeMatch("a", "a", "0");  list2.makeMatch("b", "a", "1");  list2.makeMatch("a", "b", "2");  list2.makeMatch("b", "b", "3");  assert(list2.howManyMatches() == 4);  // testing copy constructor  OnlineDating list3 = list1;  assert(list3.howManyMatches() == 16); // confirms that list3 is a full copy of list 1  OnlineDating list4 = list1; // making list4 for later use    // tests that the lists match  // list3 is currently a copy of the 16-element list that was originally list1  list3.confirmMatch(0, first, last, value);  assert(value == "0");  list3.confirmMatch(5, first, last, value);  assert(value == "5");  list3.confirmMatch(10, first, last, value);  assert(value == "10");  list3.confirmMatch(15, first, last, value);  assert(value == "15");      // testing assignment operator  list1 = list2;  assert(list1.howManyMatches() == 4); // confirms that list1 is a full copy of list2  // tests that the lists match (assignment operator working)  // list1 is currently a copy of the 4-element list that was originally list 2  list1.confirmMatch(0, first, last, value);  assert((first == "a" && last == "a" && value == "0"));  list1.confirmMatch(1, first, last, value);  assert((first == "b" && last == "a" && value == "1"));  list1.confirmMatch(2, first, last, value);  assert((first == "a" && last == "b" && value == "2"));  list1.confirmMatch(3, first, last, value);  assert((first == "b" && last == "b" && value == "3"));    // testing assignment operator  list2 = list3;  assert(list2.howManyMatches() == 16); // confirms that list2 is a full copy of list3      // testing transformMatch on the first, last, and a middle value  assert(list2.transformMatch("a", "a", "100"));  assert(list2.confirmMatch(0, first, last, value) && value == "100");  assert(list2.transformMatch("d", "d", "200"));  assert(list2.confirmMatch(15, first, last, value) && value == "200");  assert(list2.transformMatch("b", "b", "300"));  assert(list2.confirmMatch(5, first, last, value) && value == "300");  // testing transformMatch on names not in the list  assert(!list2.transformMatch("e", "e", "300"));  assert(!list2.transformMatch("e", "a", "300"));    // testing makeOrTransform on the first, last, and a middle value  // testing the transform function  assert(list1.makeOrTransform("a", "a", "100"));  assert(list1.confirmMatch(0, first, last, value) && value == "100");  assert(list1.makeOrTransform("b", "a", "200"));  assert(list1.confirmMatch(1, first, last, value) && value == "200");  assert(list1.makeOrTransform("a", "b", "300"));  assert(list1.confirmMatch(2, first, last, value) && value == "300");  assert(list1.makeOrTransform("b", "b", "400"));  assert(list1.confirmMatch(3, first, last, value) && value == "400");  // testing the make function  assert(list1.makeOrTransform("c", "c", "500"));  assert(list1.confirmMatch(4, first, last, value) && value == "500");  // makeOrTransform should always return true  assert(list1.makeOrTransform("", "", ""));  // testing blockPreviousMatch on the first, last, and a middle value  // testing someoneAmongMatches on non-matching values  // list 3 is still a copy of the 16-element list that was originally list1  // blocking one match should set the value and the position parameter off by one  assert(list3.blockPreviousMatch("a", "a"));  assert(!list3.someoneAmongMatches("a", "a") && list3.confirmMatch(0, first, last, value) && value == "1");  assert(list3.howManyMatches() == 15); // checks that m\_nMatches was correctly decremented  assert(list3.blockPreviousMatch("d", "d"));  assert(!list3.someoneAmongMatches("d", "d") && list3.confirmMatch(13, first, last, value) && value == "14");  assert(list3.howManyMatches() == 14); // checks that m\_nMatches was correctly decremented  assert(list3.blockPreviousMatch("b", "b"));  assert(!list3.someoneAmongMatches("b", "b") && list3.confirmMatch(3, first, last, value) && value == "4");  assert(list3.howManyMatches() == 13); // checks that m\_nMatches was correctly decremented  // nothing to block (confirms returns false)  assert(!list3.blockPreviousMatch("f", "f"));    // testing lookAtMatches on first, last, middle, and nonexistent values  // list4 is a copy of the 16-element list that was originally list1  string testValue = "no matches";  assert(!list4.lookAtMatches("e", "e", testValue) && testValue == "no matches"); // no change made to value  assert(list4.lookAtMatches("a", "a", testValue) && testValue == "0");  assert(list4.lookAtMatches("b", "b", testValue) && testValue == "5");  assert(list4.lookAtMatches("c", "c", testValue) && testValue == "10");  assert(list4.lookAtMatches("d", "d", testValue) && testValue == "15");    // testing confirmMatch // CONFIRM MATCH HAS BEEN TESTED IN THE ABOVE TESTS    // testing tradeMatches  OnlineDating list5;  list5.makeMatch("a", "a", "0");  list5.makeMatch("b", "b", "1");  assert(list5.howManyMatches() == 2 && list4.howManyMatches() == 16);  list4.tradeMatches(list5);  assert(list5.howManyMatches() == 16 && list4.howManyMatches() == 2); // confirms that m\_nMatches was traded  // confirms that list4 now contains list5  assert(list4.confirmMatch(0, first, last, value) && first == "a" && last == "a" && value == "0");  assert(list4.confirmMatch(1, first, last, value) && first == "b" && last == "b" && value == "1");  // confirms that list5 now contains list 4  assert(list5.confirmMatch(15, first, last, value) && first == "d" && last == "d" && value == "15");  // testing mergeMatches  OnlineDating listOne;  listOne.makeMatch("a", "a", "0");  listOne.makeMatch("b", "b", "0"); // should be sorted into "badMatches", or full name match with diff val  listOne.makeMatch("c", "c", "3"); // should be sorted into "goodMatches", or full name match with same val  listOne.makeMatch("d", "d", "3");    OnlineDating listTwo;  listTwo.makeMatch("e", "e", "0");  listTwo.makeMatch("f", "f", "0");  listTwo.makeMatch("b", "b", "3"); // should be sorted into "badMatches", or full name match with diff val  listTwo.makeMatch("c", "c", "3"); // should be sorted into "goodMatches", or full name match with same val    OnlineDating mergedList;  // should contain aa0, cc3, dd3, ee0, ff0    assert(!mergeMatches(listOne, listTwo, mergedList)); // merge matches should return false  // testing that the list doesn't contain "bad" matches, contains the "good" match, and contains the correct #  assert(mergedList.howManyMatches() == 5 && !mergedList.someoneAmongMatches("b", "b") && mergedList.someoneAmongMatches("c", "c"));  // testing individual points  assert(mergedList.confirmMatch(0, first, last, value) && first == "a" && last == "a" && value == "0");  assert(mergedList.confirmMatch(1, first, last, value) && first == "c" && last == "c" && value == "3");  assert(mergedList.confirmMatch(2, first, last, value) && first == "d" && last == "d" && value == "3");  assert(mergedList.confirmMatch(3, first, last, value) && first == "e" && last == "e" && value == "0");  assert(mergedList.confirmMatch(4, first, last, value) && first == "f" && last == "f" && value == "0");  // testing a mergedList on a non-empty result list  assert(mergeMatches(mergedList, listOne, listTwo)); // should return true, no bad values  assert(listTwo.howManyMatches() == 6 && listTwo.someoneAmongMatches("a", "a") && listTwo.someoneAmongMatches("c", "c") && listTwo.someoneAmongMatches("d", "d"));  // testing authenticateMatches  OnlineDating coffeeMeetsBagel;  coffeeMeetsBagel.makeMatch("Cobey", "C", "35");  coffeeMeetsBagel.makeMatch("Dan", "H", "38");  coffeeMeetsBagel.makeMatch("Dan", "V", "40");  coffeeMeetsBagel.makeMatch("Dion", "V", "45");    OnlineDating resultList;  // wild card last name  authenticateMatches("Dan", "\*", coffeeMeetsBagel, resultList);  assert(resultList.howManyMatches() == 2 && resultList.someoneAmongMatches("Dan", "H") && resultList.someoneAmongMatches("Dan", "V"));  // wild card first name  authenticateMatches("\*", "V", coffeeMeetsBagel, resultList);  assert(resultList.howManyMatches() == 2 && resultList.someoneAmongMatches("Dan", "V") && resultList.someoneAmongMatches("Dion", "V"));  // wild card both names  authenticateMatches("\*", "\*", coffeeMeetsBagel, resultList);  assert(resultList.howManyMatches() == 4);  // full name search  authenticateMatches("Dan", "H", coffeeMeetsBagel, resultList);  assert(resultList.howManyMatches() == 1 && resultList.someoneAmongMatches("Dan", "H"));  // aliasing  resultList = coffeeMeetsBagel;  authenticateMatches("Dan", "\*", coffeeMeetsBagel, resultList);  assert(resultList.howManyMatches() == 2 && resultList.someoneAmongMatches("Dan", "H") && resultList.someoneAmongMatches("Dan", "V"));  **return** 0;  } |