UID: 005738216

Project 4 Report

**Summary:**

I have finished implementing all parts of this project, including the RadixTree class and my classes do not utilise any banned components. After testing my program and running it with both Xcode and g32, to my knowledge, there are no bugs that are affecting the performance of my program. Summaries of how the functions in each class were tested are listed below.

**Testing Classes:**

**PersonProfile Class:**

The PersonProfile class was the first class that I fully implemented. After writing the functions, in main, I created multiple PersonProfile objects. After that, I called GetName() and GetEmail() on each PersonProfile to ensure that the proper string was being returned. To test AddAttValPair, I created multiple AttValPairs for each person. Since a person cannot have duplicates, I inserted some AttValPairs twice into a person profile to make sure that those duplicates weren’t kept. I was able to make sure there were no duplicates by calling my GetNumAttValPairs() function which returns the number of AttValPairs. This function was useful for testing since if I inserted 4 total AttValPairs into a person profile and 3 of them were the same, the function should return 4 and not 5. Along with that, I used the GettAttVal function to loop through all the AttValPairs of the PersonProfile and ensure that everything was properly inserted.

e.g.

PersonProfile c = PersonProfile(“Alissa”, “alissatong@ucla.edu”);

assert(c.GetName() == “alissa” && c.GetEmail() == “alissatong@ucla.edu”);

AttValPair test = AttValPair("trait", "transparent");

AttValPair test = AttValPair("job", "student");

AttValPair test = AttValPair("hobby", "baking");

AttValPair test = AttValPair("job", "student");

assert(c.GetNumAttValPairs() == 3);

for(int i = 0; i < c.GetNumAttValPairs(); i++)

{

AttValPair test;

c.GetAttVal(i, test);

std::cout << test.attribute << “ “ << test.value << std::endl;

}

**AttributeTranslator Class:**

For the AttributeTranslator class, I was ultimately able to test that the Load(std::string filename) function worked when I tested the FindCompatibleAttValPairs(const AttValPair& source) function since this function should return a vector for the appropriate compatible pairs given that the Load function properly loaded the file. Since the actual translator.txt is so long, I created a translatorTest.txt that was shorter and had fewer lines to parse. Using this tester text file, I created an AttValPair and called FindCompatibleAttValPairs on it. I then iterated through the returned vector and printed out all the attribute-value pairs to ensure that they were correct.

e.g.

AttValPair test = AttValPair("trait", "transparent");

std::vector<AttValPair> t = at.FindCompatibleAttValPairs(test);

for(int i = 0; i < t.size(); i++)

{

std::cout << t[i].attribute << " " << t[i].value << std::endl;

}

**MemberDatabase Class:**

Similarly to the AttributeTranslator class, my testing of whether the LoadDatabase(std::string filename) function worked properly was ultimately done through the testing of the other functions in the MemberDatabase class. To test the FindMatchingMembers(const AttValPair& input) function, I made a smaller text file with fewer people and used the translatorTest.txt file mentioned above to check that the proper members were being matched. I created an AttValPair and then called FindMatchingMembers. I then proceeded to loop through the vector returned by the function and print out the emails of all the people with that AttValPair. Through this, I was able to tell whether the proper people were being located or not.

e.g.

AttValPair pair = AttValPair("trait", "transparent");

std::vector<std::string> test2 = mdb.FindMatchingMembers(pair);

std::cout << test2.size();

for(int i = 0; i < test2.size(); i++)

{

std::cout << test2[i] << std::endl;

}

To test the GetMemberByEmail(std::string email) function, I simply called the function with various different emails, some that did exist in my membersTest.txt and some that did not. Since the function returns a pointer, I then checked whether the pointer was a nullptr, indicating that that email doesn’t exist, or not. If it was not a nullptr, I called GetName() to make sure the email was associated with the proper individual.

e.g.

Let’s say that all these people were in the memberTest.txt and have been loaded into the database:

Alissa Tong

alissatong@ucla.edu

0

Carey Nachenberg

climberkip@gmail.com

0

David Smallberg

das@cs.ucla.edu

0

In main, this is how I tested the function:

PersonProfile\* a = mdb.GetMemberByEmail(“alissatong@ucla.edu”);

PersonProfile\* b = mdb.GetMemberByEmail(“climberkip@gmail.com”);

PersonProfile\* c = mdb.GetMemberByEmail(“das@cs.ucla.edu”);

PersonProfile\* d = mdb.GetMemberByEmail(“cs32@cs.ucla.edu”);

if(a != nullptr)

std::cout << a->getName();

if(b != nullptr)

std::cout << b->getName();

if(c != nullptr)

std::cout << c->getName();

if(d != nullptr)

std::cout << d->getName();

**MatchMaker Class:**

For the MatchMaker class, I tested the IdentifyRankedMatches using the translatorTest.txt and membersTest.txt previously mentioned. Because these were smaller sets of data, it was easier for me to tell whether members were properly matched. I chose a member within the testing database and chose to output all members who had 3 or more matches. Since the databases were small enough, I manually matched this particular member to all the other members that had compatible AttValPairs. Thus, after I ran the program, I compared the output against my manual matching to ensure that they were the same. After I could confirm that MatchMaker, along with all my other classes ran properly with my testing text files, I ran the full program with the actual member database and translator text and compared my output against the sample program provided on the CS 32 website.

To test that the destructor worked properly and there were no memory leaks in my program, I ran my program on g32 and made sure that when my program terminated, I didn’t get any messages regarding leaked memory.

**RadixTree Class:**

For the RadixTree class, I used the TA slides from week 9 as reference for building a proper radix tree. I originally did not have a second array in my Node struct to hold strings and had a single string variable. However, the implementation was not working properly with the single string variable so I decided to use a second array instead. In order to make inserting and searching easier, I created two private functions which were insertHelper(Node\* root, std::string str, const ValueType& value) and searchHelper(Node\* root, std::string key) which were called in insert(std::string str) and search(std::string str, const ValueType& value) respectively. I first tested that these functions worked by simply inserting keys into a RadixTree and then searching for them to ensure that they were inserted properly. I tested various edge conditions which is shown in the test cases below:

RadixTree<int> tester = RadixTree<int>();

tester.insert(“warm”, 1);

tester.insert(“warmer”, 2);

tester.insert(“war”, 3);

tester.insert(“childish”, 4);

tester.insert(“child”, 5);

assert( \*(tester.search(warm)) == 1);

assert( \*(tester.search(warmer)) == 2);

assert( \*(tester.search(war)) == 3);

assert( \*(tester.search(childish)) == 4);

assert( \*(tester.search(child)) == 1);