Kelly Yu

Dr. Smallberg

3/15/22

Project 4 Report

I was able to finish the project completely and I did not use any banned STL components.

I tested my classes individually using a mix of my own test functions and test cases, as well as those from the project spec. I included the classes one at a time to test each class independently, with the exception of some classes needing a predefined object of another class type. I also used my own short list of member texts and attribute translations to target how many matches I should expect, and what the various radix trees should be holding. I called these test functions testPP(), testAT(), testMD(), testMM(), and testRadixTree() in my main function.

Person Profile Class:

|  |
| --- |
| #include "provided.h"  #include "PersonProfile.h"  #include <string>  #include <iostream>  #include <cassert>  **using** **namespace** std;  **void** testPP(){  PersonProfile a1("Carey", "snail");  PersonProfile a2("David", "gmail");  AttValPair b1("hobby,", "none");  AttValPair b3("fav movie,", "starwars");  AttValPair b4("hobby,", "np");  AttValPair b5("hack", "lol");  a1.AddAttValPair(b1);  a1.AddAttValPair(b1); //no dups  a1.AddAttValPair(b3);  a1.AddAttValPair(b4); //radix tree: hobby maps to vector w none & np  a1.AddAttValPair(b5); //radix tree: h -> obby & ack  //no dups, set up radix tree right  assert(a1.GetName() == "Carey" && a1.GetEmail() == "snail" && a1.GetNumAttValPairs() == 4);  assert(a2.GetName() == "David" && a2.GetEmail() == "gmail" && a2.GetNumAttValPairs() == 0);  AttValPair replace1("A,", "B");  assert(a1.GetAttVal(0, replace1) && replace1.attribute == "hobby," && replace1.value == "none");  //out of bounds  AttValPair replace2("C," , "D");  assert(!a1.GetAttVal(-1, replace2) && replace2.attribute == "C," && replace2.value == "D");  assert(!a1.GetAttVal(5, replace2) && replace2.attribute == "C," && replace2.value == "D");  AttValPair b2("hobby," , "lobby");  a2.AddAttValPair(b1);  a2.AddAttValPair(b2);  a2.AddAttValPair(AttValPair("fav food,", "chocolate"));  assert(a2.GetName() == "David" && a2.GetEmail() == "gmail" && a2.GetNumAttValPairs() == 3);  cerr << "Passed PP test cases!" << endl;  } |

To test this class, I created two Person Profile objects. (The strings I inserted into the functions are 100% random ☺.) I tested:

* if addAttValPair function prevents duplicate pairs by adding two identical attribute value pairs into “Carey”
* if two attribute value pairs with the same attribute but different value are both added to my vector container of attribute value pairs, & are both stored in the vector of values the attribute maps attributes to in the radix tree
* if adding a second attribute value pairs with an attribute that shares a common prefix with an existing attribute value pair’s attribute -- will create a new Radix node that links to Radix Nodes with attributes mapping to the attribute value pairs
* if GetName, GetEmail, and GetNumAttValPair functions returned the right values
* if getting an attribute value pair with indexes within bounds would replace the attvalpair that was passed in by reference, and if indexes out of bounds did not impact the attvalpair passed in

Attribute Translator Class

|  |
| --- |
| #include "AttributeTranslator.h"  #include <string>  #include <iostream>  #include <cassert>  **using** **namespace** std;  **void** listCompatiblePairs(**const** AttributeTranslator& translator, AttValPair att) {  std::vector<AttValPair> result = translator.FindCompatibleAttValPairs(att);  **if** (!result.empty()) {  std::cout << "Compatible attributes and values:" << std::endl;  **for** (**const** **auto**& p: result)  std::cout << p.attribute << " -> " << p.value << std::endl;  }  **else**  std::cout << "Empty vector" << std::endl;  }  **void** testAT(){  AttributeTranslator at;  assert(at.Load(TRANSLATOR\_FILE));  //checked duplication with double favfood translations to del taco  cout << "Test 1: " << endl;  listCompatiblePairs(at, AttValPair("favorite\_food", "del taco"));  cout << "Test 2: " << endl;  listCompatiblePairs(at, AttValPair("notAtt", "notVal"));  cerr << "Passed AT test cases!" << endl;  } |

I used my debugger to check if the radix trees the correctly mapped source attribute value pairs to compatible attribute value pairs. I checked if the source attribute value strings were correctly splitting apart when sub-prefixes, new prefixes, and new suffixes were introduced. (I explain sub-prefixes, prefixes, and suffixes later in Radix Tree Class).

I tested the Load function of Attribute Translator. I used translator text file that contained:

favorite\_food,del taco,favorite\_food,del taco

favorite\_food,del taco,favorite\_food,del taco

favorite\_food,del taco,hobby,dancing

favorite\_food,del taco,favorite\_food,mexican

favorite\_food,del taco,occupation,del taco employee

Of the 5 translations, only four are unique. I used the listCompatiblePairs to check if the output statement only contained 4 lines, one for each unique translation.

Additionally, I tested if listCompatiblePairs printed out “Empty vector” when I asked it to find translations for a source attribute value pair that was not loaded and inserted into the Radix Tree.

Member Database Class

|  |
| --- |
| #include "provided.h"  #include "PersonProfile.h"  #include "AttributeTranslator.h"  #include "MemberDatabase.h"  #include "MatchMaker.h"  #include "RadixTree.h"  #include <string>  #include <iostream>  #include <cassert>  **using** **namespace** std;  **void** testMD(){  MemberDatabase md;  assert(md.LoadDatabase(MY\_MEMBERS\_FILE));  vector<string> matchesEm;  //hobby,skiing: only 1 member has this attribute value pair  matchesEm = md.FindMatchingMembers(AttValPair("hobby", "skiing"));  assert(matchesEm.size() == 1);  //hobby, skating: 1 of 2 members has a duplicate; only 2 members  matchesEm = md.FindMatchingMembers(AttValPair("hobby", "skating"));  assert(matchesEm.size() == 2);  matchesEm = md.FindMatchingMembers(AttValPair("hobby", "eating"));  assert(matchesEm.size() == 0);  **const** PersonProfile\* pp1;  **const** PersonProfile\* pp2;  **const** PersonProfile\* pp3;  pp1 = md.GetMemberByEmail("k@gmail.com");  pp2 = md.GetMemberByEmail("p@gmail.com");  pp3 = md.GetMemberByEmail("a@gmail.com");  assert(pp1->GetName() == "Carot" && pp2->GetName() == "Parot" && pp3 == **nullptr**);  matchesEm = md.FindMatchingMembers(AttValPair("occupation", "professor"));  assert(matchesEm.size() == 2);  matchesEm = md.FindMatchingMembers(AttValPair("hobby", "pigeon racing"));  assert(matchesEm.size() == 1);  cerr << "Passed MD test cases!" << endl;  } |

I tested the LoadDatabase function by using the debugger to check if the member database’s radix trees correctly map the string emails to Person Profile objects and source attribute value pair strings to a vector of emails. I also used the debugger to check if I did have 100,000 Person Profile pointers in my vector of Person Profile pointers that point to each Person Profile, since there was 100,000 members the given members.txt file I loaded.

I tested the FindMatchingMembers function by searching for a vector of members with the attribute value in three ways.

* First, I searched for an attribute value pair that only 1 member has listed once under their name in members.txt file.
* Second, I searched for an attribute value pair that only 2 members have, but one of which list the attribute value pair twice. Since loading the member database involves creating PersonProfile objects who only store distinct attribute value pairs, I made sure to the vector returned from this test case should result in a vector of 2 items, (the emails of the 2 members).
* Third, I searched for members who have a non-existent attribute value pair, so my vector should be an empty one.

I tested the GetMemberByEmail function by seeing if the pointer to Person Profile returned indeed points to the Person Profile containing the name of the member with the email. I performed this test by finding two existent and one nonexistent member with 2valid emails and 1 invalid email.

Match Maker Class

|  |
| --- |
| #include "provided.h"  #include "AttributeTranslator.h"  #include "MemberDatabase.h"  #include "MatchMaker.h"  #include <string>  #include <iostream>  #include <cassert>  **using** **namespace** std;  **void** findMatches(**const** MatchMaker& mm, **const** std::string& member\_email, **int** threshold) {  std::vector<EmailCount> results = mm.IdentifyRankedMatches(member\_email, threshold);  **if** (results.empty())  std::cout << "We found no one who was compatible :-(" << std::endl;  **else** {  **for** (**const** **auto**& match: results) {  std::cout << match.email << " has " << match.count  << " attribute-value pairs in common with " << member\_email << std::endl;  }  }  }  **void** testMM(){  MemberDatabase mdata;  assert(mdata.LoadDatabase(MY\_MEMBERS\_FILE));  AttributeTranslator atr;  assert(atr.Load(MY\_TRANSLATOR\_FILE));  MatchMaker mm(mdata, atr);  vector<EmailCount> matches = mm.IdentifyRankedMatches("anisha@gmail.com", 2);  assert(matches.size() == 3);  **for**(**int** i=0; i<matches.size(); i++){  cout << "Emails: " << matches[i].email << " Count: " << matches[i].count << endl;  }  //no attributes  matches = mm.IdentifyRankedMatches("d@gmail.com", 2);  assert(matches.size() == 0);  matches = mm.IdentifyRankedMatches("d@gmail.com", 0);  assert(matches.size() == 9);  matches = mm.IdentifyRankedMatches("d@gmail.com", -1);  assert(matches.size() == 9);    std::cerr << "Passed MM test cases!" <<std::endl;  findMatches(mm, "anisha@gmail.com", 2);  } |

I tested the Match Maker class using the provided test case from the project specifications. My member.txt file contained these members:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Anisha  anisha@gmail.com  4  likes,cookies  likes,brownies  likes,coding  occupation,software engineer | Tjader  tjader@gmail.com  3  likes,baking  hobby,weight training  occupation,salesperson | Hercumur  hercumur@gmail.com  3  likes,baking  likes,reviewing code  occupation,QA engineer | Angus  angus@gmail.com  2  likes,baking  occupation,QA engineer | Andrea  andrea@gmail.com  3  hobby,mahjong  likes,reviewing code  occupation,QA engineer |

I checked to see if my IdentifiedRankedMatches returned the vector with the correct emails of members that met the threshold and correct counts of how many times the member is matched with the matchee. I did this by outputting the emails and counts for each matched member who passed the threshold.

I also tried searching matches for a member who has no attribute value pairs. This member is defined as:

Different

d@gmail.com

0

When the threshold was positive, this resulted in a vector of 0 as no one can possibly share any matches with that member. When the threshold was 0, this resulted in a vector of 9, as there were 9 other members in my text file, and all 9 members have at least 0 (0 or more) matches with the member. Similarly, this happens again when the threshold is negative.

Radix Tree Class

|  |
| --- |
| #include "RadixTree.h"  #include <string>  #include <iostream>  #include <cassert>  **using** **namespace** std;  **void** testRadixTree(){  RadixTree<std::string> strToStr1;  strToStr1.insert("test", "A"); //added new key  strToStr1.insert("slow", "B"); //added new key  strToStr1.insert("water", "C"); //added new key  strToStr1.insert("slower", "D"); //added new suffix  strToStr1.insert("team", "E"); //broke prefix, added suffix  // std::cerr << "--------------------" << std::endl;  assert(strToStr1.search("test") != **nullptr**); //word found  assert(strToStr1.search("te") == **nullptr**); //word == prefix to other words  assert(strToStr1.search("slo") == **nullptr**); //word == part of existing word  assert(strToStr1.search("tick") == **nullptr**); //mismatch found  assert(strToStr1.search("") == **nullptr**); //no keys are empty (but root who's not a valid key)  RadixTree<std::string> strToStr2;  // std::cerr <<"---------------adding suffixes test--------------------" << std::endl;  strToStr2.insert("sand", "A"); //add sand new key  strToStr2.insert("sandbox", "B"); //add box suffix  strToStr2.insert("race", "D"); //add race new key  strToStr2.insert("racecar", "E"); //add car suffix  strToStr2.insert("racecarman", "F"); //add man suffix  strToStr2.insert("racecarmans", "G"); //add s suffix  //test adding new prefix  // std::cerr <<"---------------adding prefixes test--------------------" << std::endl;  strToStr2.insert("tester", "A");  strToStr2.insert("test", "B");  strToStr2.insert("sky", "D");  strToStr2.insert("skydiver", "E");  strToStr2.insert("skydive", "F");  //test breaking prefix + adding suffix!  // std::cerr <<"-------breaking prefixes & adding suffixes test---------" << std::endl;  // std::cerr << "----- existing key is shorter than new key -----" << std::endl;  strToStr2.insert("saxon", "A"); //add new key: saxon  strToStr2.insert("saxophone", "B"); //break prefix: saxo + n add:suffix phone  // std::cerr << "--" << std::endl;  strToStr2.insert("Jaz", "D"); //add new key: Jaz  strToStr2.insert("Jazmin", "E"); //add suffix: min  strToStr2.insert("Jazminee", "F"); //add suffix: ee  strToStr2.insert("Jazminean", "G"); //break prefix: e + e add suffix: an  // std::cerr << "----- existing key is longer than new key -----" << std::endl;  strToStr2.insert("peter", "A"); //add new key: peter  strToStr2.insert("pets", "B"); //break prefix: pet + er add suffix: s  // std::cerr << "--" << std::endl;  strToStr2.insert("Peter", "D"); //add new key: Peter  strToStr2.insert("PeterPane", "E"); //add suffix: Pane  strToStr2.insert("PeterPen", "F"); //break prefix: P + ane add suffix: en  // std::cerr << "----- existing key is same length as new key -----" << std::endl;  // strToStr2.insert("son", "A"); //add new key: son  // strToStr2.insert("sun", "B"); //break prefix: s + on add suffix: un  strToStr2.insert("test", "A"); //add new key: test  strToStr2.insert("team", "B"); //break prefix: te + st add suffix: am  // std::cerr << "--" << std::endl;  strToStr2.insert("apple", "D"); //add new key: apple  strToStr2.insert("applebig", "E"); //add suffix: big  strToStr2.insert("applebid", "F"); //break prefix: bi + g add suffix: d  string\* s = strToStr2.search("applebid");  assert(\*s == "F");  strToStr2.insert("applebid", "changed");  s = strToStr2.search("applebid");  assert(\*s == "changed");  cerr << "Radix Tree test cases passed!" << endl;  } |

I tested my radix tree using a mix of UPE’s example radix trees and a series of my own radix trees. The first 5 inserts were examples from UPE’s project hack session.

For my search function, I tested if they I was trying to find my RadixTree was either an existing key, a prefix for other keys, part of existing keys, and not an existing key at all. I also tested the function with an empty key, which should return nullptr since there should not be any empty keys.

For my insert function, I split my tests into 5 sets. For each set, I have 2 subsets, one that involved one traversal through the nodes, and another one that involved 2 or more to make sure that I can move down the levels of the radix tree. Additionally, I had std::cerr statements that printed out statements like “**Added a new suffix!**” and “**Broke prefix with part 1: te part 2: st**” so that I could see the right sections of the function were called on.

For the first set, I tested for adding a key in which part of the key is an existing key already in the radix tree. Therefore, the “prefix” of the new key is already in the tree, and I must add a “suffix”.

For the second set, I tested for adding a key in that is altogether part of an existing key. Therefore, I have to insert a node in between the parent of the existing key and the existing key to insert the new key as a new “prefix” and remove the “prefix” portion from my existing key.

For the third, fourth, and fifth sets, I tested for adding a new key who only shares a portion of characters with an existing key. This requires me to split a node into two, with the first portion containing only the characters that the new key and existing key shared. The second portion contains the remaining characters of the existing key after the common characters and contains the chain of nodes rooted at the existing key’s node. Lastly, I also have to add a new node that contains the remainder of the new key’s characters that did not match with the existing key.

Particularly for the third set, I used test cases where the existing key is shorter than the new key. For the fourth set, I used test cases where the existing key is longer than the new key. For the fifth set, I used test cases where the existing key is of equal length with the new key.

Lastly, I also tested if inserting a duplicate key with a new value will update the value that the key maps to.